Dive into Deep Learning

Release 0.17.1

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Dec 12, 2021

Contents

Prefácio 1

Instalação 9

Notação 13

1 Introdução 17

1.1 Um exemplo motivador . . . 18
1.2 Componentes chave . . . 20
1.3 Tipos de Problemas de *Machine Learning* . . . 23
1.4 Raízes . . . 35
1.5 O Caminho Para o *Deep Learning* . . . 37
1.6 Histórias de Sucesso . . . 40
1.7 Características . . . 42
1.8 Resumo . . . 43
1.9 Exercícios . . . 43

2 Preliminares 45

2.1 Manipulação de Dados . . . 46
2.1.1 Iniciando . . . 46
2.1.2 Operações . . . 48
2.1.3 Mecanismo de *Broadcasting* . . . 50
2.1.4 Indexação e Fatiamento . . . 51
2.1.5 Economizando memória . . . 52
2.1.6 Conversão para outros objetos Python . . . 53
2.1.7 Sumário . . . 53
2.1.8 Exercícios . . . 53
2.2 Pré-processamento de Dados . . . 53
2.2.1 Lendo o *Dataset* . . . 54
2.2.2 Lidando com Dados Faltantes . . . 54
2.2.3 Convertendo para o Formato Tensor . . . 55
2.2.4 Sumário . . . 56
2.2.5 Exercícios . . . 56
2.3 Álgebra Linear . . . 56
2.3.1 Escalares . . . 56
2.3.2 Vetores . . . 57
2.3.3 Matrizes . . . 58
2.3.4 Tensores . . . 60
2.3.5 Propriedades Básicas de Aritmética de Tensores . . . 61
2.3.6 Redução . . . 62
2.3.7 Produto Escalar . . . 64

2.3.8 Produtos Matriz-Vetor . . . 65
2.3.9 Multiplicação Matriz Matriz . . . 65
2.3.10 Normas . . . 66
2.3.11 Mais sobre Algebra Linear . . . 68
2.3.12 Sumário . . . 68
2.3.13 Exercícios . . . 69
2.4 Cálculo . . . 69
2.4.1 Derivadas e Diferenciação . . . 70
2.4.2 Derivadas Parciais . . . 74
2.4.3 Gradientes . . . 74
2.4.4 Regra da Cadeia . . . 74
2.4.5 Sumário . . . 75
2.4.6 Exercícios . . . 75
2.5 Diferenciação automática . . . 75
2.5.1 Um exemplo simples . . . 76
2.5.2 Retroceder para variáveis não escalares . . . 77
2.5.3 Computação *Detaching* . . . 77
2.5.4 Computando o Gradiente do *Python Control Flow* . . . 78
2.5.5 Sumário . . . 79
2.5.6 Exercícios . . . 79
2.6 Probabilidade . . . 79
2.6.1 Teoria Básica de Probabilidade . . . 81
2.6.2 Lidando com Múltiplas Variáveis Aleatórias . . . 84
2.6.3 Expectativa e Variância . . . 87
2.6.4 Sumário . . . 88
2.6.5 Exercícios . . . 88
2.7 Documentação . . . 88
2.7.1 Encontrando Todas as Funções e Classes em um Módulo . . . 88
2.7.2 Buscando o Uso de Funções e Classes Específicas . . . 89
2.7.3 Sumário . . . 90
2.7.4 Exercícios . . . 90

3 Linear Neural NetworkRedes Neurais Lineares 91
3.1 Linear Regression . . . 91
3.1.1 Elementos Básicos de Regressão Linear . . . 91
3.1.2 Vetorização para Velocidade . . . 96
3.1.3 A Distribuição Normal e Perda Quadrada . . . 97
3.1.4 Da Regressão Linear às Redes Profundas . . . 99
3.2 Linear Regression Implementation from Scratch . . . 102
3.2.1 Gerando o Dataset . . . 102
3.2.2 Lendo o *Dataset* . . . 103
3.2.3 Initializing Model Parameters . . . 105
3.2.4 Definindo o Modelo . . . 105
3.2.5 Definindo a Função de Perda . . . 105
3.2.6 Definindo o Algoritmo de Otimização . . . 106
3.2.7 Treinamento . . . 106
3.2.8 Resumo . . . 107
3.2.9 Exercícios . . . 108
3.3 Implementação Concisa de Regressão Linear . . . 108
3.3.1 Gerando the Dataset . . . 108
3.3.2 Lendo o Dataset . . . 109

3.3.3 Definindo o Modelo . . . 110
3.3.4 Inicializando os Parâmetros do Modelo . . . 110
3.3.5 Definindo a Função de Perda . . . 111
3.3.6 Definindo o Algoritmo de Otimização . . . 111
3.3.7 Treinamento . . . 111
3.3.8 Resumo . . . 112
3.3.9 Exercícios . . . 112
3.4 Regressão *Softmax* . . . 113
3.4.1 Problema de Classificação . . . 113
3.4.2 Arquitetura de Rede . . . 114
3.4.3 Custo de Parametrização de Camadas Totalmente Conectadas . . . 114
3.4.4 Operação do *Softmax* . . . 115
3.4.5 Vetorização para *Minibatches* . . . 116
3.4.6 Função de Perda . . . 116
3.4.7 Fundamentos da Teoria da Informação . . . 118
3.4.8 Predição do Modelo e Avaliação . . . 119
3.4.9 Resumo . . . 119
3.4.10 Exercícios . . . 119
3.5 O *Dataset* de Classificação de Imagens . . . 120
3.5.1 Lendo o Dataset . . . 120
3.5.2 Lendo um *Minibatch* . . . 122
3.5.3 Juntando Tudo . . . 122
3.5.4 Resumo . . . 123
3.5.5 Exercícios . . . 123
3.6 Implementação da Regressão *Softmax* do Zero . . . 123
3.6.1 Inicializando os Parâmetros do Modelo . . . 124
3.6.2 Definindo a Operação do *Softmax* . . . 124
3.6.3 Definindo o Modelo . . . 125
3.6.4 Definindo a Função de Perda . . . 125
3.6.5 Exatidão da Classificação . . . 126
3.6.6 Treinamento . . . 127
3.6.7 Predição . . . 130
3.6.8 Resumo . . . 131
3.6.9 Exercícios . . . 131
3.7 Implementação Concisa da Regressão *Softmax* . . . 131
3.7.1 Inicializando os Parâmetros do Modelo . . . 132
3.7.2 Implementação do *Softmax* Revisitada . . . 132
3.7.3 Otimização do Algoritmo . . . 133
3.7.4 Trainamento . . . 133
3.7.5 Resumo . . . 134
3.7.6 Exercícios . . . 134

4 *Perceptrons* Multicamada 135
4.1 *Perceptrons* Multicamada . . . 135
4.1.1 Camadas Ocultas . . . 135
4.1.2 Funções de Ativação . . . 138
4.1.3 Resumo . . . 143
4.1.4 Exercícios . . . 143
4.2 Implementação de Perceptrons Multicamadas do Zero . . . 144
4.2.1 Inicializando os Parâmetros do Modelo . . . 144
4.2.2 Função de Ativação . . . 144

4.2.3 Modelo . . . 145
4.2.4 Função de Perda . . . 145
4.2.5 Trainamento . . . 145
4.2.6 Resumo . . . 146
4.2.7 Exercícios . . . 146
4.3 Implementação Concisa de *Perceptrons* Multicamadas . . . 147
4.3.1 Modelo . . . 147
4.3.2 Resumo . . . 148
4.3.3 Exercícios . . . 148
4.4 Seleção do Modelo, *Underfitting*, e *Overfitting* . . . 148
4.4.1 Erro de Treinamento e Erro de Generalização . . . 149
4.4.2 Seleção do Modelo . . . 152
4.4.3 *Underfitting* ou *Overfitting*? . . . 153
4.4.4 Regressão Polinomial . . . 154
4.4.5 Resumo . . . 158
4.4.6 Exercícios . . . 159
4.5 *Weight Decay* . . . 159
4.5.1 Normas e *Weight Decay* . . . 160
4.5.2 Regressão Linear de Alta Dimensão . . . 161
4.5.3 Implementação do Zero . . . 162
4.5.4 Implementação Concisa . . . 164
4.5.5 Resumo . . . 166
4.5.6 Exercícios . . . 166
4.6 *Dropout* . . . 166
4.6.1 *Overfitting* Revisitado . . . 167
4.6.2 Robustez por Meio de Perturbações . . . 167
4.6.3 *Dropout* na Prática . . . 168
4.6.4 Implementação do Zero . . . 169
4.6.5 Implementação Concisa . . . 171
4.6.6 Resumo . . . 172
4.6.7 Exercícios . . . 173
4.7 Propagação Direta, Propagação Reversa e Gráficos Computacionais . . . 173
4.7.1 Propagação Direta . . . 174
4.7.2 Gráfico Computatcional de Propagação Direta . . . 174
4.7.3 Propagação Reversa . . . 175
4.7.4 Treinando Redes Neurais . . . 176
4.7.5 Resumo . . . 177
4.7.6 Exercícios . . . 177
4.8 Estabilidade Numérica e Inicialização . . . 177
4.8.1 Explosão e Desaparecimento de Gradientes . . . 178
4.8.2 Inicialização de Parâmetros . . . 180
4.8.3 Resumo . . . 182
4.8.4 Exercícios . . . 182
4.9 Mudança de Ambiente e Distribuição . . . 183
4.9.1 Tipos de Turno de Distribuição . . . 183
4.9.2 Exemplos de Mudança de Distribuição . . . 186
4.9.3 Correção de Mudança de Distribuição . . . 187
4.9.4 Uma taxonomia de Problemas de Aprendizagem . . . 191
4.9.5 Justiça, Responsabilidade e Transparência no *Machine Learning* . . . 192
4.9.6 Resumo . . . 193
4.9.7 Exercícios . . . 194

4.10 Previsão de Preços de Imóveis no Kaggle 194
4.10.1 *Download* e *Cache* de *datasets* 194
4.10.2 *Kaggle* 196
4.10.3 Acessando e Lendo o Conjunto de Dados 197
4.10.4 Pré-processamento de Dados 198
4.10.5 Treinamento 199
4.10.6 Validação Cruzada K-Fold 200
4.10.7 Seleção de Modelo 201
4.10.8 Enviando Previsões no Kaggle 202
4.10.9 Resumo 204
4.10.10 Exercícios 204

5 *Deep Learning* Computacional **207**
5.1 Camadas e Blocos 207
5.1.1 Um Bloco Personalizado 209
5.1.2 O Bloco Sequencial 211
5.1.3 Execução de Código na Função de Propagação Direta 212
5.1.4 Eficiência 213
5.1.5 Sumário 213
5.1.6 Exercícios 214
5.2 Gerenciamento de Parâmetros 214
5.2.1 Acesso a Parâmetros 215
5.2.2 Inicialização de Parâmetros 217
5.2.3 Parâmetros *Tied* 219
5.2.4 Sumário 220
5.2.5 Exercícios 220
5.3 Camadas Personalizadas 220
5.3.1 Camadas Sem Parâmetros 221
5.3.2 Camadas com Parâmetros 221
5.3.3 Sumário 223
5.3.4 Exercícios 223
5.4 Entrada e Saída de Arquivos 223
5.4.1 Carregando e Salvando Tensores 223
5.4.2 Carregando e Salvando Parâmetros de Modelos 224
5.4.3 Sumário 225
5.4.4 Exercícios 225
5.5 GPUs 226
5.5.1 Dispositivos Computacionais 227
5.5.2 Tensores e GPUs 228
5.5.3 Redes Neurais e GPUs 230
5.5.4 Sumário 231
5.5.5 Exercícios 231

6 Convolutional Neural Networks **233**
6.1 De Camadas Totalmente Conectadas às Convoluções 234
6.1.1 Invariância 234
6.1.2 Restringindo o MLP 235
6.1.3 Convoluções 237
6.1.4 “Onde está Wally” Revisitado 237
6.1.5 Resumo 238
6.1.6 Exercícios 239

6.2 Convolução para Imagens . . . 239
6.2.1 A Operação de Correlação Cruzada . . . 239
6.2.2 Camadas Convolucionais . . . 241
6.2.3 Detecção de Borda de Objeto em Imagens . . . 241
6.2.4 Aprendendo um Kernel . . . 242
6.2.5 Correlação Cruzada e Convolução . . . 243
6.2.6 Mapa de Características e Campo Receptivo . . . 244
6.2.7 Resumo . . . 244
6.2.8 Exercícios . . . 245
6.3 Preenchimento e Saltos . . . 245
6.3.1 Preenchimento . . . 246
6.3.2 Saltos . . . 247
6.3.3 Resumo . . . 249
6.4 Canais de Múltiplas Entradas e Saídas . . . 249
6.4.1 Canais de Entrada Múltiplos . . . 249
6.4.2 Canais de Saída Múltiplos . . . 251
6.4.3 Camada Convolucional 1×1 . . . 252
6.4.4 Resumo . . . 253
6.4.5 Exercícios . . . 253
6.5 *Pooling* . . . 254
6.5.1 *Pooling* Máximo e *Pooling* Médio . . . 254
6.5.2 Preenchimento e Passos . . . 256
6.5.3 Canais Múltiplos . . . 257
6.5.4 Resumo . . . 258
6.5.5 Exercícios . . . 258
6.6 Redes Neurais Convolucionais (LeNet) . . . 259
6.6.1 LeNet . . . 259
6.6.2 Trainamento . . . 262
6.6.3 Resumo . . . 264
6.6.4 Exercícios . . . 264

7 Modern Convolutional Neural Networks **265**
7.1 Redes Neurais Convolucionais Profundas (AlexNet) . . . 265
7.1.1 Representaçao do Aprendizado . . . 266
7.1.2 AlexNet . . . 269
7.1.3 Lendo o Dataset . . . 272
7.1.4 Treinamento . . . 272
7.1.5 Sumário . . . 273
7.1.6 Exercício . . . 273
7.2 Redes Usando Blocos (VGG) . . . 274
7.2.1 VGG Blocks . . . 274
7.2.2 Camadas VGG . . . 275
7.2.3 Treinamento . . . 276
7.2.4 Sumário . . . 277
7.2.5 Exercícios . . . 277
7.3 Network in Network (NiN) . . . 278
7.3.1 Blocos NiN . . . 278
7.3.2 Modelo NiN . . . 280
7.3.3 Treinamento . . . 281
7.3.4 Sumário . . . 281
7.3.5 Exercícios . . . 282

7.4 Redes com Concatenações Paralelas (GoogLeNet) . . . 282
7.4.1 Inception Blocks . . . 282
7.4.2 Modelo GoogLeNet . . . 284
7.4.3 Treinamento . . . 286
7.4.4 Sumário . . . 286
7.4.5 Exercícios . . . 287
7.5 Normalização de Lotes . . . 287
7.5.1 Treinando Redes Profundas . . . 287
7.5.2 Camadas de Normalização de Lotes . . . 289
7.5.3 Implementação do Zero . . . 290
7.5.4 Aplicando Normalização de Lotes em LeNet . . . 292
7.5.5 Implementação Concisa . . . 293
7.5.6 Controvérsia . . . 294
7.5.7 Sumário . . . 294
7.5.8 Exercícios . . . 295
7.6 Redes Residuais (ResNet) . . . 295
7.6.1 Classes Função . . . 295
7.6.2 Blocos Residuais . . . 297
7.6.3 Modelo ResNet . . . 299
7.6.4 Treinamento . . . 302
7.6.5 Sumário . . . 302
7.6.6 Exercícios . . . 303
7.7 Redes Densamente Conectadas (DenseNet) . . . 303
7.7.1 De ResNet para DenseNet . . . 303
7.7.2 Blocos Densos . . . 304
7.7.3 Camadas de Transição . . . 305
7.7.4 Modelo DenseNet . . . 306
7.7.5 Treinamento . . . 307
7.7.6 Sumário . . . 307
7.7.7 Exercícios . . . 307

8 Redes Neurais Recorrentes **309**
8.1 Modelos Sequenciais . . . 310
8.1.1 Ferramentas Estatísticas . . . 311
8.1.2 Trainamento . . . 313
8.1.3 Predição . . . 315
8.1.4 Resumo . . . 317
8.1.5 Exercícios . . . 318
8.2 Preprocessamento de Texto . . . 318
8.2.1 Lendo o Dataset . . . 319
8.2.2 Tokenização . . . 319
8.2.3 Vocabulário . . . 320
8.2.4 Juntando Todas as Coisas . . . 321
8.2.5 Resumo . . . 322
8.2.6 Exercícios . . . 322
8.3 Modelos de Linguagem e o *Dataset* . . . 322
8.3.1 Aprendendo um Modelo de Linguagem . . . 323
8.3.2 Modelos de Markov e n-gramas . . . 324
8.3.3 Estatísticas de Linguagem Natural . . . 325
8.3.4 Leitura de Dados de Longa Sequência . . . 327
8.3.5 Resumo . . . 330

8.3.6 Exercícios . . . 331
8.4 Redes Neurais Recorrentes (RNNs) . . . 331
8.4.1 Redes Neurais sem Estados Ocultos . . . 332
8.4.2 Redes Neurais Recorrentes com Estados Ocultos . . . 332
8.4.3 Modelos de Linguagem em Nível de Caracteres Baseados em RNN . . . 334
8.4.4 Perplexidade . . . 335
8.4.5 Resumo . . . 336
8.4.6 Exercícios . . . 336
8.5 Implementação de Redes Neurais Recorrentes do Zero . . . 337
8.5.1 Codificação One-Hot . . . 337
8.5.2 Inicializando os Parâmetros do Modelo . . . 338
8.5.3 Modelo RNN . . . 338
8.5.4 Predição . . . 339
8.5.5 Recorte de Gradiente . . . 340
8.5.6 Treinamento . . . 341
8.5.7 Resumo . . . 344
8.5.8 Exercícios . . . 344
8.6 Implementação Concisa de Redes Neurais Recorrentes . . . 345
8.6.1 Definindo o Modelo . . . 345
8.6.2 Treinamento e Previsão . . . 347
8.6.3 Resumo . . . 348
8.6.4 Exercícios . . . 348
8.7 Retropropagação ao Longo do Tempo . . . 348
8.7.1 Análise de Gradientes em RNNs . . . 349
8.7.2 Retropropagação ao Longo do Tempo em Detalhes . . . 351
8.7.3 Resumo . . . 354
8.7.4 Exercícios . . . 354

9 Redes Neurais Recorrentes Modernas 355
9.1 Gated Recurrent Units (GRU) . . . 355
9.1.1 Estado Oculto Fechado . . . 356
9.1.2 Implementação do zero . . . 359
9.1.3 Implementação concisa . . . 361
9.1.4 Sumário . . . 362
9.1.5 Exercícios . . . 362
9.2 Memória Longa de Curto Prazo (LSTM) . . . 363
9.2.1 Célula de Memória Bloqueada . . . 363
9.2.2 Implementação do zero . . . 366
9.2.3 Implementação concisa . . . 368
9.2.4 Resumo . . . 369
9.2.5 Exercícios . . . 369
9.3 Redes neurais recorrentes profundas . . . 370
9.3.1 Dependência Funcional . . . 370
9.3.2 Implementação Concisa . . . 371
9.3.3 Treinamento e Predição . . . 372
9.3.4 Sumário . . . 372
9.3.5 Exercícios . . . 372
9.4 Redes Neurais Recorrentes Bidirecionais . . . 373
9.4.1 Programação dinâmica em modelos de Markov ocultos . . . 373
9.4.2 Modelo Bidirecional . . . 376
9.4.3 Treinar um RNN bidirecional para uma aplicação errada . . . 377

9.4.4 Sumário . . . 378
9.4.5 Exercícios . . . 379
9.5 Tradução Automática e o Conjunto de Dados . . . 379
9.5.1 Download e Pré-processamento do Conjunto de Dados . . . 380
9.5.2 Tokenização . . . 381
9.5.3 Vocabulário . . . 382
9.5.4 Carregando o Conjunto de Dados . . . 382
9.5.5 Juntando todas as coisas . . . 383
9.5.6 Sumário . . . 384
9.5.7 Exercícios . . . 384
9.6 Arquitetura Encoder-Decoder . . . 385
9.6.1 Encoder . . . 385
9.6.2 Decoder . . . 386
9.6.3 Somando o Encoder e o Decoder . . . 386
9.6.4 Sumário . . . 387
9.6.5 Exercícios . . . 387
9.7 Aprendizado Sequência a Sequência . . . 387
9.7.1 Encoder . . . 388
9.7.2 Decoder . . . 390
9.7.3 Função de Perdas . . . 392
9.7.4 Treinamento . . . 393
9.7.5 Predição . . . 395
9.7.6 Avaliação de Sequências Preditas . . . 396
9.7.7 Sumário . . . 397
9.7.8 Exercícios . . . 397
9.8 Pesquisa de feixe . . . 398
9.8.1 Busca Gulosa . . . 398
9.8.2 Busca Exaustiva . . . 399
9.8.3 Busca de Feixe . . . 400
9.8.4 Sumário . . . 401
9.8.5 Exercícios . . . 401

10 Mecanismos de Atenção **403**
10.1 Dicas para atenção . . . 403
10.1.1 Dicas de Atenção em Biologia . . . 404
10.1.2 Consultas, Chaves e Valores . . . 405
10.1.3 Visualização da Atenção . . . 406
10.1.4 Resumo . . . 407
10.1.5 Exercícios . . . 408
10.2 *Pooling* de Atenção: Regressão de Kernel de Nadaraya-Watson . . . 408
10.2.1 Gerando o Dataset . . . 408
10.2.2 *Pooling* Médio . . . 409
10.2.3 *Pooling* de Atenção não-Paramétrico . . . 410
10.2.4 *Pooling* de Atenção Paramétrica . . . 412
10.2.5 Resumo . . . 415
10.2.6 Exercícios . . . 415
10.3 Funções de Pontuação de Atenção . . . 416
10.3.1 Operação *Softmax* Mascarada . . . 417
10.3.2 Atenção Aditiva . . . 418
10.3.3 Atenção de Produto Escalar em Escala . . . 419
10.3.4 Resumo . . . 421

10.3.5 Exercícios . . . 421
10.4 Atenção de Bahdanau . . . 421
10.4.1 Modelo . . . 422
10.4.2 Definindo o Decodificador com Atenção . . . 422
10.4.3 Treinamento . . . 424
10.4.4 Resumo . . . 426
10.4.5 Exercícios . . . 426
10.5 Atenção Multi-Head . . . 426
10.5.1 Modelo . . . 427
10.5.2 Implementação . . . 428
10.5.3 Resumo . . . 430
10.5.4 Exercícios . . . 430
10.6 Autoatenção e Codificação Posicional . . . 430
10.6.1 Autoatenção . . . 430
10.6.2 Comparando CNNs, RNNs e Autoatenção . . . 431
10.6.3 Codificação Posicional . . . 433
10.6.4 Resumo . . . 436
10.6.5 Exercícios . . . 436
10.7 Transformador . . . 436
10.7.1 Modelo . . . 436
10.7.2 Redes *Positionwise Feed-Forward* . . . 438
10.7.3 Conexão residual e normalização de camada . . . 439
10.7.4 *Encoder* . . . 440
10.7.5 *Decoder* . . . 441
10.7.6 Treinamento . . . 444
10.7.7 Resumo . . . 447
10.7.8 Exercícios . . . 447

11 Algoritmos de Otimização **449**
11.1 Optimização e Deep Learning . . . 449
11.1.1 Objetivos da Optimização . . . 450
11.1.2 Desafios de otimização em Deep Learning . . . 451
11.1.3 Sumário . . . 454
11.1.4 Exercícios . . . 455
11.2 Convexidade . . . 455
11.2.1 Definições . . . 456
11.2.2 Propriedades . . . 459
11.2.3 Restrições . . . 461
11.2.4 Sumário . . . 463
11.2.5 Exercícios . . . 463
11.3 Gradiente descendente . . . 464
11.3.1 Gradiente descendente em uma dimensão . . . 464
11.3.2 Gradiente descendente multivariado . . . 468
11.3.3 Métodos Adaptativos . . . 469
11.3.4 Sumário . . . 473
11.3.5 Exercícios . . . 474
11.4 Gradiente Descendente Estocástico . . . 474
11.4.1 Atualizações de gradiente estocástico . . . 474
11.4.2 Taxa de aprendizagem dinâmica . . . 476
11.4.3 Análise de convergência para objetivos convexos . . . 478
11.4.4 Gradientes estocásticos e amostras finitas . . . 479

11.4.5 Sumário . . . 480
11.4.6 Exercícios . . . 480
11.5 Gradiente Estocástico Descendente Minibatch . . . 481
11.5.1 Vetorização e caches . . . 481
11.5.2 Minibatches . . . 483
11.5.3 Lendo o conjunto de dados . . . 484
11.5.4 Implementação do zero . . . 485
11.5.5 Implementação concisa . . . 488
11.5.6 Sumário . . . 489
11.5.7 Exercícios . . . 490
11.6 Momentum . . . 490
11.6.1 Fundamentos . . . 490
11.6.2 Experimentos Práticos . . . 495
11.6.3 Análise teórica . . . 498
11.6.4 Sumário . . . 500
11.6.5 Exercícios . . . 500
11.7 Adagrad . . . 501
11.7.1 Recursos esparsos e taxas de aprendizado . . . 501
11.7.2 Precondicionamento . . . 502
11.7.3 O Algoritmo . . . 503
11.7.4 Implementação do zero . . . 505
11.7.5 Implementação concisa . . . 505
11.7.6 Sumário . . . 506
11.7.7 Exercícios . . . 506
11.8 RMSProp . . . 507
11.8.1 O Algoritmo . . . 507
11.8.2 Implementação do zero . . . 508
11.8.3 Implementação concisa . . . 510
11.8.4 Sumário . . . 511
11.8.5 Exercícios . . . 511
11.9 Adadelta . . . 511
11.9.1 O Algoritmo . . . 511
11.9.2 Implementação . . . 512
11.9.3 Sumário . . . 513
11.9.4 Exercícios . . . 514
11.10 Adam . . . 514
11.10.1 O Algoritmo . . . 514
11.10.2 Implementação . . . 515
11.10.3 Yogi . . . 517
11.10.4 Sumário . . . 518
11.10.5 Exercícios . . . 518
11.11 Programação da taxa de aprendizagem . . . 519
11.11.1 Problema Amostra . . . 519
11.11.2 Agendadores . . . 522
11.11.3 Políticas . . . 523
11.11.4 Sumário . . . 528
11.11.5 Exercícios . . . 529

12 Desempenho Computacional **531**
12.1 Compiladores e Interpretadores . . . 531
12.1.1 Programação Simbólica . . . 532

12.1.2 Programação Híbrida 533
12.1.3 Híbrido-Sequencial 534
12.2 Computação Assíncrona 536
12.2.1 Assincronismo via *Back-end* 536
12.2.2 Barreiras e Bloqueadores 538
12.2.3 Melhorando a Computação 539
12.2.4 Melhorando o *Footprint* de Memória 539
12.2.5 Resumo 540
12.2.6 Exercícios 540
12.3 Paralelismo Automático 541
12.3.1 Computação Paralela em GPUs 541
12.3.2 Computação Paralela e Comunicação 542
12.3.3 Resumo 544
12.3.4 Exercícios 545
12.4 *Hardware* 545
12.4.1 Computadores 546
12.4.2 Memória 547
12.4.3 Armazenamento 548
12.4.4 CPUs 549
12.4.5 GPUs e outros Aceleradores 554
12.4.6 Redes e Barramentos 556
12.4.7 Resumo 557
12.4.8 Mais Números de Latência 557
12.4.9 Exercícios 558
12.5 Treinamento em Várias GPUs 559
12.5.1 Dividindo o Problema 560
12.5.2 Paralelismo de Dados 561
12.5.3 Uma Rede Exemplo 563
12.5.4 Sincronização de Dados 564
12.5.5 Distribuindo Dados 565
12.5.6 Treinamento 566
12.5.7 Experimento 567
12.5.8 Resumo 568
12.5.9 Exercícios 568
12.6 Implementação Concisa para Várias GPUs 568
12.6.1 Uma Rede de Exemplo 569
12.6.2 Inicialização de Parâmetros e Logística 569
12.6.3 Treinamento 570
12.6.4 Experimentos 571
12.6.5 Resumo 572
12.6.6 Exercícios 572
12.7 Servidores de Parâmetros 572
12.7.1 Treinamento Paralelo de Dados 573
12.7.2 Sincronização em Anel 575
12.7.3 Treinamento Multi-Máquina 578
12.7.4 Armazenamento de (key,value) 580
12.7.5 Resumo 581
12.7.6 Exercícios 581

13 Visão Computacional **583**
13.1 Aumento de Imagem 583

13.1.1 Método Comum de Aumento de Imagem 584
13.1.2 Usando um Modelo de Treinamento de Aumento de Imagem 588
13.1.3 Resumo 591
13.1.4 Exercícios 591
13.2 Ajustes 591
13.2.1 Reconhecimento de Cachorro-quente 593
13.2.2 Resumo 597
13.2.3 Exercícios 597
13.3 Detecção de Objetos e Caixas Delimitadoras 598
13.3.1 Caixa Delimitadora 599
13.3.2 Resumo 600
13.3.3 Exercícios 600
13.4 Caixas de Âncora 601
13.4.1 Gerando Várias Caixas de Âncora 601
13.4.2 Interseção sobre União 604
13.4.3 Rotulagem de Treinamento para Definir Caixas de Âncora 605
13.4.4 Caixas Delimitadoras para Previsão 610
13.4.5 Resumo 613
13.4.6 Exercícios 613
13.5 Detecção de Objetos Multiescala 614
13.5.1 Resumo 617
13.5.2 Exercícios 617
13.6 O *Dataset* de Detecção de Objetos 617
13.6.1 Baixando Dataset 617
13.6.2 Lendo o Dataset 618
13.6.3 Demonstração 619
13.6.4 Resumo 620
13.6.5 Exercícios 620
13.7 Detecção *Single Shot Multibox* (SSD) 620
13.7.1 Modelo 620
13.7.2 Treinamento 626
13.7.3 Predição 628
13.7.4 Resumo 629
13.7.5 Exercícios 630
13.8 Region-based CNNs (R-CNNs) 632
13.8.1 R-CNNs 632
13.8.2 Fast R-CNN 633
13.8.3 R-CNN Mais Rápido 635
13.8.4 Máscara R-CNN 636
13.8.5 Resumo 637
13.8.6 Exercícios 637
13.9 Segmentação Semântica e o *Dataset* 637
13.9.1 Segmentação de Imagem e Segmentação detion and Instâancia 638
13.9.2 O Conjunto de Dados de Segmentação Semântica Pascal VOC2012 638
13.9.3 The Pascal VOC2012 Semantic Segmentation Dataset 638
13.9.4 Resumo 644
13.9.5 Exercícioises 644
13.10 Convolução Transposta 644
13.10.1 Convolução Transposta 2D Básica 645
13.10.2 Preenchimento, Passos e Canais 646
13.10.3 Analogia à Transposição de Matriz 646

13.10.4 Resumo . . . 648
13.10.5 Exercícios . . . 648
13.11 Redes Totalmente Convolucionais (*Fully Convolutional Networks*, FCN) . . . 648
13.11.1 Construindo um Modelo . . . 649
13.11.2 Inicializando a Camada de Convolução Transposta . . . 650
13.11.3 Lendo o *Dataset* . . . 652
13.11.4 Treinamento . . . 652
13.11.5 Predição . . . 653
13.11.6 Resumo . . . 654
13.11.7 Exercícios . . . 655
13.12 Transferência de Estilo Neural . . . 655
13.12.1 Técnica . . . 656
13.12.2 Lendo o Conteúdo e as Imagens de Estilo . . . 657
13.12.3 Pré-processamento e Pós-processamento . . . 658
13.12.4 Extraindo *Features* . . . 659
13.12.5 Definindo a Função de Perda . . . 660
13.12.6 Criação e inicialização da imagem composta . . . 662
13.12.7 Treinamento . . . 662
13.12.8 Resumo . . . 665
13.12.9 Exercícios . . . 665
13.13 Classificação de Imagens (CIFAR-10) no Kaggle . . . 665
13.13.1 Obtendo e Organizando o *Dataset* . . . 666
13.13.2 Aumento de Imagem . . . 669
13.13.3 Lendo o *Dataset* . . . 670
13.13.4 Definindo o Modelo . . . 670
13.13.5 Definindo as Funções de Treinamento . . . 671
13.13.6 Treinamento e Validação do Modelo . . . 671
13.13.7 Classificando o Conjunto de Testes e Enviando Resultados no Kaggle . . . 672
13.13.8 Resumo . . . 673
13.13.9 Exercícios . . . 673
13.14 Identificação de Raça de Cachorro (*ImageNet Dogs*) no Kaggle . . . 673
13.14.1 Obtenção e organização do *Dataset* . . . 674
13.14.2 Aumento de Imagem . . . 676
13.14.3 Lendo o Dataset . . . 676
13.14.4 Definindo o Modelo . . . 677
13.14.5 Definindo as Funções de Treinamento . . . 678
13.14.6 Treinamento e Validação do Modelo . . . 679
13.14.7 Classificando o Conjunto de Testes e Enviando Resultados no Kaggle . . . 679
13.14.8 Resumo . . . 680
13.14.9 Exercícios . . . 680

14 Processamento de linguagem natural: Pré-treinamento 681
14.1 Incorporação de Palavras (word2vec) . . . 682
14.1.1 Por que não usar vetores one-hot? . . . 682
14.1.2 O Modelo Skip-Gram . . . 683
14.1.3 O modelo do conjunto contínuo de palavras (CBOW) . . . 685
14.1.4 Sumário . . . 686
14.1.5 Exercícios . . . 686
14.2 Treinamento Aproximado . . . 687
14.2.1 Amostragem Negativa . . . 687
14.2.2 Hierárquico Softmax . . . 689

14.2.3 Sumário 690
14.2.4 Exercícios 690
14.3 O conjunto de dados para incorporação de palavras com pré-treinamento 690
14.3.1 Leitura e pré-processamento do conjunto de dados 691
14.3.2 Subamostragem 691
14.3.3 Carregando o conjunto de dados 693
14.3.4 Juntando todas as coisas 696
14.3.5 Sumário 697
14.3.6 Exercícios 698
14.4 Pré-treinamento do word2vec 698
14.4.1 O Modelo Skip-Gram 698
14.4.2 Treinamento 700
14.4.3 Aplicando o modelo de incorporação de palavras 702
14.4.4 Sumário 703
14.4.5 Exercícios 703
14.5 Incorporação de palavras com vetores globais (GloVe) 703
14.5.1 O modelo GloVe 704
14.5.2 Compreendendo o GloVe a partir das razões de probabilidade condicionais 705
14.5.3 Sumário 706
14.5.4 Exercícios 706
14.6 Encontrando sinônimos e analogias 706
14.6.1 Usando vetores de palavras pré-treinados 707
14.6.2 Aplicação de vetores de palavras pré-treinados 708
14.6.3 Sumário 711
14.6.4 Exercícios 711
14.7 Representações de codificador bidirecional de transformadores (BERT) 711
14.7.1 De Independente do Contexto para Sensível ao Contexto 711
14.7.2 De Task-Specific para Task-Agnostic 712
14.7.3 BERT: Combinando o melhor dos dois mundos 712
14.7.4 Representação de entrada 713
14.7.5 Tarefas de pré-treinamento 716
14.7.6 Juntando todas as coisas 718
14.7.7 Sumário 719
14.7.8 Exercícios 720
14.8 O conjunto de dados para pré-treinamento de BERT 720
14.8.1 Definindo funções auxiliares para tarefas de pré-treinamento 721
14.8.2 Transformando texto em conjunto de dados de pré-treinamento 723
14.8.3 Sumário 725
14.8.4 Exercícios 726
14.9 Pré-treinando BERT 726
14.9.1 Pré-treinamento de BERT 726
14.9.2 Representando Texto com BERT 729
14.9.3 Sumário 730
14.9.4 Exercícios 730

15 Processamento de Linguagem Natural: Aplicações **731**
15.1 Análise de Sentimentos e o *Dataset* 732
15.1.1 O Dataset de Análise de Sentimento 732
15.1.2 Juntando Tudo 735
15.1.3 Resumo 735
15.1.4 Exercícios 735

15.2 Análise de Sentimento: Usando Redes Neurais Recorrentes 735
15.2.1 Usando um Modelo de Rede Neural Recorrente 736
15.2.2 Resumo 739
15.2.3 Exercícios 739
15.3 Análise de Sentimento: Usando Redes Neurais Convolucionais 739
15.3.1 Camada Convolucional Unidimensional 740
15.3.2 Camada de Pooling Máximo ao Longo do Tempo 742
15.3.3 O Modelo TextCNN 743
15.3.4 Resumo 746
15.3.5 Exercícios 746
15.4 Inferência de Linguagem Natural e o *Dataset* 746
15.4.1 Inferência de Linguagem Natural 747
15.4.2 Conjunto de dados Stanford Natural Language Inference (SNLI) 747
15.4.3 Resumo 751
15.4.4 Exercícios 751
15.5 Inferência de Linguagem Natural: Usando a Atenção 751
15.5.1 O Modelo 752
15.5.2 Treinamento e Avaliação do Modelo 756
15.5.3 Resumo 757
15.5.4 Exercícios 758
15.6 Ajuste Fino de BERT para Aplicações de Nível de Sequência e de Token 758
15.6.1 Classificação de Texto Único 759
15.6.2 Classificação ou Regressão de Pares de Texto 759
15.6.3 Marcação de Texto 760
15.6.4 Resposta a Perguntas 761
15.6.5 Resumo 762
15.6.6 Exercícios 763
15.7 Inferência de Linguagem Natural: Ajuste Fino do BERT 763
15.7.1 Carregando o BERT Pré-treinado 764
15.7.2 O Conjunto de Dados para Ajuste Fino do BERT 765
15.7.3 Ajuste Fino do BERT 766
15.7.4 Resumo 768
15.7.5 Exercícios 768

16 Sistemas de Recomendação **769**
16.1 Visão geral dos sistemas de recomendação 769
16.1.1 Filtragem colaborativa 770
16.1.2 Feedback explícito e feedback implícito 771
16.1.3 Tarefas de recomendação 771
16.1.4 Sumário 772
16.1.5 Exercícios 772

17 Redes Adversariais Generativas **773**
17.1 Redes Adversariais Generativas 773
17.1.1 Gerando Alguns Dados “Reais” 775
17.1.2 Gerador 776
17.1.3 Discriminador 776
17.1.4 Treinamento 776
17.1.5 Resumo 778
17.1.6 Exercícios 778
17.2 Redes Adversariais Gerativas Convolucionais Profundas 779

17.2.1 O *Dataset* de Pokémon . . . 779
17.2.2 O Gerador . . . 780
17.2.3 Discriminador . . . 782
17.2.4 Treinamento . . . 784
17.2.5 Resumo . . . 785
17.2.6 Exercícios . . . 786

18 Apêndice: Matemática para *Deep Learning* **787**
18.1 Operações de Geometria e Álgebra Linear . . . 788
18.1.1 Geometria Vetorial . . . 788
18.1.2 Produto Escalar e Ângulos . . . 790
18.1.3 Hiperplanos . . . 792
18.1.4 Geometria de Transformações Lineares . . . 795
18.1.5 Dependência Linear . . . 797
18.1.6 Classificação . . . 798
18.1.7 Invertibilidade . . . 798
18.1.8 Determinante . . . 800
18.1.9 Tensores e Operações de Álgebra Linear Comum . . . 801
18.1.10 Resumo . . . 803
18.1.11 Exercícios . . . 804
18.2 Autovalores e Autovetores . . . 804
18.2.1 Encontrando Autovalores . . . 805
18.2.2 Matrizes de Decomposição . . . 806
18.2.3 Operações em Autovalores e Autovetores . . . 806
18.2.4 Composições Originais de Matrizes Simétricas . . . 807
18.2.5 Teorema do Círculo de Gershgorin . . . 807
18.2.6 Uma Aplicação Útil: o Crescimento de Mapas Iterados . . . 808
18.2.7 Conclusões . . . 813
18.2.8 Resumo . . . 813
18.2.9 Exercícios . . . 813
18.3 Cálculo de Variável Única . . . 814
18.3.1 Cálculo diferencial . . . 814
18.3.2 Regras de Cálculo . . . 818
18.3.3 Resumo . . . 825
18.3.4 Exercícios . . . 825
18.4 Cálculo Multivariável . . . 825
18.4.1 Diferenciação de Dimensões Superiores . . . 826
18.4.2 Geometria de Gradientes e Gradiente Descendente . . . 827
18.4.3 Uma Nota Sobre Otimização Matemática . . . 828
18.4.4 Regra da Cadeia Multivariada . . . 829
18.4.5 O Algoritmo de Retropropagação . . . 831
18.4.6 Hessians . . . 834
18.4.7 Um Pouco de Cálculo Matricial . . . 836
18.4.8 Resumo . . . 840
18.4.9 Exercícios . . . 841
18.5 Cálculo Integral . . . 841
18.5.1 Interpretação Geométrica . . . 841
18.5.2 O Teorema Fundamental do Cálculo . . . 844
18.5.3 Mudança de Variável . . . 845
18.5.4 Um Comentário Sobre as Convenções de Sinais . . . 846
18.5.5 Integrais Múltiplas . . . 847

18.5.6 Mudança de Variáveis em Integrais Múltiplas 849
18.5.7 Resumo 850
18.5.8 Exercícios 851
18.6 Variáveis Aleatórias 851
18.6.1 Variáveis Aleatórias Contínuas 851
18.6.2 Resumo 868
18.6.3 Exercícios 868
18.7 Máxima verossimilhança 869
18.7.1 O Princípio da Máxima Verossimilhança 869
18.7.2 Otimização Numérica e o Log-Probabilidade Negativa 871
18.7.3 Máxima probabilidade para variáveis contínuas 873
18.7.4 Resumo 874
18.7.5 Exercícios 874
18.8 Distribuições 874
18.8.1 Bernoulli 875
18.8.2 Uniforme e Discreta 876
18.8.3 Uniforme e Contínua 878
18.8.4 Binomial 880
18.8.5 Poisson 882
18.8.6 Gaussiana 885
18.8.7 Família Exponencial 888
18.8.8 Resumo 889
18.8.9 Exercícios 890
18.9 Naive Bayes 890
18.9.1 Reconhecimento Ótico de Caracteres 890
18.9.2 O Modelo Probabilístico para Classificação 892
18.9.3 O Classificador Naive Bayes 892
18.9.4 Trainamento 893
18.9.5 Resumo 896
18.9.6 Exercícios 897
18.10 Estatísticas 897
18.10.1 Avaliando e comparando estimadores 898
18.10.2 Conducting Hypothesis Tests 902
18.10.3 Construindo Intervalos de Confiança 905
18.10.4 Resumo 908
18.10.5 Exercícios 908
18.11 Teoria da Informação 909
18.11.1 Informação 909
18.11.2 Entropia 911
18.11.3 Informcação Mútua 913
18.11.4 Divergência de Kullback–Leibler 917
18.11.5 Entropia Cruzada 919
18.11.6 Resumo 922
18.11.7 Exercícios 922

19 Apêndice: Ferramentas para Deep Learning 925
19.1 Usando Jupyter 925
19.1.1 Editando e executando o código localmente 925
19.1.2 Opções avançadas 929
19.1.3 Sumário 930
19.1.4 Exercícios 930

19.2 Usando Amazon SageMaker . . . 931
19.2.1 Registro e login . . . 931
19.2.2 Criação de uma instância do SageMaker . . . 931
19.2.3 Executando e parando uma instância . . . 932
19.2.4 Atualizando Notebooks . . . 933
19.2.5 Sumário . . . 934
19.2.6 Exercícios . . . 934
19.3 Usando instâncias AWS EC2 . . . 934
19.3.1 Criação e execução de uma instância EC2 . . . 935
19.3.2 Instalando CUDA . . . 940
19.3.3 Instalação do MXNet e download dos notebooks D2L . . . 941
19.3.4 Executando Jupyter . . . 942
19.3.5 Fechando instâncias não utilizadas . . . 942
19.3.6 Sumário . . . 943
19.3.7 Exercícios . . . 943
19.4 Usando Google Colab . . . 943
19.4.1 Sumário . . . 944
19.4.2 Exercícios . . . 944
19.5 Seleção de servidores e GPUs . . . 944
19.5.1 Selecionando Servidores . . . 945
19.5.2 Sumário . . . 948
19.6 Contribuindo para este livro . . . 949
19.6.1 Pequenas alterações de texto . . . 949
19.6.2 Propor uma mudança importante . . . 950
19.6.3 Adicionando uma nova seção ou uma nova implementação de estrutura . 950
19.6.4 Enviando uma Mudança Principal . . . 951
19.6.5 Sumário . . . 954
19.6.6 Exercícios . . . 954
19.7 Documento da API `d2l` . . . 954

Bibliography **979**

Python Module Index **989**

Index **991**

Prefácio

Apenas alguns anos atrás, não havia legiões de cientistas de *deep learning* desenvolvendo produtos e serviços inteligentes em grandes empresas e *startups*. Quando o mais jovem entre nós (os autores) entrou no campo, o *machine learning* não comandava as manchetes dos jornais diários. Nossos pais não faziam ideia do que era *machine learning*, muito menos por que podemos preferir isso a uma carreira em medicina ou direito. Machine learning era uma disciplina acadêmica voltada para o futuro com um conjunto restrito de aplicações do mundo real. E essas aplicações, por exemplo, reconhecimento de voz e visão computacional, exigiam tanto conhecimento de domínio que muitas vezes eram considerados como áreas inteiramente separadas para as quais o aprendizado de máquina era um pequeno componente. Redes neurais, então, os antecedentes dos modelos de aprendizagem profunda nos quais nos concentramos neste livro, eram considerados ferramentas obsoletas.

Apenas nos últimos cinco anos, o *deep learning* pegou o mundo de surpresa, impulsionando o rápido progresso em campos tão diversos como a visão computacional, processamento de linguagem natural, reconhecimento automático de fala, aprendizagem por reforço e modelagem estatística. Com esses avanços em mãos, agora podemos construir carros que se dirigem sozinhos com mais autonomia do que nunca (e menos autonomia do que algumas empresas podem fazer você acreditar), sistemas de resposta inteligente que redigem automaticamente os e-mails mais comuns, ajudando as pessoas a se livrarem de caixas de entrada opressivamente grandes, e agentes de *software* que dominam os melhores humanos do mundo em jogos de tabuleiro como Go, um feito que se pensava estar a décadas de distância. Essas ferramentas já exercem impactos cada vez maiores na indústria e na sociedade, mudando a forma como os filmes são feitos, as doenças são diagnosticadas, e desempenhando um papel crescente nas ciências básicas — da astrofísica à biologia.

Sobre Este Livro

Esse livro representa nossa tentativa de tornar o *deep learning* acessível, lhes ensinando os *conceitos*, o *contexto* e o *código*.

Um Meio (?) Combinando Código, Matemática e HTML

Para que qualquer tecnologia de computação alcance seu impacto total, deve ser bem compreendido, bem documentado e apoiado por ferramentas maduras e bem conservadas. As ideias-chave devem ser claramente destiladas, minimizando o tempo de integração necessário para atualizar os novos praticantes. Bibliotecas maduras devem automatizar tarefas comuns, e o código exemplar deve tornar mais fácil para os profissionais para modificar, aplicar e estender aplicativos comuns para atender às suas necessidades. Considere os aplicativos da Web dinâmicos como exemplo. Apesar de um grande número de empresas, como a Amazon, desenvolver aplicativos da web baseados em banco de dados de sucesso na década de 1990, o potencial desta tecnologia para auxiliar empreendedores criativos foi percebido em um grau muito maior nos últimos dez anos, devido em parte ao desenvolvimento de *frameworks* poderosos e bem documentados.

Testar o potencial do *deep learning* apresenta desafios únicos porque qualquer aplicativo reúne várias disciplinas. Aplicar o *deep learning* requer compreensão simultânea (i) as motivações para definir um problema de uma maneira particular; (ii) a matemática de uma dada abordagem de modelagem; (iii) os algoritmos de otimização para ajustar os modelos aos dados; e (iv) a engenharia necessária para treinar modelos de forma eficiente, navegando nas armadilhas da computação numérica e obter o máximo do *hardware* disponível. Ensinar as habilidades de pensamento crítico necessárias para formular problemas, a matemática para resolvê-los e as ferramentas de *software* para implementar tais soluções em um só lugar apresentam desafios formidáveis. Nosso objetivo neste livro é apresentar um recurso unificado para trazer os praticantes em potencial.

Na época em que começamos o projeto deste livro, não havia recursos que simultaneamente (i) estavam em dia; (ii) cobriam toda a largura de *machine learning* moderno com profundidade técnica substancial; e (iii) intercalassem exposição da qualidade que se espera de um livro envolvente com o código limpo executável que se espera encontrar em tutoriais práticos. Encontramos muitos exemplos de código para como usar um determinado *framework* de aprendizado profundo (por exemplo, como fazer computação numérica básica com matrizes no *TensorFlow*) ou para a implementação de técnicas particulares (por exemplo, *snippets* de código para LeNet, AlexNet, ResNets, etc) espalhados por vários posts de blog e repositórios GitHub. No entanto, esses exemplos normalmente se concentram em *como* implementar uma determinada abordagem, mas deixou de fora a discussão de *por que* certas decisões algorítmicas são feitas. Embora alguns recursos interativos tenham surgido esporadicamente para abordar um tópico específico, por exemplo, as postagens de blog envolventes publicado no site [Distill] (http://distill.pub), ou blogs pessoais, eles cobriram apenas tópicos selecionados no aprendizado profundo, e muitas vezes não tinham código associado. Por outro lado, embora vários livros tenham surgido, mais notavelmente: cite: `Goodfellow.Bengio.Courville.2016`, que oferece uma pesquisa abrangente dos conceitos por trás do aprendizado profundo, esses recursos não combinam com as descrições às realizações dos conceitos no código, às vezes deixando os leitores sem noção de como implementá-los. Além disso, muitos recursos estão escondidos atrás dos *paywalls* de fornecedores de cursos comerciais.

We set out to create a resource that could (i) be freely available for everyone; (ii) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist; (iii) include runnable code, showing readers *how* to solve problems in practice; (iv) allow for rapid updates, both by us and also by the community at large; and (v) be complemented by a forum[2] for interactive discussion of technical details and to answer questions.

Propusemo-nos a criar um recurso que pudesse (i) estar disponível gratuitamente para todos; (ii) oferecer profundidade técnica suficiente para fornecer um ponto de partida no caminho para

[2] http://discuss.d2l.ai

realmente se tornar um cientista de *machine learning* aplicado; (iii) incluir código executável, mostrando aos leitores *como* resolver problemas na prática; (iv) permitir atualizações rápidas, tanto por nós e também pela comunidade em geral; e (v) ser complementado por um [fórum] (http://discuss.d2l.ai) para uma discussão interativa de detalhes técnicos e para responder a perguntas.

Esses objetivos costumavam estar em conflito. Equações, teoremas e citações são melhor gerenciados e apresentados em LaTeX. O código é melhor descrito em Python. E as páginas da web são nativas em HTML e JavaScript. Além disso, queremos que o conteúdo seja acessível tanto como código executável, como livro físico, como um PDF para *download* e na Internet como um site. No momento não existem ferramentas e nenhum *workflow* perfeitamente adequado a essas demandas, então tivemos que montar o nosso próprio. Descrevemos nossa abordagem em detalhes em: numref: `sec_how_to_contribute`. Decidimos usar o GitHub para compartilhar a fonte e permitir edições, *Notebooks* Jupyter para misturar código, equações e texto, Sphinx como um mecanismo de renderização para gerar várias saídas, e *Discourse* para o fórum. Embora nosso sistema ainda não seja perfeito, essas escolhas fornecem um bom compromisso entre as preocupações concorrentes. Acreditamos que este seja o primeiro livro publicado usando um *workflow* integrado.

Aprendendo Fazendo

Muitos livros ensinam uma série de tópicos, cada um com detalhes exaustivos. Por exemplo, o excelente livro de Chris Bishop: cite: `Bishop.2006`, ensina cada tópico tão completamente, que chegar ao capítulo na regressão linear requer uma quantidade não trivial de trabalho. Embora os especialistas amem este livro precisamente por sua eficácia, para iniciantes, essa propriedade limita sua utilidade como um texto introdutório.

Neste livro, ensinaremos a maioria dos conceitos *just in time*. Em outras palavras, você aprenderá conceitos no exato momento que eles são necessários para realizar algum fim prático. Enquanto levamos algum tempo no início para ensinar preliminares fundamentais, como álgebra linear e probabilidade, queremos que você experimente a satisfação de treinar seu primeiro modelo antes de se preocupar com distribuições de probabilidade mais esotéricas.

Além de alguns cadernos preliminares que fornecem um curso intensivo no *background* matemático básico, cada capítulo subsequente apresenta um número razoável de novos conceitos e fornece exemplos de trabalho auto-contidos únicos — usando *datasets* reais. Isso representa um desafio organizacional. Alguns modelos podem ser agrupados logicamente em um único *notebook*. E algumas idéias podem ser melhor ensinadas executando vários modelos em sucessão. Por outro lado, há uma grande vantagem em aderir a uma política de *um exemplo funcional, um notebook*: Isso torna o mais fácil possível para você comece seus próprios projetos de pesquisa aproveitando nosso código. Basta copiar um *notebook* e começar a modificá-lo.

Vamos intercalar o código executável com o *background* de material, conforme necessário. Em geral, muitas vezes erramos por fazer ferramentas disponíveis antes de explicá-los totalmente (e vamos acompanhar por explicando o *background* mais tarde). Por exemplo, podemos usar *gradiente descendente estocástico* antes de explicar completamente porque é útil ou porque funciona. Isso ajuda a dar aos profissionais a munição necessária para resolver problemas rapidamente, às custas de exigir do leitor que nos confie algumas decisões curatoriais.

Este livro vai ensinar conceitos de *deep learning* do zero. Às vezes, queremos nos aprofundar em detalhes sobre os modelos que normalmente ficaria oculto do usuário pelas abstrações avançadas dos *frameworks* de *deep learning*. Isso surge especialmente nos tutoriais básicos, onde queremos que você entenda tudo que acontece em uma determinada camada ou otimizador. Nesses casos, apresentaremos frequentemente duas versões do exemplo: onde implementamos tudo do

zero, contando apenas com a interface NumPy e diferenciação automática, e outro exemplo mais prático, onde escrevemos código sucinto usando APIs de alto nível de *frameworks* de *deep learning*. Depois de ensinar a você como alguns componentes funcionam, podemos apenas usar as APIs de alto nível em tutoriais subsequentes.

Conteúdo e Estrutura

O livro pode ser dividido em três partes, que são apresentados por cores diferentes em: numref: `fig_book_org`:

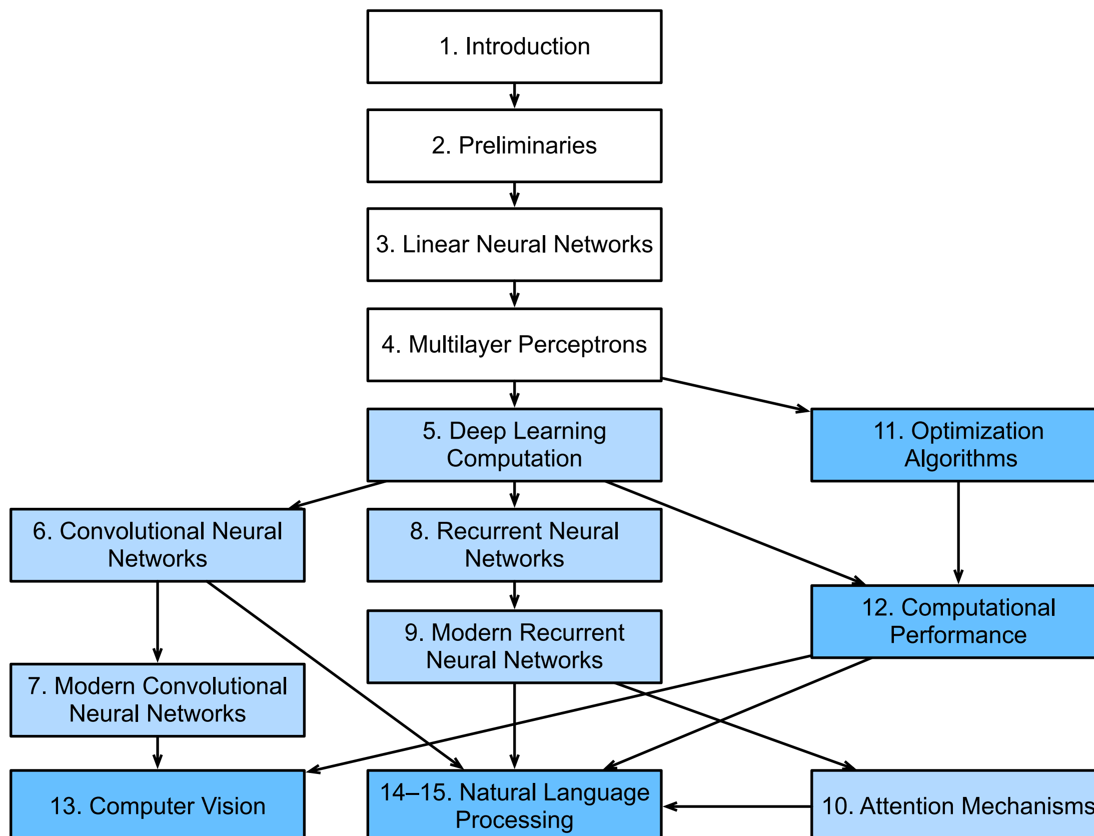


Fig. 1: Book structure

- **A primeira parte cobre os princípios básicos e preliminares.** numref: `chap_introduction` oferece uma introdução ao *deep learning*. Então, em: numref: `chap_preliminaries`, nós o informamos rapidamente sobre os pré-requisitos exigidos para *deep learning* prático, como armazenar e manipular dados, e como aplicar várias operações numéricas com base em conceitos básicos da álgebra linear, cálculo e probabilidade. numref: `chap_linear` e: numref:`chap_perceptrons` cobrem os conceitos e técnicas mais básicos de aprendizagem profunda, como regressão linear, *multilayer perceptrons* e regularização.
- **Os próximos cinco capítulos enfocam as técnicas modernas de *deep learning*.** numref: `chap_computation` descreve os vários componentes-chave dos cálculos do *deep learning* e estabelece as bases para que possamos posteriormente implementar modelos mais complexos. A seguir, em: numref: `chap_cnn` e: numref:`chap_modern_cnn`, apresentamos redes neurais convolucionais (CNNs, do inglês *convolutional neural*

networks), ferramentas poderosas que formam a espinha dorsal da maioria dos sistemas modernos de visão computacional. Posteriormente, em: numref: `chap_rnn` e: numref:`chap_modern_rnn`, apresentamos redes neurais recorrentes (RNNs, do inglês *recurrent neural networks*), modelos que exploram estrutura temporal ou sequencial em dados, e são comumente usados para processamento de linguagem natural e previsão de séries temporais. Em: numref: `chap_attention`, apresentamos uma nova classe de modelos que empregam uma técnica chamada mecanismos de atenção, que recentemente começaram a deslocar RNNs no processamento de linguagem natural. Estas seções irão ajudá-lo a aprender sobre as ferramentas básicas por trás da maioria das aplicações modernas de *deep learning*.

- A parte três discute escalabilidade, eficiência e aplicações. Primeiro, em: numref: `chap_optimization`, discutimos vários algoritmos de otimização comuns usado para treinar modelos de *deep learning*. O próximo capítulo,: numref: `chap_performance` examina vários fatores-chave que influenciam o desempenho computacional de seu código de *deep learning*. Em: numref: `chap_cv`, nós ilustramos as principais aplicações de *deep learning* em visão computacional. Em: numref: `chap_nlp_pretrain` e: numref:`chap_nlp_app`, mostramos como pré-treinar modelos de representação de linguagem e aplicar para tarefas de processamento de linguagem natural.

Códigos

A maioria das seções deste livro apresenta código executável devido a acreditarmos na importância de uma experiência de aprendizagem interativa em *deep learning*. No momento, certas intuições só podem ser desenvolvidas por tentativa e erro, ajustando o código em pequenas formas e observando os resultados. Idealmente, uma elegante teoria matemática pode nos dizer precisamente como ajustar nosso código para alcançar o resultado desejado. Infelizmente, no momento, essas teorias elegantes nos escapam. Apesar de nossas melhores tentativas, explicações formais para várias técnicas ainda faltam, tanto porque a matemática para caracterizar esses modelos pode ser tão difícil e também porque uma investigação séria sobre esses tópicos só recentemente entrou em foco. Temos esperança de que, à medida que a teoria do *deep learning* avança, futuras edições deste livro serão capazes de fornecer *insights* em lugares em que a presente edição não pode.

Às vezes, para evitar repetição desnecessária, encapsulamos as funções, classes, etc. importadas e mencionadas com frequência neste livro no *package* `d2l`. Para qualquer bloco, como uma função, uma classe ou vários *imports* ser salvo no pacote, vamos marcá-lo com `# @ save`. Oferecemos uma visão geral detalhada dessas funções e classes em: numref: `sec_d2l`. O *package* `d2l` é leve e requer apenas os seguintes *packages* e módulos como dependências:

```
#@save
import collections
import hashlib
import math
import os
import random
import re
import shutil
import sys
import tarfile
import time
import zipfile
```

(continues on next page)

(continued from previous page)

```
from collections import defaultdict
import pandas as pd
import requests
from IPython import display
from matplotlib import pyplot as plt

d2l = sys.modules[__name__]
```

: begin_tab: `mxnet` A maior parte do código neste livro é baseada no Apache MXNet. MXNet é um *framework* de código aberto (*oper-source*) para *deep learning* e a escolha preferida de AWS (*Amazon Web Services*), bem como muitas faculdades e empresas. Todo o código neste livro passou nos testes da versão mais recente do MXNet. No entanto, devido ao rápido desenvolvimento do *deep learning*, alguns códigos *na edição impressa* podem não funcionar corretamente em versões futuras do MXNet. No entanto, planejamos manter a versão *online* atualizada. Caso você encontre algum desses problemas, consulte: ref: `chap_installation` para atualizar seu código e ambiente de execução.

Aqui está como importamos módulos do MXNet. end_tab: begin_tab: `pytorch` A maior parte do código neste livro é baseada no PyTorch. PyTorch é uma estrutura de código aberto para *deep learning*, que é extremamente popular na comunidade de pesquisa. Todo o código neste livro passou nos testes do mais novo PyTorch. No entanto, devido ao rápido desenvolvimento do *deep learning*, alguns códigos *na edição impressa* podem não funcionar corretamente em versões futuras do PyTorch. No entanto, planejamos manter a versão *online* atualizada. Caso você encontre algum desses problemas, consulte: ref: `chap_installation` para atualizar seu código e ambiente de execução.

Aqui está como importamos módulos do PyTorch. end_tab: begin_tab: `tensorflow` A maior parte do código deste livro é baseada no TensorFlow. TensorFlow é uma estrutura de código aberto para *deep learning*, que é extremamente popular na comunidade de pesquisa e na indústria. Todo o código deste livro passou nos testes do TensorFlow mais recente. No entanto, devido ao rápido desenvolvimento do *deep learning*, alguns códigos *na edição impressa* podem não funcionar corretamente em versões futuras do TensorFlow. No entanto, planejamos manter a versão online atualizada. Caso você encontre algum desses problemas, consulte: ref: `chap_installation` para atualizar seu código e ambiente de execução.

Aqui está como importamos módulos do TensorFlow. end_tab:

```
#@save
import numpy as np
import torch
import torchvision
from PIL import Image
from torch import nn
from torch.nn import functional as F
from torch.utils import data
from torchvision import transforms
```

Público-alvo

Este livro é para estudantes (graduação ou pós-graduação), engenheiros e pesquisadores que buscam uma compreensão sólida das técnicas práticas de *deep learning*. Porque explicamos cada conceito do zero, nenhuma experiência anterior em *deep learning* ou *machine learning* é necessária. Explicando totalmente os métodos de *deep learning* requer matemática e programação, mas vamos apenas supor que você veio com algumas noções básicas, incluindo (o básico de) álgebra linear, cálculo, probabilidade, e programação Python. Além disso, no Apêndice, fornecemos uma atualização na maior parte da matemática abordada neste livro. Na maioria das vezes, priorizaremos intuição e ideias sobre o rigor matemático. Existem muitos livros fantásticos que podem levar o leitor interessado ainda mais longe. Por exemplo, *Linear Analysis* de Bela Bollobas: cite: `Bollobas.1999` cobre álgebra linear e análise funcional em grande profundidade. *All of Statistics*: cite: `Wasserman.2013` é um excelente guia para estatísticas. E se você nunca usou Python antes, você pode querer dar uma olhada neste [tutorial de Python] (http://learnpython.org/).

Fórum

Associado a este livro, lançamos um fórum de discussão, localizado em [discuss.d2l.ai] (https://discuss.d2l.ai/). Quando você tiver dúvidas sobre qualquer seção do livro, você pode encontrar o link da página de discussão associada no final de cada capítulo.

Agradecimentos

Estamos em dívida com as centenas de contribuintes de ambos os esboços ingleses e chineses e brasileiros. Eles ajudaram a melhorar o conteúdo e ofereceram feedback valioso. Especificamente, agradecemos a todos os contribuintes deste rascunho em inglês para torná-lo melhor para todos. Seus IDs ou nomes do GitHub são (sem nenhuma ordem específica): alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, criptonauta, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, Igor Dzreyev, Ha Nguyen, pmuens, Andrei Lukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, Ruslan Baratov, Rafael Schlatter, liusy182, Giannis Pappas, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phuc Le, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen, noelo, hansent, Giel Dops, dvincent1337, WhiteD3vil, Peter Kulits, codypenta, joseppinilla, ahmaurya, karolszk, heytitle, Peter Goetz, rigtorp, Tiep Vu, sfilip, mlxd, Kale-ab Tessera, Sanjar Adilov, MatteoFerrara, hsneto, Katarzyna Biesialska, Gregory Bruss, Duy – Thanh Doan, paulaurel, graytowne, Duc Pham, sl7423, Jaedong Hwang, Yida Wang, cys4, clhm, Jean Kaddour, austinmw, trebeljahr, tbaums, Cuong V. Nguyen, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the-great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, biagiom, abhinavsp0730, jonathanhrandall, ysraell, Nodar Okroshiashvili, UgurKap, Jiyang Kang, StevenJokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, Hoa Nguyen, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, Josiah Yoder, mathresearch, mzz2017, jroberayalas, iluu, ghejc,

BSharmi, vkramdev, simonwardjones, LakshKD, TalNeoran, djliden, Nikhil95, Oren Barkan, guoweis, haozhu233, pratikhack, 315930399, tayfununal, steinsag, charleybeller, Andrew Lumsdaine, Jiekui Zhang, Deepak Pathak, Florian Donhauser, Tim Gates, Adriaan Tijsseling, Ron Medina, Gaurav Saha, Murat Semerci, Lei Mao, Levi McClenny, Joshua Broyde, jake221, jonbally, zyhazwraith, Brian Pulfer, Nick Tomasino.

Agradecemos à Amazon Web Services, especialmente a Swami Sivasubramanian, Raju Gulabani, Charlie Bell e Andrew Jassy por seu generoso apoio ao escrever este livro. Sem o tempo disponível, recursos, discussões com colegas e incentivo contínuo, este livro não teria acontecido.

Resumo

- O *deep learning* revolucionou o reconhecimento de padrões, introduzindo tecnologia que agora capacita uma ampla gama de tecnologias, incluindo visão computacional, processamento de linguagem natural e reconhecimento automático de fala.
- Para aplicar com sucesso o *deep learning*, você deve entender como lançar um problema, a matemática da modelagem, os algoritmos para ajustar seus modelos aos dados e as técnicas de engenharia para implementar tudo isso.
- Este livro apresenta um recurso abrangente, incluindo prosa, figuras, matemática e código, tudo em um só lugar.
- Para responder a perguntas relacionadas a este livro, visite nosso fórum em https://discuss.d2l.ai/.
- Todos os *notebooks* estão disponíveis para *download* no GitHub.

Exercícios

1. Registre uma conta no fórum de discussão deste livro [discuss.d2l.ai] (https://discuss.d2l.ai/).
2. Instale Python em seu computador.
3. Siga os links na parte inferior da seção para o fórum, onde você poderá buscar ajuda e discutir o livro e encontrar respostas para suas perguntas envolvendo os autores e a comunidade em geral.

Discussions[3]

[3] https://discuss.d2l.ai/t/20

Instalação

Para prepara-lo a ter uma experiência prática de aprendizado, precisamos configurar o ambiente para executar *Python*, *Jupyter notebooks*, as bibliotecas relevantes, e o código necessário para executar o livro em si.

Instalando Miniconda

A maneira mais simples de começar será instalar Miniconda[4]. A versão *Python* 3.x é necessária. Você pode pular as etapas a seguir se o conda já tiver sido instalado. Baixe o arquivo Miniconda sh correspondente do site e então execute a instalação a partir da linha de comando usando `sh <FILENAME> -b`. Para usuários do macOS:

```
# O nome do arquivo pode estar diferente
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

Para os usuários de Linux:

```
# O nome do arquivo pode estar diferente
sh Miniconda3-latest-Linux-x86_64.sh -b
```

A seguir, inicialize o *shell* para que possamos executar `conda` diretamente.

```
~/miniconda3/bin/conda init
```

Agora feche e reabra seu *shell* atual. Você deve ser capaz de criar um novo ambiente da seguinte forma:

```
conda create --name d2l python=3.8 -y
```

[4] https://conda.io/en/latest/miniconda.html

Baixando os *Notebooks D2L*

Em seguida, precisamos baixar o código deste livro. Você pode clicar no botão "All Notebooks" na parte superior de qualquer página HTML para baixar e descompactar o código. Alternativamente, se você tiver `unzip` (caso contrário, execute`sudo apt install unzip`) disponível:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Agora precisamos ativar o ambiente `d2l`.

```
conda activate d2l
```

Instalando o *Framework* e o pacote `d2l`

Antes de instalar o *Framework* de *Deep Learning*, primeiro verifique se você tem ou não GPUs adequadas em sua máquina (as GPUs que alimentam a tela em um laptop padrão não contam para nossos propósitos). Se você estiver instalando em um servidor GPU, proceda para: ref: `subsec_gpu` para instruções para instalar uma versão compatível com GPU.

Caso contrário, você pode instalar a versão da CPU da seguinte maneira. Isso será mais do que potência suficiente para você pelos primeiros capítulos, mas você precisará acessar GPUs para executar modelos maiores.

```
pip install torch torchvision -f https://download.pytorch.org/whl/torch_stable.html
```

Nós também instalamos o pacote `d2l` que encapsula funções e classes frequentemente usadas neste livro.

```
# -U: Atualiza todos os pacotes para as versões mais atuais disponíveis
pip install -U d2l
```

Após realizadas as instalações podemos abrir os *notebooks Jupyter* através do seguinte comando:

```
jupyter notebook
```

Nesse ponto, você pode abrir http://localhost:8888 (geralmente abre automaticamente) no navegador da web. Em seguida, podemos executar o código para cada seção do livro. Sempre execute `conda activate d2l` para ativar o ambiente de execução antes de executar o código do livro ou atualizar o *framework* de *Deep Learning* ou o pacote `d2l`. Para sair do ambiente, execute `conda deactivate`.

Compatibilidade com GPU

Por padrão, o Framework de Deep Learning é instalada com suporte para GPU. Se o seu computador tem GPUs NVIDIA e instalou CUDA[5], então está tudo pronto.

Exercícios

1. Baixe o código do livro e instale o ambiente de execução.

Discussão[6]

[5] https://developer.nvidia.com/cuda-downloads
[6] https://discuss.d2l.ai/t/24

Notação

A notação usada ao longo deste livro é resumida a seguir.

Numbers

- x: um escalar
- $\mathbf{x}$: um vetor
- $\mathbf{X}$: uma matriz
- X: um tensor
- $\mathbf{I}$: Uma matriz identidade
- x_i,$[\mathbf{x}]_i$: O elemento i^th do vetor $\mathbf{x}$
- x_{ij},$x_{i,j}$,$[\mathbf{X}]_{ij}$,$[\mathbf{X}]_{i,j}$: O elemento da matriz $\mathbf{X}$ na linha i e coluna j

Teoria de conjuntos

- $\mathcal{X}$: um conjunto
- $\mathbb{Z}$: O conjunto dos inteiros
- $\mathbb{Z}^+$: O conjunto dos inteiros positivos
- $\mathbb{R}$: O conjunto dos números reais
- $\mathbb{R}^n$: O conjunto dos vetores n-dimensionais de números reais
- $\mathbb{R}^{a\times b}$: O conjunto de matrizes de números reais com a linhas e b colunas
- $|\mathcal{X}|$: Cardinalidade (número de elementos) do conjunto $\mathcal{X}$
- $\mathcal{A}\cup\mathcal{B}$: União dos conjuntos $\mathcal{A}$ e $\mathcal{B}$
- $\mathcal{A}\cap\mathcal{B}$: Interseção dos conjuntos $\mathcal{A}$ e $\mathcal{B}$
- $\mathcal{A}\setminus\mathcal{B}$: Subtração do conjunto $\mathcal{B}$ do conjunto $\mathcal{A}$

Funções e operadores

- $f(\cdot)$: uma função
- $\log(\cdot)$: O logaritmo natural
- $\exp(\cdot)$: A função exponencial
- $\mathbf{1}_{\mathcal{X}}$: a função do indicador
- $(\cdot)^\top$: Transposta de uma matriz ou vetor
- $\mathbf{X}^{-1}$: Inversa da matriz $\mathbf{X}$
- $\odot$: produto Hadamard (elemento a elemento)
- $[\cdot, \cdot]$: Concatenação
- $:raw-latex:*lvert* `:*raw-latex:*mathcal {X}` **:raw-latex:`\rvert `$: Cardinalidade do conjunto :math:`**mathcal {X}`
- $| **:raw-latex:`\cdot `\| _p $: :math:`L_p`** norma
- $| \cdot \|$: L_2 norma
- $:raw-latex:*langle* `:*raw-latex:*mathbf {x}`, **:raw-latex:`\mathbf {y}` :raw-latex:`\rangle `$: Produto escalar dos vetores :math:`**mathbf {x}` e $:raw-latex:*mathbf {y}* $
- $:raw-latex:**`**sum **`**$: adição de séries
- $:raw-latex:**`**prod **`**$: multiplicação de séries
- $:raw-latex:*stackrel {mathrm {def}} {=}* $: Definição

Cálculo

- $:raw-latex:*frac{dy}{dx}* $: Derivada de y em relação a x
- $:raw-latex:*frac{partial y}{partial x}* $: Derivada parcial de y em relação a x
- $:raw-latex:*nabla_*{**:raw-latex:`\mathbf {x}`**} y $: Gradiente de y em relação a $\mathbf{x}$
- $:raw-latex:*int_*a ^ b f (x) ; dx $: Integral definida de f de a a b em relação a x
- $:raw-latex:*int `f (x) ; dx $: Integral indefinida de :math:`f* em relação a x

Probabilidade e Teoria da Informação

- $P (**:raw-latex:`\cdot`**) $: distribuição de probabilidade
- $z **:raw-latex:`\sim `P $: Variável aleatória :math:`z`** que tem distribuição de probabilidade P
- $P (X **:raw-latex:`\mid `Y) $: probabilidade condicional de :math:`X\mid Y`**
- $p (x) $: função de densidade de probabilidade
- ${E} _ {x} [f (x)] $: Esperança de f em relação a x

- $X **:raw-latex:`\perp `Y $: Variáveis aleatórias :math:`X`** e Y são independentes
- $X **:raw-latex:`\perp `Y :raw-latex:`**mid $Z$ *$: Variáveis aleatórias :math:`* X e Y são condicionalmente independentes, dada a variável aleatória Z
- $:raw-latex:*mathrm {Var}* (X) $: Variância da variável aleatória X
- $:raw-latex:*sigma*_X $: Desvio padrão da variável aleatória X
- $:raw-latex:*mathrm {Cov}* (X, Y) $: Covariância das variáveis aleatórias X e Y
- $:raw-latex:*rho `(X, Y) $: Correlação de variáveis aleatórias :math:`* X e Y
- $H (X) $: Entropia da variável aleatória X
- $D _ {**:raw-latex:`\mathrm {KL}`**} (P | Q) $: KL-divergência das distribuições P e Q

Complexidade

- $\mathcal{O}$: notação *Big O*

Discussions[7]

[7] https://discuss.d2l.ai/t/25

1 | Introdução

Até recentemente, quase todos os programas de computador com os quais interagimos diariamente eram codificados por desenvolvedores de software desde os primeiros princípios. Digamos que quiséssemos escrever um aplicativo para gerenciar uma plataforma de *e-commerce*. Depois de se amontoar em um quadro branco por algumas horas para refletir sobre o problema, iríamos apresentar os traços gerais de uma solução de trabalho que provavelmente se pareceria com isto: (i) os usuários interagem com o aplicativo por meio de uma interface executando em um navegador da *web* ou aplicativo móvel; (ii) nosso aplicativo interage com um mecanismo de banco de dados de nível comercial para acompanhar o estado de cada usuário e manter registros de histórico de transações; e (iii) no cerne de nossa aplicação, a *lógica de negócios* (você pode dizer, os *cérebros*) de nosso aplicativo descreve em detalhes metódicos a ação apropriada que nosso programa deve levar em todas as circunstâncias concebíveis.

Para construir o cérebro de nosso aplicativo, teríamos que percorrer todos os casos esquivos possíveis que antecipamos encontrar, criando regras apropriadas. Cada vez que um cliente clica para adicionar um item ao carrinho de compras, adicionamos uma entrada à tabela de banco de dados do carrinho de compras, associando o ID desse usuário ao ID do produto solicitado. Embora poucos desenvolvedores acertem completamente na primeira vez (podem ser necessários alguns testes para resolver os problemas), na maior parte, poderíamos escrever esse programa a partir dos primeiros princípios e lançá-lo com confiança *antes* de ver um cliente real. Nossa capacidade de projetar sistemas automatizados a partir dos primeiros princípios que impulsionam o funcionamento de produtos, sistemas e, frequentemente em novas situações, é um feito cognitivo notável. E quando você é capaz de conceber soluções que funcionam 100% do tempo, você não deveria usar o *machine learning*.

Felizmente para a crescente comunidade de cientistas de *machine learning*, muitas tarefas que gostaríamos de automatizar não se curvam tão facilmente à habilidade humana. Imagine se amontoar em volta do quadro branco com as mentes mais inteligentes que você conhece, mas desta vez você está lidando com um dos seguintes problemas:

- Escreva um programa que preveja o clima de amanhã com base em informações geográficas, imagens de satélite e uma janela de rastreamento do tempo passado.
- Escreva um programa que aceite uma pergunta, expressa em texto de forma livre, e a responda corretamente.
- Escreva um programa que, dada uma imagem, possa identificar todas as pessoas que ela contém, desenhando contornos em torno de cada uma.
- Escreva um programa que apresente aos usuários produtos que eles provavelmente irão gostar, mas que provavelmente não encontrarão no curso natural da navegação.

Em cada um desses casos, mesmo programadores de elite são incapazes de codificar soluções do zero. As razões para isso podem variar. Às vezes, o programa que procuramos segue um padrão

que muda com o tempo, e precisamos que nossos programas se adaptem. Em outros casos, a relação (digamos, entre pixels, e categorias abstratas) podem ser muito complicadas, exigindo milhares ou milhões de cálculos que estão além da nossa compreensão consciente mesmo que nossos olhos administrem a tarefa sem esforço. *Machine learning* é o estudo de poderosas técnicas que podem aprender com a experiência. À medida que um algoritmo de *machine learning* acumula mais experiência, normalmente na forma de dados observacionais ou interações com um ambiente, seu desempenho melhora. Compare isso com nossa plataforma de comércio eletrônico determinística, que funciona de acordo com a mesma lógica de negócios, não importa quanta experiência acumule, até que os próprios desenvolvedores aprendam e decidam que é hora de atualizar o *software*. Neste livro, ensinaremos os fundamentos do *machine learning*, e foco em particular no *deep learning*, um poderoso conjunto de técnicas impulsionando inovações em áreas tão diversas como a visão computacional, processamento de linguagem natural, saúde e genômica.

1.1 Um exemplo motivador

Antes de começar a escrever, os autores deste livro, como grande parte da força de trabalho, tiveram que se tornar cafeinados. Entramos no carro e começamos a dirigir. Usando um iPhone, Alex gritou "Ei, Siri", despertando o sistema de reconhecimento de voz do telefone. Então Mu comandou "rota para a cafeteria *Blue Bottle*". O telefone rapidamente exibiu a transcrição de seu comando. Ele também reconheceu que estávamos pedindo direções e abriu o aplicativo *Maps* (app) para cumprir nosso pedido. Depois de laberto, o aplicativo *Maps* identificou várias rotas. Ao lado de cada rota, o telefone exibia um tempo de trânsito previsto. Enquanto fabricamos esta história por conveniência pedagógica, isso demonstra que no intervalo de apenas alguns segundos, nossas interações diárias com um telefone inteligente podem envolver vários modelos de *machine learning*.

Imagine apenas escrever um programa para responder a uma *palavra de alerta* como "Alexa", "OK Google" e "Hey Siri". Tente codificar em uma sala sozinho com nada além de um computador e um editor de código, conforme ilustrado em: numref: `fig_wake_word`. Como você escreveria tal programa a partir dos primeiros princípios? Pense nisso ... o problema é difícil. A cada segundo, o microfone irá coletar aproximadamente 44.000 amostras. Cada amostra é uma medida da amplitude da onda sonora. Que regra poderia mapear de forma confiável, de um trecho de áudio bruto a previsões confiáveis $\{\text{yes}, \text{no}\}$ sobre se o trecho de áudio contém a palavra de ativação? Se você estiver travado, não se preocupe. Também não sabemos escrever tal programa do zero. É por isso que usamos o *machine learning*.

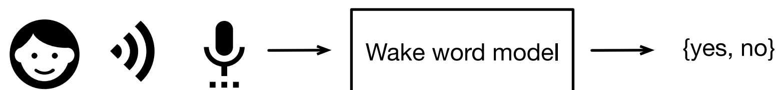


Fig. 1.1.1: Identificar uma palavra de ativação.

Aqui está o truque. Muitas vezes, mesmo quando não sabemos como dizer a um computador explicitamente como mapear de entradas para saídas, ainda assim, somos capazes de realizar a façanha cognitiva por nós mesmos. Em outras palavras, mesmo que você não saiba como programar um computador para reconhecer a palavra "Alexa", você mesmo é capaz de reconhecê-lo. Armados com essa habilidade, podemos coletar um enorme *dataset* contendo exemplos de áudio e rotular aqueles que contêm e que não contêm a palavra de ativação. Na abordagem de *machine learning*, não tentamos projetar um sistema *explicitamente* para reconhecer palavras de ativação.

Em vez disso, definimos um programa flexível cujo comportamento é determinado por vários *parâmetros*. Em seguida, usamos o conjunto de dados para determinar o melhor conjunto possível de parâmetros, aqueles que melhoram o desempenho do nosso programa com respeito a alguma medida de desempenho na tarefa de interesse.

Você pode pensar nos parâmetros como botões que podemos girar, manipulando o comportamento do programa. Fixando os parâmetros, chamamos o programa de *modelo*. O conjunto de todos os programas distintos (mapeamentos de entrada-saída) que podemos produzir apenas manipulando os parâmetros é chamada de *família* de modelos. E o meta-programa que usa nosso conjunto de dados para escolher os parâmetros é chamado de *algoritmo de aprendizagem*.

Antes de prosseguirmos e envolvermos o algoritmo de aprendizagem, temos que definir o problema com precisão, identificando a natureza exata das entradas e saídas, e escolher uma família modelo apropriada. Nesse caso, nosso modelo recebe um trecho de áudio como *entrada*, e o modelo gera uma seleção entre $\{\text{yes}, \text{no}\}$ como *saída*. Se tudo correr de acordo com o plano as suposições da modelo vão normalmente estar corretas quanto a se o áudio contém a palavra de ativação.

Se escolhermos a família certa de modelos, deve haver uma configuração dos botões de forma que o modelo dispara "sim" toda vez que ouve a palavra "Alexa". Como a escolha exata da palavra de ativação é arbitrária, provavelmente precisaremos de uma família modelo suficientemente rica que, por meio de outra configuração dos botões, ele poderia disparar "sim" somente ao ouvir a palavra "Damasco". Esperamos que a mesma família de modelo seja adequada para reconhecimento "Alexa" e reconhecimento "Damasco" porque parecem, intuitivamente, tarefas semelhantes. No entanto, podemos precisar de uma família totalmente diferente de modelos se quisermos lidar com entradas ou saídas fundamentalmente diferentes, digamos que se quiséssemos mapear de imagens para legendas, ou de frases em inglês para frases em chinês.

Como você pode imaginar, se apenas definirmos todos os botões aleatoriamente, é improvável que nosso modelo reconheça "Alexa", "Apricot", ou qualquer outra palavra em inglês. No *machine learning*, o *aprendizado* (*learning*) é o processo pelo qual descobrimos a configuração certa dos botões coagindo o comportamento desejado de nosso modelo. Em outras palavras, nós *treinamos* nosso modelo com dados. Conforme mostrado em: numref:`fig_ml_loop`, o processo de treinamento geralmente se parece com o seguinte:

1. Comece com um modelo inicializado aleatoriamente que não pode fazer nada útil.
2. Pegue alguns de seus dados (por exemplo, trechos de áudio e *labels* $\{\text{yes}, \text{no}\}$ correspondentes).
3. Ajuste os botões para que o modelo seja menos ruim em relação a esses exemplos.
4. Repita as etapas 2 e 3 até que o modelo esteja incrível.

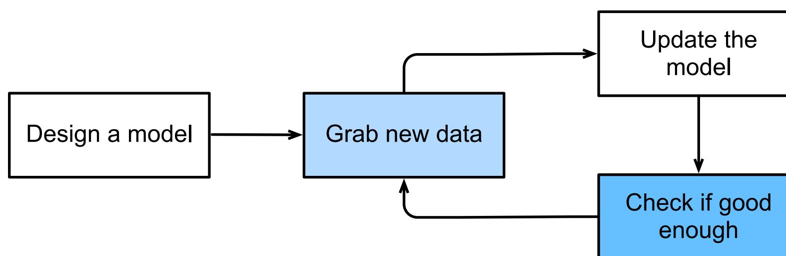


Fig. 1.1.2: Um processo de treinamento típico.

Para resumir, em vez de codificar um reconhecedor de palavra de acionamento, nós codificamos

um programa que pode *aprender* a reconhecê-las se o apresentarmos com um grande *dataset* rotulado. Você pode pensar neste ato de determinar o comportamento de um programa apresentando-o com um *dataset* como *programação com dados*. Quer dizer, podemos "programar" um detector de gatos, fornecendo nosso sistema de aprendizado de máquina com muitos exemplos de cães e gatos. Dessa forma, o detector aprenderá a emitir um número positivo muito grande se for um gato, um número negativo muito grande se for um cachorro, e algo mais próximo de zero se não houver certeza, e isso é apenas a ponta do *iceberg* do que o *machine learning* pode fazer. *Deep learning*, que iremos explicar em maiores detalhes posteriormente, é apenas um entre muitos métodos populares para resolver problemas de *machine learning*.

1.2 Componentes chave

Em nosso exemplo de palavra de ativação, descrevemos um *dataset* consistindo em trechos de áudio e *labels* binários, e nós demos uma sensação ondulante de como podemos treinar um modelo para aproximar um mapeamento de áudios para classificações. Esse tipo de problema, onde tentamos prever um *label* desconhecido designado com base em entradas conhecidas dado um conjunto de dados que consiste em exemplos para os quais os rótulos são conhecidos, é chamado de *aprendizagem supervisionada*. Esse é apenas um entre muitos tipos de problemas de *machine learning*. Posteriormente, mergulharemos profundamente em diferentes problemas de *machine learning*. Primeiro, gostaríamos de lançar mais luz sobre alguns componentes principais que nos acompanharão, independentemente do tipo de problema de *machine learning* que enfrentarmos:

1. Os *dados* com os quais podemos aprender.
2. Um *modelo* de como transformar os dados.
3. Uma *função objetivo* que quantifica o quão bem (ou mal) o modelo está indo.
4. Um *algoritmo* para ajustar os parâmetros do modelo para otimizar a função objetivo.

1.2.1 Dados

Nem é preciso dizer que você não pode fazer ciência de dados sem dados. Podemos perder centenas de páginas pensando no que exatamente constitui os dados, mas por agora, vamos errar no lado prático e focar nas principais propriedades com as quais se preocupar. Geralmente, estamos preocupados com uma coleção de exemplos. Para trabalhar com dados de maneira útil, nós tipicamente precisamos chegar a uma representação numérica adequada. Cada *exemplo* (ou *ponto de dados*, *instância de dados*, *amostra*) normalmente consiste em um conjunto de atributos chamados *recursos* (ou *covariáveis*), a partir do qual o modelo deve fazer suas previsões. Nos problemas de aprendizagem supervisionada acima, a coisa a prever é um atributo especial que é designado como o *rótulo* (*label*) (ou *alvo*).

Se estivéssemos trabalhando com dados de imagem, cada fotografia individual pode constituir um exemplo, cada um representado por uma lista ordenada de valores numéricos correspondendo ao brilho de cada pixel. Uma fotografia colorida de 200×200 consistiria em $200 \times 200 \times 3 = 120000$ valores numéricos, correspondentes ao brilho dos canais vermelho, verde e azul para cada pixel. Em outra tarefa tradicional, podemos tentar prever se um paciente vai sobreviver ou não, dado um conjunto padrão de recursos, como idade, sinais vitais e diagnósticos.

Quando cada exemplo é caracterizado pelo mesmo número de valores numéricos, dizemos que os dados consistem em vetores de comprimento fixo e descrevemos o comprimento constante dos vetores como a *dimensionalidade* dos dados. Como você pode imaginar, o comprimento fixo

pode ser uma propriedade conveniente. Se quiséssemos treinar um modelo para reconhecer o câncer em imagens microscópicas, entradas de comprimento fixo significam que temos uma coisa a menos com que nos preocupar.

No entanto, nem todos os dados podem ser facilmente representados como vetores de *comprimento fixo*. Embora possamos esperar que as imagens do microscópio venham de equipamentos padrão, não podemos esperar imagens extraídas da Internet aparecerem todas com a mesma resolução ou formato. Para imagens, podemos considerar cortá-los todas em um tamanho padrão, mas essa estratégia só nos leva até certo ponto. Corremos o risco de perder informações nas partes cortadas. Além disso, os dados de texto resistem a representações de comprimento fixo ainda mais obstinadamente. Considere os comentários de clientes deixados em sites de comércio eletrônico como Amazon, IMDB e TripAdvisor. Alguns são curtos: "é uma porcaria!". Outros vagam por páginas. Uma das principais vantagens do *deep learning* sobre os métodos tradicionais é a graça comparativa com a qual os modelos modernos podem lidar com dados de *comprimento variável*.

Geralmente, quanto mais dados temos, mais fácil se torna nosso trabalho. Quando temos mais dados, podemos treinar modelos mais poderosos e dependem menos de suposições pré-concebidas. A mudança de regime de (comparativamente) pequeno para *big data* é um dos principais contribuintes para o sucesso do *deep learning* moderno. Para esclarecer, muitos dos modelos mais interessantes de *deep learning* não funcionam sem grandes *datasets*. Alguns outros trabalham no regime de pequenos dados, mas não são melhores do que as abordagens tradicionais.

Por fim, não basta ter muitos dados e processá-los com inteligência. Precisamos dos dados *certos*. Se os dados estiverem cheios de erros, ou se os recursos escolhidos não são preditivos da quantidade alvo de interesse, o aprendizado vai falhar. A situação é bem capturada pelo clichê: *entra lixo, sai lixo*. Além disso, o desempenho preditivo ruim não é a única consequência potencial. Em aplicativos sensíveis de *machine learning*, como policiamento preditivo, triagem de currículo e modelos de risco usados para empréstimos, devemos estar especialmente alertas para as consequências de dados inúteis. Um modo de falha comum ocorre em conjuntos de dados onde alguns grupos de pessoas não são representados nos dados de treinamento. Imagine aplicar um sistema de reconhecimento de câncer de pele na natureza que nunca tinha visto pele negra antes. A falha também pode ocorrer quando os dados não apenas sub-representem alguns grupos mas refletem preconceitos sociais. Por exemplo, se as decisões de contratação anteriores forem usadas para treinar um modelo preditivo que será usado para selecionar currículos, então os modelos de aprendizado de máquina poderiam inadvertidamente capturar e automatizar injustiças históricas. Observe que tudo isso pode acontecer sem o cientista de dados conspirar ativamente, ou mesmo estar ciente.

1.2.2 Modelos

A maior parte do *machine learning* envolve transformar os dados de alguma forma. Talvez queiramos construir um sistema que ingere fotos e preveja sorrisos. Alternativamente, podemos querer ingerir um conjunto de leituras de sensor e prever quão normais ou anômalas são as leituras. Por *modelo*, denotamos a maquinaria computacional para ingestão de dados de um tipo, e cuspir previsões de um tipo possivelmente diferente. Em particular, estamos interessados em modelos estatísticos que podem ser estimados a partir de dados. Embora os modelos simples sejam perfeitamente capazes de abordar problemas apropriadamente simples, os problemas nos quais nos concentramos neste livro, ampliam os limites dos métodos clássicos. O *deep learning* é diferenciado das abordagens clássicas principalmente pelo conjunto de modelos poderosos em que se concentra. Esses modelos consistem em muitas transformações sucessivas dos dados que

são encadeados de cima para baixo, daí o nome *deep learning*. No caminho para discutir modelos profundos, também discutiremos alguns métodos mais tradicionais.

1.2.3 Funções Objetivo

Anteriormente, apresentamos o *machine learning* como aprendizado com a experiência. Por *aprender* aqui, queremos dizer melhorar em alguma tarefa ao longo do tempo. Mas quem pode dizer o que constitui uma melhoria? Você pode imaginar que poderíamos propor a atualização do nosso modelo, e algumas pessoas podem discordar sobre se a atualização proposta constituiu uma melhoria ou um declínio.

A fim de desenvolver um sistema matemático formal de máquinas de aprendizagem, precisamos ter medidas formais de quão bons (ou ruins) nossos modelos são. No *machine learning*, e na otimização em geral, chamamos elas de *funções objetivo*. Por convenção, geralmente definimos funções objetivo de modo que quanto menor, melhor. Esta é apenas uma convenção. Você pode assumir qualquer função para a qual mais alto é melhor, e transformá-la em uma nova função que é qualitativamente idêntica, mas para a qual menor é melhor, invertendo o sinal. Porque quanto menor é melhor, essas funções às vezes são chamadas *funções de perda* (*loss functions*).

Ao tentar prever valores numéricos, a função de perda mais comum é *erro quadrático*, ou seja, o quadrado da diferença entre a previsão e a verdade fundamental. Para classificação, o objetivo mais comum é minimizar a taxa de erro, ou seja, a fração de exemplos em que nossas previsões discordam da verdade fundamental. Alguns objetivos (por exemplo, erro quadrático) são fáceis de otimizar. Outros (por exemplo, taxa de erro) são difíceis de otimizar diretamente, devido à indiferenciabilidade ou outras complicações. Nesses casos, é comum otimizar um *objetivo substituto*.

Normalmente, a função de perda é definida no que diz respeito aos parâmetros do modelo e depende do conjunto de dados. Nós aprendemos os melhores valores dos parâmetros do nosso modelo minimizando a perda incorrida em um conjunto consistindo em alguns exemplos coletados para treinamento. No entanto, indo bem nos dados de treinamento não garante que teremos um bom desempenho com dados não vistos. Portanto, normalmente queremos dividir os dados disponíveis em duas partições: o *dataset de treinamento* (ou *conjunto de treinamento*, para ajustar os parâmetros do modelo) e o *dataset de teste* (ou *conjunto de teste*, que é apresentado para avaliação), relatando o desempenho do modelo em ambos. Você pode pensar no desempenho do treinamento como sendo as pontuações de um aluno em exames práticos usado para se preparar para algum exame final real. Mesmo que os resultados sejam encorajadores, isso não garante sucesso no exame final. Em outras palavras, o desempenho do teste pode divergir significativamente do desempenho do treinamento. Quando um modelo tem um bom desempenho no conjunto de treinamento mas falha em generalizar para dados invisíveis, dizemos que está fazendo *overfitting*. Em termos da vida real, é como ser reprovado no exame real apesar de ir bem nos exames práticos.

1.2.4 Algoritmos de Otimização

Assim que tivermos alguma fonte de dados e representação, um modelo e uma função objetivo bem definida, precisamos de um algoritmo capaz de pesquisar para obter os melhores parâmetros possíveis para minimizar a função de perda. Algoritmos de otimização populares para aprendizagem profunda baseiam-se em uma abordagem chamada *gradiente descendente*. Em suma, em cada etapa, este método verifica, para cada parâmetro, para que lado a perda do conjunto de treinamento se moveria se você perturbou esse parâmetro apenas um pouco. Em seguida, atualiza o parâmetro na direção que pode reduzir a perda.

1.3 Tipos de Problemas de *Machine Learning*

O problema da palavra de ativação em nosso exemplo motivador é apenas um entre muitos problemas que o *machine learning* pode resolver. Para motivar ainda mais o leitor e nos fornecer uma linguagem comum quando falarmos sobre mais problemas ao longo do livro, a seguir nós listamos uma amostra dos problemas de *machine learning*. Estaremos constantemente nos referindo a nossos conceitos acima mencionados como dados, modelos e técnicas de treinamento.

1.3.1 Aprendizagem Supervisionada

A aprendizagem supervisionada (*supervised learning*) aborda a tarefa de prever *labels* com recursos de entrada. Cada par recurso-rótulo é chamado de exemplo. Às vezes, quando o contexto é claro, podemos usar o termo *exemplos* para se referir a uma coleção de entradas, mesmo quando os *labels* correspondentes são desconhecidos. Nosso objetivo é produzir um modelo que mapeia qualquer entrada para uma previsão de *label*.

Para fundamentar esta descrição em um exemplo concreto, se estivéssemos trabalhando na área de saúde, então podemos querer prever se um paciente teria um ataque cardíaco ou não. Esta observação, "ataque cardíaco" ou "sem ataque cardíaco", seria nosso *label*. Os recursos de entrada podem ser sinais vitais como frequência cardíaca, pressão arterial diastólica, e pressão arterial sistólica.

A supervisão entra em jogo porque para a escolha dos parâmetros, nós (os supervisores) fornecemos ao modelo um conjunto de dados consistindo em exemplos rotulados, onde cada exemplo é correspondido com o *label* da verdade fundamental. Em termos probabilísticos, normalmente estamos interessados em estimar a probabilidade condicional de determinados recursos de entrada de um *label*. Embora seja apenas um entre vários paradigmas no *machine learning*, a aprendizagem supervisionada é responsável pela maioria das bem-sucedidas aplicações de *machine learning* na indústria. Em parte, isso ocorre porque muitas tarefas importantes podem ser descritas nitidamente como estimar a probabilidade de algo desconhecido dado um determinado *dataset* disponível:

- Prever câncer versus não câncer, dada uma imagem de tomografia computadorizada.
- Prever a tradução correta em francês, dada uma frase em inglês.
- Prever o preço de uma ação no próximo mês com base nos dados de relatórios financeiros deste mês.

Mesmo com a descrição simples "previsão de *labels* com recursos de entrada" a aprendizagem supervisionada pode assumir muitas formas e exigem muitas decisões de modelagem, dependendo (entre outras considerações) do tipo, tamanho, e o número de entradas e saídas. Por exemplo,

usamos diferentes modelos para processar sequências de comprimentos arbitrários e para processar representações de vetores de comprimento fixo. Visitaremos muitos desses problemas em profundidade ao longo deste livro.

Informalmente, o processo de aprendizagem se parece com o seguinte. Primeiro, pegue uma grande coleção de exemplos para os quais os recursos são conhecidos e selecione deles um subconjunto aleatório, adquirindo os *labels* da verdade fundamental para cada um. Às vezes, esses *labels* podem ser dados disponíveis que já foram coletados (por exemplo, um paciente morreu no ano seguinte?) e outras vezes, podemos precisar empregar anotadores humanos para rotular os dados, (por exemplo, atribuição de imagens a categorias). Juntas, essas entradas e os *labels* correspondentes constituem o conjunto de treinamento. Alimentamos o *dataset* de treinamento em um algoritmo de aprendizado supervisionado, uma função que recebe como entrada um conjunto de dados e produz outra função: o modelo aprendido. Finalmente, podemos alimentar entradas não vistas anteriormente para o modelo aprendido, usando suas saídas como previsões do rótulo correspondente. O processo completo é desenhado em: numref: `fig_supervised_learning`.

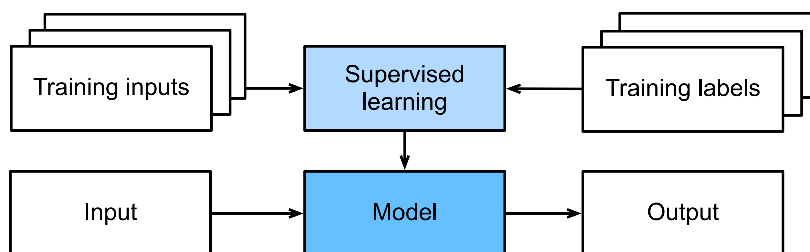


Fig. 1.3.1: Aprendizagem supervisionada.

Regressão

Talvez a tarefa de aprendizagem supervisionada mais simples para entender é *regressão*. Considere, por exemplo, um conjunto de dados coletados de um banco de dados de vendas de casas. Podemos construir uma mesa, onde cada linha corresponde a uma casa diferente, e cada coluna corresponde a algum atributo relevante, como a metragem quadrada de uma casa, o número de quartos, o número de banheiros e o número de minutos (caminhando) até o centro da cidade. Neste conjunto de dados, cada exemplo seria uma casa específica, e o vetor de recurso correspondente seria uma linha na tabela. Se você mora em Nova York ou São Francisco, e você não é o CEO da Amazon, Google, Microsoft ou Facebook, o vetor de recursos (metragem quadrada, nº de quartos, nº de banheiros, distância a pé) para sua casa pode ser algo como: $ [56, 1, 1, 60] $. No entanto, se você mora em Pittsburgh, pode ser parecido com $ [279, 4, 3, 10] $. Vetores de recursos como este são essenciais para a maioria dos algoritmos clássicos de *machine learning*.

O que torna um problema uma regressão é, na verdade, o resultado. Digamos que você esteja em busca de uma nova casa. Você pode querer estimar o valor justo de mercado de uma casa, dados alguns recursos como acima. O *label*, o preço de venda, é um valor numérico. Quando os *labels* assumem valores numéricos arbitrários, chamamos isso de problema de *regressão*. Nosso objetivo é produzir um modelo cujas previsões aproximar os valores reais do *label*.

Muitos problemas práticos são problemas de regressão bem descritos. Prever a avaliação que um usuário atribuirá a um filme pode ser pensado como um problema de regressão e se você projetou um ótimo algoritmo para realizar essa façanha em 2009, você pode ter ganho o [prêmio de 1 milhão de dólares da Netflix] (https://en.wikipedia.org/wiki/Netflix_Prize). Previsão do tempo de permanência de pacientes no hospital também é um problema de regressão. Uma boa regra é: qualquer problema de *quanto?* Ou *quantos?* deve sugerir regressão, tal como:

- Quantas horas durará esta cirurgia?
- Quanta chuva esta cidade terá nas próximas seis horas?

Mesmo que você nunca tenha trabalhado com *machine learning* antes, você provavelmente já trabalhou em um problema de regressão informalmente. Imagine, por exemplo, que você mandou consertar seus ralos e que seu contratante gastou 3 horas removendo sujeira de seus canos de esgoto. Então ele lhe enviou uma conta de 350 dólares. Agora imagine que seu amigo contratou o mesmo empreiteiro por 2 horas e que ele recebeu uma nota de 250 dólares. Se alguém lhe perguntasse quanto esperar em sua próxima fatura de remoção de sujeira você pode fazer algumas suposições razoáveis, como mais horas trabalhadas custam mais dólares. Você também pode presumir que há alguma carga básica e que o contratante cobra por hora. Se essas suposições forem verdadeiras, dados esses dois exemplos de dados, você já pode identificar a estrutura de preços do contratante: 100 dólares por hora mais 50 dólares para aparecer em sua casa. Se você acompanhou esse exemplo, então você já entendeu a ideia de alto nível por trás da regressão linear.

Neste caso, poderíamos produzir os parâmetros que correspondem exatamente aos preços do contratante. Às vezes isso não é possível, por exemplo, se alguma variação se deve a algum fator além dos dois citados. Nestes casos, tentaremos aprender modelos que minimizam a distância entre nossas previsões e os valores observados. Na maioria de nossos capítulos, vamos nos concentrar em minimizar a função de perda de erro quadrático. Como veremos mais adiante, essa perda corresponde ao pressuposto que nossos dados foram corrompidos pelo ruído gaussiano.

Classificação

Embora os modelos de regressão sejam ótimos para responder às questões *quantos?*, muitos problemas não se adaptam confortavelmente a este modelo. Por exemplo, um banco deseja adicionar a digitalização de cheques ao seu aplicativo móvel. Isso envolveria o cliente tirando uma foto de um cheque com a câmera do smartphone deles e o aplicativo precisaria ser capaz de entender automaticamente o texto visto na imagem. Especificamente, também precisaria entender o texto manuscrito para ser ainda mais robusto, como mapear um caractere escrito à mão a um dos personagens conhecidos. Este tipo de problema de *qual?* É chamado de *classificação*. É tratado com um conjunto diferente de algoritmos do que aqueles usados para regressão, embora muitas técnicas sejam transportadas.

Na *classificação*, queremos que nosso modelo analise os recursos, por exemplo, os valores de pixel em uma imagem, e, em seguida, prever qual *categoria* (formalmente chamada de *classe*), entre alguns conjuntos discretos de opções, um exemplo pertence. Para dígitos manuscritos, podemos ter dez classes, correspondendo aos dígitos de 0 a 9. A forma mais simples de classificação é quando existem apenas duas classes, um problema que chamamos de *classificação binária*. Por exemplo, nosso conjunto de dados pode consistir em imagens de animais e nossos rótulos podem ser as classes $\{\mathrm{cat}, \mathrm{dog}\}$. Durante a regressão, buscamos um regressor para produzir um valor numérico, na classificação, buscamos um classificador, cuja saída é a atribuição de classe prevista.

Por razões que abordaremos à medida que o livro se torna mais técnico, pode ser difícil otimizar um modelo que só pode produzir uma tarefa categórica difícil, por exemplo, "gato" ou "cachorro". Nesses casos, geralmente é muito mais fácil expressar nosso modelo na linguagem das probabilidades. Dados os recursos de um exemplo, nosso modelo atribui uma probabilidade para cada classe possível. Voltando ao nosso exemplo de classificação animal onde as classes são $\{\mathrm{gato}, \mathrm{cachorro}\}$, um classificador pode ver uma imagem e gerar a probabilidade que a imagem é um gato como 0,9. Podemos interpretar esse número dizendo que o classificador tem 90% de certeza de que a imagem representa um gato. A magnitude da probabilidade para a classe prevista

transmite uma noção de incerteza. Esta não é a única noção de incerteza e discutiremos outros em capítulos mais avançados.

Quando temos mais de duas classes possíveis, chamamos o problema de *classificação multiclasse*. Exemplos comuns incluem reconhecimento de caracteres escritos à mão $\{0, 1, 2, ...9, \mathrm{a}, \mathrm{b}, \mathrm{c}, ...\}$. Enquanto atacamos problemas de regressão tentando minimizar a função de perda de erro quadrático, a função de perda comum para problemas de classificação é chamada de *entropia cruzada* (*cross-entropy*), cujo nome pode ser desmistificado por meio de uma introdução à teoria da informação nos capítulos subsequentes.

Observe que a classe mais provável não é necessariamente aquela que você usará para sua decisão. Suponha que você encontre um lindo cogumelo em seu quintal como mostrado em: numref: `fig_death_cap`.

Fig. 1.3.2: Cicuta verde[1] — não coma!

Agora, suponha que você construiu um classificador e o treinou para prever se um cogumelo é venenoso com base em uma fotografia. Digamos que nossos resultados do classificador de detecção de veneno que a probabilidade de que : numref: `fig_death_cap` contém um Cicuta verde de 0,2. Em outras palavras, o classificador tem 80% de certeza que nosso cogumelo não é um Cicuta verde. Ainda assim, você teria que ser um tolo para comê-lo. Isso porque o certo benefício de um jantar delicioso não vale a pena um risco de 20% de morrer por causa disso. Em outras palavras, o efeito do risco incerto supera o benefício de longe. Assim, precisamos calcular o risco esperado que incorremos como a função de perda, ou seja, precisamos multiplicar a probabilidade do resultado com o benefício (ou dano) associado a ele. Nesse caso, a perda incorrida ao comer o cogumelo pode ser $0,2 \times \infty + 0,8 \times 0 = \infty$, Considerando que a perda de descarte é $0,2 \times 0 + 0,8 \times 1 = 0,8$. Nossa cautela foi justificada: como qualquer micologista nos diria, o cogumelo em: numref: `fig_death_cap` na verdade é um Cicuta verde.

A classificação pode ser muito mais complicada do que apenas classificação binária, multi-classe ou mesmo com vários rótulos. Por exemplo, existem algumas variantes de classificação para abordar hierarquias. As hierarquias assumem que existem alguns relacionamentos entre as muitas classes. Portanto, nem todos os erros são iguais — se devemos errar, preferiríamos classificar incorretamente para uma classe parecida em vez de uma classe distante. Normalmente, isso é conhecido como *classificação hierárquica*. Um exemplo inicial é devido a [Linnaeus] (https://en.wikipedia.org/wiki/Carl_Linnaeus), que organizou os animais em uma hierarquia.

[1] O cogumelo *Amanita phalloides*, cujo nome popular em inglês é *Death cap*, foi traduzido para o nome popular em português, Cicuta verde.

No caso da classificação animal, pode não ser tão ruim confundir um poodle (uma raça de cachorro) com um schnauzer (outra raça de cachorro), mas nosso modelo pagaria uma grande penalidade se confundisse um poodle com um dinossauro. Qual hierarquia é relevante pode depender sobre como você planeja usar o modelo. Por exemplo, cascavéis e cobras-liga podem estar perto da árvore filogenética, mas confundir uma cascavel com uma cobra-liga pode ser mortal.

Tags

Alguns problemas de classificação se encaixam perfeitamente nas configurações de classificação binária ou multiclasse. Por exemplo, podemos treinar um classificador binário normal para distinguir gatos de cães. Dado o estado atual da visão computacional, podemos fazer isso facilmente, com ferramentas disponíveis no mercado. No entanto, não importa o quão preciso seja o nosso modelo, podemos ter problemas quando o classificador encontra uma imagem dos *Músicos da Cidade de Bremen*, um conto de fadas alemão popular com quatro animais in: numref: `fig_stackedanimals`.

Fig. 1.3.3: Um burro, um cachorro, um gato e um galo.

Como você pode ver, há um gato em: numref: `fig_stackedanimals`, e um galo, um cachorro e um burro, com algumas árvores ao fundo. Dependendo do que queremos fazer com nosso modelo em última análise, tratando isso como um problema de classificação binária pode não fazer muito sentido. Em vez disso, podemos dar ao modelo a opção de dizer que a imagem retrata um gato, um cachorro, um burro, *e* um galo.

O problema de aprender a prever classes que são não mutuamente exclusivas é chamado de *classificação multi-rótulo*. Os problemas de *tags* automáticas são geralmente mais bem descritos como

problemas de classificação multi-rótulo. Pense nas *tags* que as pessoas podem aplicar a postagens em um blog técnico, por exemplo, "*machine learning*", "tecnologia", "*gadgets*", "linguagens de programação", "Linux", "computação em nuvem", "AWS". Um artigo típico pode ter de 5 a 10 *tags* aplicadas porque esses conceitos estão correlacionados. Postagens sobre "computação em nuvem" provavelmente mencionarão "AWS" e postagens sobre "*machine learning*" também podem tratar de "linguagens de programação".

Também temos que lidar com esse tipo de problema ao lidar com a literatura biomédica, onde etiquetar corretamente os artigos é importante porque permite que os pesquisadores façam revisões exaustivas da literatura. Na *National Library of Medicine*, vários anotadores profissionais revisam cada artigo que é indexado no *PubMed* para associá-lo aos termos relevantes do MeSH, uma coleção de aproximadamente 28000 *tags*. Este é um processo demorado e o os anotadores normalmente têm um atraso de um ano entre o arquivamento e a definição das *tags*. O *machine learning* pode ser usado aqui para fornecer *tags* provisórias até que cada artigo possa ter uma revisão manual adequada. Na verdade, por vários anos, a organização BioASQ tem [sediado concursos] (http://bioasq.org/) para fazer exatamente isso.

Busca

Às vezes, não queremos apenas atribuir cada exemplo a um valor real. No campo da recuperação de informações, queremos impor uma classificação a um conjunto de itens. Tome como exemplo a pesquisa na web. O objetivo é menos determinar se uma página específica é relevante para uma consulta, mas, em vez disso, qual dentre a infinidade de resultados de pesquisa é mais relevante para um determinado usuário. Nós realmente nos preocupamos com a ordem dos resultados de pesquisa relevantes e nosso algoritmo de aprendizagem precisa produzir subconjuntos ordenados de elementos de um conjunto maior. Em outras palavras, se formos solicitados a produzir as primeiras 5 letras do alfabeto, há uma diferença entre retornar "A B C D E" e "C A B E D". Mesmo que o conjunto de resultados seja o mesmo, a ordenação dentro do conjunto importa.

Uma possível solução para este problema é primeiro atribuir para cada elemento no conjunto uma pontuação de relevância correspondente e, em seguida, para recuperar os elementos com melhor classificação. [PageRank] (https://en.wikipedia.org/wiki/PageRank), o molho secreto original por trás do mecanismo de pesquisa do Google foi um dos primeiros exemplos de tal sistema de pontuação, mas foi peculiar por não depender da consulta real. Aqui, eles contaram com um filtro de relevância simples para identificar o conjunto de itens relevantes e, em seguida, no PageRank para ordenar esses resultados que continham o termo de consulta. Hoje em dia, os mecanismos de pesquisa usam *machine learning* e modelos comportamentais para obter pontuações de relevância dependentes de consulta. Existem conferências acadêmicas inteiras dedicadas a este assunto.

Sistemas de Recomendação

Os sistemas de recomendação são outra configuração de problema que está relacionado à pesquisa e classificação. Os problemas são semelhantes na medida em que o objetivo é exibir um conjunto de itens relevantes para o usuário. A principal diferença é a ênfase em *personalização* para usuários específicos no contexto de sistemas de recomendação. Por exemplo, para recomendações de filmes, a página de resultados para um fã de ficção científica e a página de resultados para um conhecedor das comédias de Peter Sellers podem diferir significativamente. Problemas semelhantes surgem em outras configurações de recomendação, por exemplo, para produtos de varejo, música e recomendação de notícias.

Em alguns casos, os clientes fornecem *feedback* explícito comunicando o quanto eles gostaram de um determinado produto (por exemplo, as avaliações e resenhas de produtos na Amazon, IMDb e GoodReads). Em alguns outros casos, eles fornecem *feedback* implícito, por exemplo, pulando títulos em uma lista de reprodução, o que pode indicar insatisfação, mas pode apenas indicar que a música era inadequada no contexto. Nas formulações mais simples, esses sistemas são treinados para estimar alguma pontuação, como uma avaliação estimada ou a probabilidade de compra, dado um usuário e um item.

Dado esse modelo, para qualquer usuário, poderíamos recuperar o conjunto de objetos com as maiores pontuações, que pode então ser recomendado ao usuário. Os sistemas de produção são consideravelmente mais avançados e levam a atividade detalhada do usuário e características do item em consideração ao computar essas pontuações. : numref: `fig_deeplearning_amazon` é um exemplo de livros de *deep learning* recomendados pela Amazon com base em algoritmos de personalização ajustados para capturar as preferências de alguém.

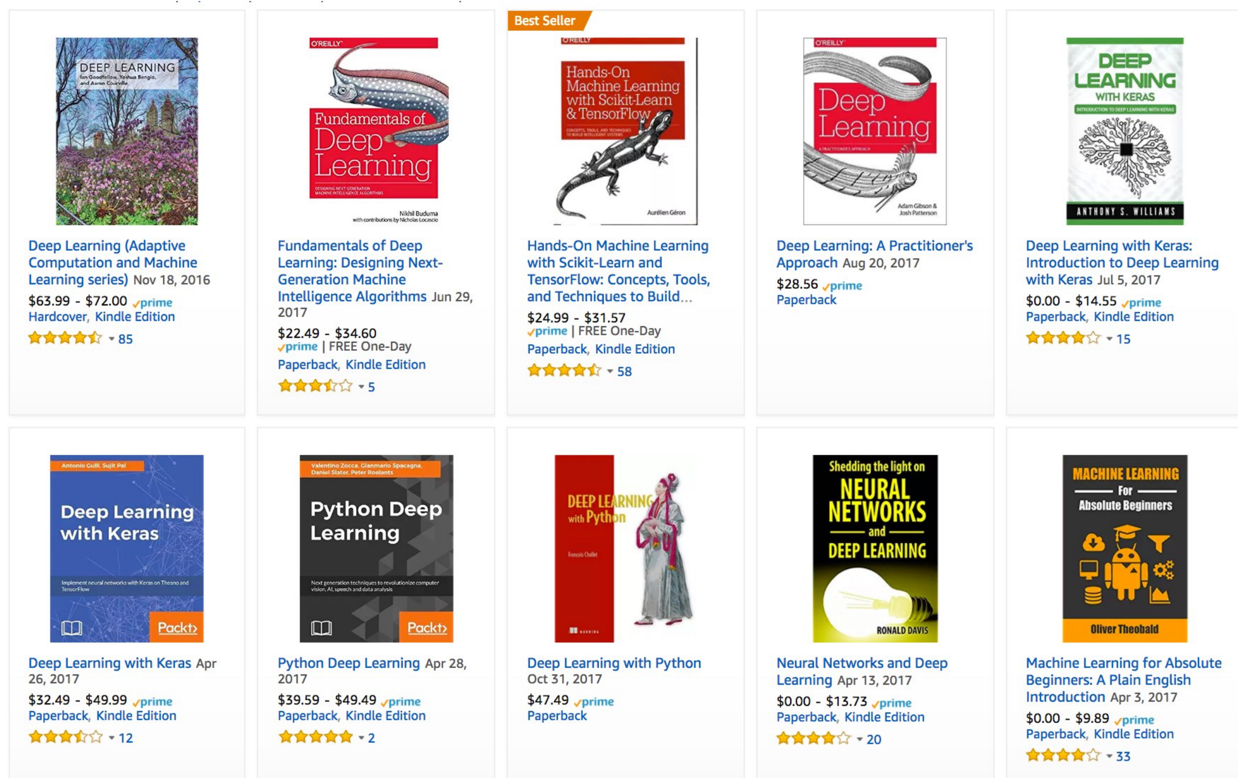


Fig. 1.3.4: Livros de *deep learning* recomendados pela Amazon.

Apesar de seu enorme valor econômico, sistemas de recomendação ingenuamente construídos em cima de modelos preditivos sofrem algumas falhas conceituais graves. Para começar, observamos apenas *feedback censurado*: os usuários avaliam preferencialmente os filmes que os consideram fortes. Por exemplo, em uma escala de cinco pontos, você pode notar que os itens recebem muitas classificações de cinco e uma estrela mas que existem visivelmente poucas avaliações de três estrelas. Além disso, os hábitos de compra atuais são muitas vezes um resultado do algoritmo de recomendação atualmente em vigor, mas os algoritmos de aprendizagem nem sempre levam esse detalhe em consideração. Assim, é possível que se formem ciclos de feedback onde um sistema de recomendação preferencialmente empurra um item que então é considerado melhor (devido a maiores compras) e, por sua vez, é recomendado com ainda mais frequência. Muitos desses problemas sobre como lidar com a censura, incentivos e ciclos de *feedback* são importantes

questões abertas de pesquisa.

Aprendizagem sequencial

Até agora, vimos problemas em que temos algum número fixo de entradas a partir dos quais produzimos um número fixo de saídas. Por exemplo, consideramos prever os preços das casas a partir de um conjunto fixo de recursos: metragem quadrada, número de quartos, número de banheiros, tempo de caminhada até o centro. Também discutimos o mapeamento de uma imagem (de dimensão fixa) às probabilidades previstas de que pertence a cada de um número fixo de classes, ou pegando um ID de usuário e um ID de produto, e prever uma classificação por estrelas. Nesses casos, uma vez que alimentamos nossa entrada de comprimento fixo no modelo para gerar uma saída, o modelo esquece imediatamente o que acabou de ver.

Isso pode ser bom se todas as nossas entradas realmente tiverem as mesmas dimensões e se as entradas sucessivas realmente não têm nada a ver umas com as outras. Mas como lidaríamos com trechos de vídeo? Nesse caso, cada fragmento pode consistir em um número diferente de quadros. E nosso palpite sobre o que está acontecendo em cada quadro pode ser muito mais forte se levarmos em consideração os quadros anteriores ou posteriores. O mesmo vale para a linguagem. Um problema popular de *deep learning* é tradução automática: a tarefa de ingerir frases em algum idioma de origem e prevendo sua tradução em outro idioma.

Esses problemas também ocorrem na medicina. Podemos querer um modelo para monitorar pacientes na unidade de terapia intensiva e disparar alertas se seus riscos de morte nas próximas 24 horas excederem algum limite. Definitivamente, não queremos que este modelo jogue fora tudo o que sabe sobre o histórico do paciente a cada hora e apenas fazer suas previsões com base nas medições mais recentes.

Esses problemas estão entre as aplicações mais interessantes de *machine learning* e são instâncias de *aprendizagem sequencial*. Eles exigem um modelo para ingerir sequências de entradas ou para emitir sequências de saídas (ou ambos). Especificamente, *sequência para aprendizagem de sequencial* considera os problemas onde entrada e saída são sequências de comprimento variável, como tradução automática e transcrição de texto da fala falada. Embora seja impossível considerar todos os tipos de transformações de sequência, vale a pena mencionar os seguintes casos especiais.

Marcação e análise. Isso envolve anotar uma sequência de texto com atributos. Em outras palavras, o número de entradas e saídas é essencialmente o mesmo. Por exemplo, podemos querer saber onde estão os verbos e os sujeitos. Como alternativa, podemos querer saber quais palavras são as entidades nomeadas. Em geral, o objetivo é decompor e anotar o texto com base na estrutura e suposições gramaticais para obter algumas anotações. Isso parece mais complexo do que realmente é. Abaixo está um exemplo muito simples de uma frase anotada com marcas que indicam quais palavras se referem a entidades nomeadas (marcadas como “Ent”).

```
Tom has dinner in Washington with Sally
Ent  -    -    -     Ent      -    Ent
```

Reconhecimento automático de fala. Com o reconhecimento de fala, a sequência de entrada é uma gravação de áudio de um alto-falante (mostrado em: numref: `fig_speech`), e a saída é a transcrição textual do que o locutor disse. O desafio é que existem muito mais quadros de áudio (o som é normalmente amostrado em 8kHz ou 16kHz) do que texto, ou seja, não há correspondência 1: 1 entre áudio e texto, já que milhares de amostras podem correspondem a uma única palavra falada. Estes são problemas de aprendizagem de sequência a sequência em que a saída é muito mais curta do que a entrada.

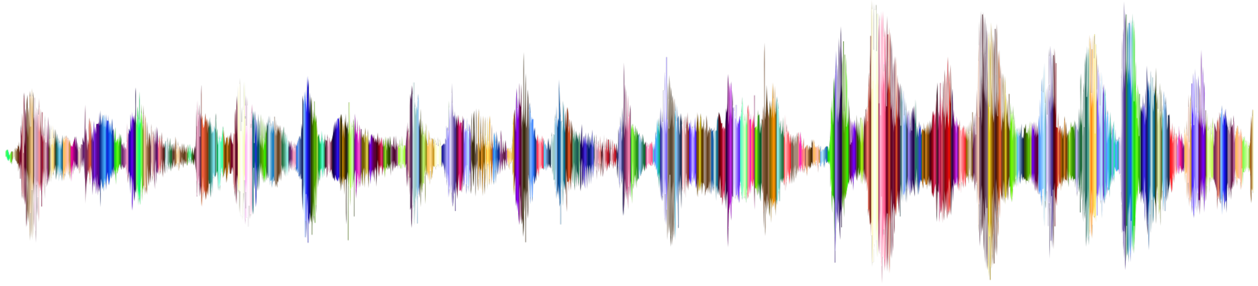


Fig. 1.3.5: `-D-e-e-p- L-ea-r-ni-ng-` in an audio recording.

Text to Speech (Texto para fala). Este é o inverso do reconhecimento automático de fala. Em outras palavras, a entrada é um texto e a saída é um arquivo de áudio. Nesse caso, a saída é muito mais longa do que a entrada. Embora seja fácil para os humanos reconhecerem um arquivo de áudio ruim, isso não é tão trivial para computadores.

Tradução por máquina. Ao contrário do caso do reconhecimento de voz, onde correspondente entradas e saídas ocorrem na mesma ordem (após o alinhamento), na tradução automática, a inversão da ordem pode ser vital. Em outras palavras, enquanto ainda estamos convertendo uma sequência em outra, nem o número de entradas e saídas, nem o pedido de exemplos de dados correspondentes são considerados iguais. Considere o seguinte exemplo ilustrativo da tendência peculiar dos alemães para colocar os verbos no final das frases.

```
German:           Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English:          Did you already check out this excellent tutorial?
Wrong alignment:  Did you yourself already this excellent tutorial looked-at?
```

Muitos problemas relacionados surgem em outras tarefas de aprendizagem. Por exemplo, determinar a ordem em que um usuário lê uma página da web é um problema de análise de *layout* bidimensional. Problemas de diálogo apresentam todos os tipos de complicações adicionais, onde determinar o que dizer a seguir requer levar em consideração conhecimento do mundo real e o estado anterior da conversa através de longas distâncias temporais. Estas são áreas ativas de pesquisa.

1.3.2 Aprendizagem não-supervisionada

Todos os exemplos até agora foram relacionados à aprendizagem supervisionada, ou seja, situações em que alimentamos o modelo com um conjunto de dados gigante contendo os recursos e os valores de rótulo correspondentes. Você pode pensar no aluno supervisionado como tendo um trabalho extremamente especializado e um chefe extremamente banal. O chefe fica por cima do seu ombro e lhe diz exatamente o que fazer em todas as situações até que você aprenda a mapear as de situações para ações. Trabalhar para um chefe assim parece muito chato. Por outro lado, é fácil agradar a esse chefe. Você apenas reconhece o padrão o mais rápido possível e imita suas ações.

De uma forma completamente oposta, pode ser frustrante trabalhar para um chefe que não tem ideia do que eles querem que você faça. No entanto, se você planeja ser um cientista de dados, é melhor se acostumar com isso. O chefe pode simplesmente entregar a você uma pilha gigante de dados e dizer para *fazer ciência de dados com eles!* Isso parece vago porque é. Chamamos essa classe de problemas de *aprendizagem não supervisionada*, e o tipo e número de perguntas que podemos

fazer é limitado apenas pela nossa criatividade. Abordaremos técnicas de aprendizado não supervisionado nos capítulos posteriores. Para abrir seu apetite por enquanto, descrevemos algumas das seguintes perguntas que você pode fazer.

- Podemos encontrar um pequeno número de protótipos que resumem os dados com precisão? Dado um conjunto de fotos, podemos agrupá-las em fotos de paisagens, fotos de cachorros, bebês, gatos e picos de montanhas? Da mesma forma, dada uma coleção de atividades de navegação dos usuários, podemos agrupá-los em usuários com comportamento semelhante? Esse problema é normalmente conhecido como *clusterização*.
- Podemos encontrar um pequeno número de parâmetros que capturam com precisão as propriedades relevantes dos dados? As trajetórias de uma bola são muito bem descritas pela velocidade, diâmetro e massa da bola. Os alfaiates desenvolveram um pequeno número de parâmetros que descrevem a forma do corpo humano com bastante precisão com o propósito de ajustar roupas. Esses problemas são chamados de *estimativa de subespaço*. Se a dependência for linear, é chamada de *análise de componentes principais (principal component analysis — PCA)*.
- Existe uma representação de objetos (estruturados arbitrariamente) no espaço euclidiano de modo que as propriedades simbólicas podem ser bem combinadas? Isso pode ser usado para descrever entidades e suas relações, como "Roma" $-$ "Itália" $+$ "França" $=$ "Paris".
- Existe uma descrição das causas comuns de muitos dos dados que observamos? Por exemplo, se tivermos dados demográficos sobre preços de casas, poluição, crime, localização, educação e salários, podemos descobrir como eles estão relacionados simplesmente com base em dados empíricos? Os campos relacionados com *causalidade* e *modelos gráficos probabilísticos* resolvem este problema.
- Outro importante e empolgante desenvolvimento recente na aprendizagem não supervisionada é o advento de *redes adversárias geradoras*. Isso nos dá uma maneira processual de sintetizar dados, até mesmo dados estruturados complicados, como imagens e áudio. Os mecanismos estatísticos subjacentes são testes para verificar se os dados reais e falsos são iguais.

1.3.3 Interagindo com um Ambiente

Até agora, não discutimos de onde os dados realmente vêm, ou o que realmente acontece quando um modelo de *machine learning* gera uma saída. Isso ocorre porque o aprendizado supervisionado e o aprendizado não supervisionado não tratam dessas questões de uma forma muito sofisticada. Em qualquer caso, pegamos uma grande pilha de dados antecipadamente, em seguida, colocamos nossas máquinas de reconhecimento de padrões em movimento sem nunca mais interagir com o ambiente novamente. Porque todo o aprendizado ocorre depois que o algoritmo é desconectado do ambiente, isso às vezes é chamado de *aprendizagem offline*. Para aprendizagem supervisionada, o processo considerando a coleta de dados de um ambiente se parece com: numref: `fig_data_collection`.

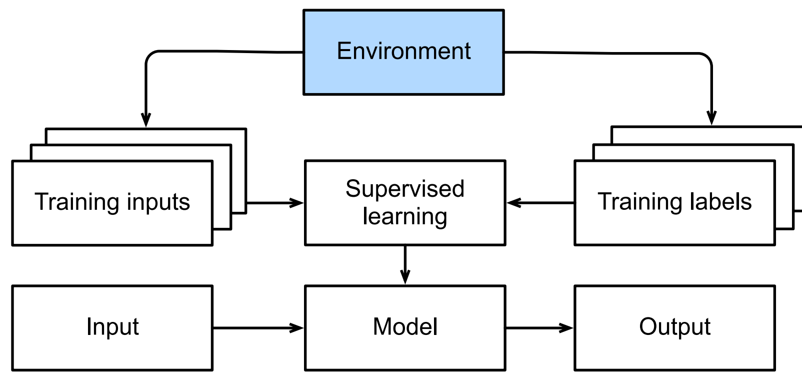


Fig. 1.3.6: Collecting data for supervised learning from an environment.

Esta simplicidade de aprendizagem offline tem seus encantos. A vantagem é que podemos nos preocupar com o reconhecimento de padrões isoladamente, sem qualquer distração desses outros problemas. Mas a desvantagem é que a formulação do problema é bastante limitadora. Se você é mais ambicioso, ou se cresceu lendo a série Robot de Asimov, então você pode imaginar *bots* com inteligência artificial, capazes não só de fazer previsões, mas também de realizar ações no mundo. Queremos pensar em *agentes* inteligentes, não apenas em modelos preditivos. Isso significa que precisamos pensar sobre como escolher *ações*, não apenas fazendo previsões. Além disso, ao contrário das previsões, ações realmente impactam o meio ambiente. Se quisermos treinar um agente inteligente, devemos levar em conta a maneira como suas ações podem impactar as observações futuras do agente.

Considerando a interação com um ambiente abre todo um conjunto de novas questões de modelagem. A seguir estão apenas alguns exemplos.

- O ambiente lembra o que fizemos anteriormente?
- O ambiente quer nos ajudar, por exemplo, um usuário lendo texto em um reconhecedor de fala?
- O ambiente quer nos derrotar, ou seja, um ambiente adversário, como filtragem de spam (contra spammers) ou um jogo (contra um oponente)?
- O ambiente não se preocupa?
- O ambiente tem mudanças dinâmicas? Por exemplo, os dados futuros sempre se parecem com o passado ou os padrões mudam com o tempo, naturalmente ou em resposta às nossas ferramentas automatizadas?

Esta última questão levanta o problema de *mudança de distribuição*, quando os dados de treinamento e teste são diferentes. É um problema que a maioria de nós já experimentou ao fazer exames escritos por um professor, enquanto a lição de casa foi composta por seus assistentes de ensino. A seguir, descreveremos brevemente o aprendizado por reforço, uma configuração que considera explicitamente as interações com um ambiente.

1.3.4 Aprendizado por Reforço

Se você estiver interessado em usar o *machine learning* para desenvolver um agente que interaja com um ambiente e tome medidas, então você provavelmente vai acabar com foco na *aprendizagem por reforço*. Isso pode incluir aplicações para robótica, para sistemas de diálogo, e até mesmo para desenvolver inteligência artificial (IA) para videogames. *Aprendizagem por reforço profundo*, que se aplica *deep learning* para problemas de aprendizagem de reforço, aumentou em popularidade. A revolucionária *deep Q-netfowk* que derrotou os humanos nos jogos da Atari usando apenas a entrada visual, e o programa AlphaGo que destronou o campeão mundial no jogo de tabuleiro Go são dois exemplos proeminentes.

A aprendizagem por reforço fornece uma declaração muito geral de um problema, em que um agente interage com um ambiente ao longo de uma série de etapas de tempo. Em cada etapa de tempo, o agente recebe alguma *observação* do ambiente e deve escolher uma *ação* que é posteriormente transmitido de volta para o ambiente por meio de algum mecanismo (às vezes chamado de atuador). Por fim, o agente recebe uma recompensa do meio ambiente. Este processo é ilustrado em: numref: `fig_rl-environment`. O agente então recebe uma observação subsequente, e escolhe uma ação subsequente e assim por diante. O comportamento de um agente de aprendizagem por reforço é governado por uma política. Em suma, uma *política* é apenas uma função que mapeia das observações do ambiente às ações. O objetivo da aprendizagem por reforço é produzir uma boa política.

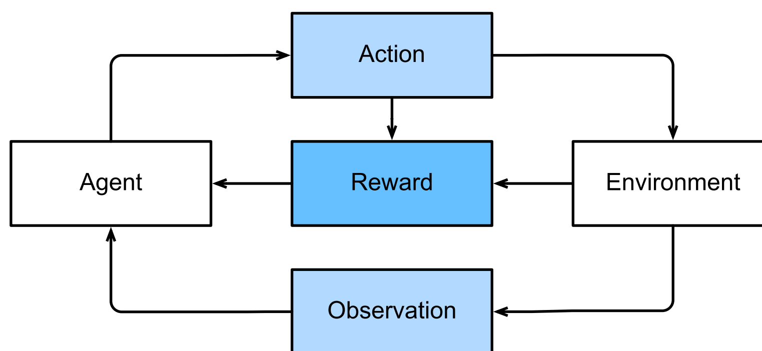


Fig. 1.3.7: The interaction between reinforcement learning and an environment.

É difícil exagerar a generalidade da estrutura de aprendizagem por reforço. Por exemplo, podemos lançar qualquer problema de aprendizado supervisionado como um problema de aprendizado por reforço. Digamos que tenhamos um problema de classificação. Poderíamos criar um agente de aprendizagem por reforço com uma ação correspondente a cada classe. Poderíamos então criar um ambiente que oferecesse uma recompensa que era exatamente igual à função de perda do problema original de aprendizagem supervisionada.

Dito isso, o aprendizado por reforço também pode resolver muitos problemas que a aprendizagem supervisionada não pode. Por exemplo, na aprendizagem supervisionada, sempre esperamos se a entrada de treinamento venha associada ao *label* correto. Mas na aprendizagem por reforço, não assumimos que para cada observação o ambiente nos diz a ação ideal. Em geral, apenas recebemos alguma recompensa. Além disso, o ambiente pode nem mesmo nos dizer quais ações levaram à recompensa.

Considere, por exemplo, o jogo de xadrez. O único sinal de recompensa real vem no final do jogo quando ganhamos, o que podemos atribuir a uma recompensa de 1, ou quando perdemos, o que podemos atribuir uma recompensa de -1. Assim, os agentes de reforço devem lidar com o problema de *atribuição de crédito*: determinar quais ações devem ser creditadas ou culpadas

por um resultado. O mesmo vale para o funcionário que for promovido no dia 11 de outubro. Essa promoção provavelmente reflete um grande número de ações bem escolhidas em relação ao ano anterior. Para obter mais promoções no futuro, é preciso descobrir quais ações ao longo do caminho levaram à promoção.

Agentes de aprendizado por reforço também podem ter que lidar com o problema da observabilidade parcial. Ou seja, a observação atual pode não dizer-lhe tudo sobre o seu estado atual. Digamos que um robô de limpeza ficou preso em um dos muitos armários idênticos em uma casa. Inferindo a localização precisa (e, portanto, o estado) do robô pode exigir a consideração de suas observações anteriores antes de entrar no armário.

Finalmente, em qualquer ponto, os agentes de reforço podem saber de uma boa política, mas pode haver muitas outras políticas melhores que o agente nunca tentou. O agente de reforço deve escolher constantemente se deve fazer um *exploit* com a melhor estratégia atualmente conhecida como uma política, ou um *explore* o espaço das estratégias, potencialmente desistindo de alguma recompensa de curto prazo em troca de conhecimento.

O problema geral de aprendizagem por reforço é uma configuração muito geral. As ações afetam as observações subsequentes. As recompensas só são observadas correspondendo às ações escolhidas. O ambiente pode ser total ou parcialmente observado. Levar em conta toda essa complexidade de uma vez pode exigir muito dos pesquisadores. Além disso, nem todo problema prático exibe toda essa complexidade. Como resultado, os pesquisadores estudaram uma série de casos especiais de problemas de aprendizagem por reforço.

Quando o ambiente é totalmente observado, chamamos o problema de aprendizagem por reforço de *processo de decisão Markov*. Quando o estado não depende das ações anteriores, chamamos o problema de *problema de bandido contextual*. Quando não há estado, apenas um conjunto de ações disponíveis com recompensas inicialmente desconhecidas, este problema é o clássico *problema de bandidos multi-armados*.

1.4 Raízes

Acabamos de revisar um pequeno subconjunto de problemas que o *machine learning* pode abordar. Para um conjunto diversificado de problemas de *machine learning*, o *deep learning* fornece ferramentas poderosas para resolvê-los. Embora muitos métodos de *deep learning* são invenções recentes, a ideia central da programação com dados e redes neurais (nomes de muitos modelos de *deep learning*) tem sido estudado há séculos. Na verdade, os humanos mantiveram o desejo de analisar dados e para prever resultados futuros por muito tempo e muito da ciência natural tem suas raízes nisso. Por exemplo, a distribuição Bernoulli é nomeada após [Jacob Bernoulli (1655–1705)] (https://en.wikipedia.org/wiki/Jacob_Bernoulli), e a distribuição gaussiana foi descoberta por [Carl Friedrich Gauss (1777–1855)] (https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss). Ele inventou, por exemplo, o algoritmo de mínimos quadrados médios, que ainda é usado hoje para inúmeros problemas de cálculos de seguros a diagnósticos médicos. Essas ferramentas deram origem a uma abordagem experimental nas ciências naturais — por exemplo, a lei de Ohm relacionar corrente e tensão em um resistor é perfeitamente descrito por um modelo linear.,

Mesmo na Idade Média, os matemáticos tinham uma aguçada intuição de estimativas. Por exemplo, o livro de geometria de [Jacob Köbel (1460–1533)] (https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry) ilustra calculando a média do comprimento de 16 pés de homens adultos para obter o comprimento médio do pé.

Fig. 1.4.1: Estimating the length of a foot.

numref `fig_koebel` ilustra como este estimador funciona. Os 16

homens adultos foram convidados a fazer fila em uma fila, ao deixar a igreja. Seu comprimento agregado foi então dividido por 16 para obter uma estimativa do que agora equivale a 1 pé. Este "algoritmo" foi melhorado mais tarde para lidar com pés deformados — os 2 homens com os pés mais curtos e os mais longos, respectivamente, foram mandados embora, calculando a média apenas sobre o restante. Este é um dos primeiros exemplos da estimativa média aparada.

As estatísticas realmente decolaram com a coleta e disponibilização de dados. Um de seus titãs, [Ronald Fisher (1890–1962)] (https://en.wikipedia.org/wiki/Ronald_Fisher), contribuiu significativamente para sua teoria e também suas aplicações em genética. Muitos de seus algoritmos (como a análise discriminante linear) e fórmula (como a matriz de informações de Fisher) ainda estão em uso frequente hoje. Na verdade, até mesmo o conjunto de dados Iris que Fisher lançou em 1936 ainda é usado às vezes para ilustrar algoritmos de aprendizado de máquina. Ele também era um defensor da eugenia, o que deve nos lembrar que o uso moralmente duvidoso da ciência de dados tem uma história tão longa e duradoura quanto seu uso produtivo na indústria e nas ciências naturais.

Uma segunda influência para o *machine learning* veio da teoria da informação por [Claude Shannon (1916–2001)] (https://en.wikipedia.org/wiki/Claude_Shannon) e a teoria da computação via [Alan Turing (1912–1954)] (https://en.wikipedia. org / wiki / Alan_Turing). Turing colocou a questão "podem as máquinas pensar?" em seu famoso artigo *Computing Machinery and Intelligence* :cite: `Turing.1950`. No que ele descreveu como o teste de Turing, uma máquina pode ser considerada *inteligente* se for difícil para um avaliador humano distinguir entre as respostas de uma máquina e um humano com base em interações textuais.

Outra influência pode ser encontrada na neurociência e na psicologia. Afinal, os humanos exibem claramente um comportamento inteligente. Portanto, é razoável perguntar se alguém

poderia explicar e possivelmente fazer engenharia reversa dessa capacidade. Um dos algoritmos mais antigos inspirados nesta moda foi formulado por [Donald Hebb (1904–1985)] (https://en.wikipedia.org/wiki/Donald_O._Hebb). Em seu livro inovador *The Organization of Behavior* :cite: `Hebb.Hebb.1949`, ele postulou que os neurônios aprendem por reforço positivo. Isso ficou conhecido como a regra de aprendizado Hebbian. É o protótipo do algoritmo de aprendizagem perceptron de Rosenblatt e lançou as bases de muitos algoritmos de gradiente descendente estocástico que sustentam o *deep learning* hoje: reforçar o comportamento desejável e diminuir o comportamento indesejável para obter boas configurações dos parâmetros em uma rede neural.

Inspiração biológica é o que deu às *redes neurais* seu nome. Por mais de um século (que remonta aos modelos de Alexander Bain, 1873 e James Sherrington, 1890), os pesquisadores tentaram reunir circuitos computacionais que se assemelham a redes de neurônios em interação. Com o tempo, a interpretação da biologia tornou-se menos literal mas o nome pegou. Em sua essência, encontram-se alguns princípios-chave que podem ser encontrados na maioria das redes hoje:

- A alternância de unidades de processamento linear e não linear, geralmente chamadas de *camadas*.
- O uso da regra da cadeia (também conhecida como *retropropagação*) para ajustar parâmetros em toda a rede de uma só vez.

Após o rápido progresso inicial, pesquisa em redes neurais definhou de cerca de 1995 até 2005. Isso se deveu principalmente a dois motivos. Primeiro, treinar uma rede é muito caro do ponto de vista computacional. Embora a memória de acesso aleatório fosse abundante no final do século passado, o poder computacional era escasso. Em segundo lugar, os conjuntos de dados eram relativamente pequenos. Na verdade, o conjunto de dados Iris de Fisher de 1932 foi uma ferramenta popular para testar a eficácia de algoritmos. O conjunto de dados MNIST com seus 60.000 dígitos manuscritos foi considerado enorme.

Dada a escassez de dados e computação, ferramentas estatísticas fortes, como métodos de *kernel*, árvores de decisão e modelos gráficos mostraram-se empiricamente superiores. Ao contrário das redes neurais, eles não levaram semanas para treinar e forneceu resultados previsíveis com fortes garantias teóricas.

1.5 O Caminho Para o *Deep Learning*

Muito disso mudou com a pronta disponibilidade de grandes quantidades de dados, devido à World Wide Web, o advento das empresas servindo centenas de milhões de usuários online, uma disseminação de sensores baratos e de alta qualidade, armazenamento de dados barato (lei de Kryder), e computação barata (lei de Moore), em particular na forma de GPUs, originalmente projetadas para jogos de computador. De repente, algoritmos e modelos que pareciam inviáveis computacionalmente tornaram-se relevante (e vice-versa). Isso é melhor ilustrado em: numref:`tab_intro_decade`.

:*Dataset* vs. memória do computador e poder computacional

Table 1.5.1: label:`tab_intro_decade`

Década	Dataset	Memória	Cálculos de *Floats* por Segundo
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (Preço de casas em Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (reconhecimento óptico de caracteres)	10 MB	10 MF (Intel 80486)
2000	10 M (páginas *web*)	100 MB	1 GF (Intel Core)
2010	10 G (anúncios)	1 GB	1 TF (Nvidia C2050)
2020	1 T (redes sociais)	100 GB	1 PF (Nvidia DGX-2)

É evidente que a memória RAM não acompanhou o crescimento dos dados. Ao mesmo tempo, o aumento do poder computacional ultrapassou o dos dados disponíveis. Isso significa que os modelos estatísticos precisam se tornar mais eficientes em termos de memória (isso normalmente é obtido adicionando não linearidades) ao mesmo tempo em que pode passar mais tempo na otimização desses parâmetros, devido ao aumento do orçamento computacional. Consequentemente, o ponto ideal em *machine learning* e estatísticas mudou de modelos lineares (generalizados) e métodos de *kernel* para redes neurais profundas. Esta é também uma das razões pelas quais muitos dos pilares de *deep learning*, como *perceptrons* multicamadas (McCulloch & Pitts, 1943), redes neurais convolucionais (LeCun et al., 1998), memória longa de curto prazo (Hochreiter & Schmidhuber, 1997), e Q-Learning (Watkins & Dayan, 1992), foram essencialmente "redescobertos" na última década, depois de ficarem relativamente dormentes por um tempo considerável.

O progresso recente em modelos estatísticos, aplicativos e algoritmos às vezes é comparado à explosão cambriana: um momento de rápido progresso na evolução das espécies. Na verdade, o estado da arte não é apenas uma mera consequência de recursos disponíveis, aplicados a algoritmos de décadas. Observe que a lista abaixo mal arranha a superfície das ideias que ajudaram os pesquisadores a alcançar um progresso tremendo na última década.

- Novos métodos de controle de capacidade, como *dropout* (Srivastava et al., 2014), ajudaram a mitigar o perigo de *overfitting*. Isso foi conseguido aplicando injeção de ruído (Bishop, 1995) em toda a rede neural, substituindo pesos por variáveis aleatórias para fins de treinamento.
- Mecanismos de atenção resolveram um segundo problema que atormentou as estatísticas por mais de um século: como aumentar a memória e a complexidade de um sistema sem aumentar o número de parâmetros aprendíveis. Os pesquisadores encontraram uma solução elegante usando o que só pode ser visto como uma estrutura de ponteiro aprendível (Bahdanau et al., 2014). Em vez de ter que lembrar uma sequência de texto inteira, por exemplo, para tradução automática em uma representação de dimensão fixa, tudo o que precisava ser armazenado era um ponteiro para o estado intermediário do processo de tradução. Isso permitiu significativamente maior precisão para sequências longas, uma vez que o modelo não precisa mais lembrar de toda a sequência antes de começar a geração de uma nova sequência.
- Projetos de vários estágios, por exemplo, por meio das redes de memória (Sukhbaatar et al., 2015) e o programador-intérprete neural:cite:`Reed.De-Freitas.2015` permitiram modeladores estatísticos para descrever abordagens iterativas de raciocínio. Essas ferramentas permitem um estado interno da rede neural profunda a ser modificado repetidamente, realizando assim as etapas subsequentes em uma cadeia de raciocínio, semelhante a como um processador pode modificar a memória para um cálculo.
- Outro desenvolvimento importante foi a invenção de redes adversárias geradoras (Goodfel-

low et al., 2014). Tradicionalmente, os métodos estatísticos para estimativa de densidade e modelos generativos focados em encontrar distribuições de probabilidade adequadas e algoritmos (geralmente aproximados) para amostragem deles. Como resultado, esses algoritmos foram amplamente limitados pela falta de flexibilidade inerente aos modelos estatísticos. A inovação crucial nas redes adversárias geradoras foi substituir o amostrador por um algoritmo arbitrário com parâmetros diferenciáveis. Estes são ajustados de tal forma que o discriminador (efetivamente um teste de duas amostras) não consegue distinguir dados falsos de dados reais. Através da capacidade de usar algoritmos arbitrários para gerar dados, abriu-se a estimativa de densidade para uma ampla variedade de técnicas. Exemplos de Zebras galopando (Zhu et al., 2017) e de rostos falsos de celebridades (Karras et al., 2017) são ambos testemunhos desse progresso. Até mesmo *doodlers* amadores podem produzir imagens fotorrealísticas baseadas apenas em esboços que descrevem como o *layout* de uma cena se parece (Park et al., 2019).

- Em muitos casos, uma única GPU é insuficiente para processar a grande quantidade de dados disponíveis para treinamento. Na última década, a capacidade de construir paralelos e algoritmos de treinamento distribuído melhoraram significativamente. Um dos principais desafios no projeto de algoritmos escalonáveis é a força motriz da otimização de *deep learning*, o gradiente descendente estocástico, depende de pequenos *minibatches* de dados a serem processados. Ao mesmo tempo, pequenos *batches* limitam a eficiência das GPUs. Portanto, o treinamento em 1024 GPUs com um tamanho de *minibatch* de, digamos que 32 imagens por *batch* equivale a um *minibatch* agregado de cerca de 32.000 imagens. Trabalhos recentes, primeiro de Li (Li, 2017), e posteriormente por (You et al., 2017) e (Jia et al., 2018) aumentaram o tamanho para 64.000 observações, reduzindo o tempo de treinamento do modelo ResNet-50 no *dataset* ImageNet para menos de 7 minutos. Para comparação — inicialmente os tempos de treinamento foram medidos na ordem de dias.

- A capacidade de paralelizar a computação também contribuiu de maneira crucial para progredir na aprendizagem por reforço, pelo menos sempre que a simulação é uma opção. Isso levou a um progresso significativo na obtenção de desempenho sobre-humano em Go, jogos Atari, Starcraft e em física simulações (por exemplo, usando MuJoCo). Veja, por exemplo, (Silver et al., 2016) para uma descrição de como conseguir isso no AlphaGo. Em poucas palavras, o aprendizado por reforço funciona melhor se muitas tuplas (estado, ação, recompensa) estiverem disponíveis, ou seja, sempre que for possível experimentar muitas coisas para aprender como elas se relacionam com cada outra. A simulação fornece esse caminho.

- *Frameworks* de *deep learning* têm desempenhado um papel crucial na disseminação de ideias. A primeira geração de *frameworks* permitindo fácil modelagem englobava Caffe[8], Torch[9], e Theano[10]. Muitos artigos seminais foram escritos usando essas ferramentas. Até agora, eles foram substituídos por TensorFlow[11] (frequentemente usado por meio de sua API de alto nível [Keras] (https://github.com/keras-team/keras)), CNTK[12], Caffe 2[13] e Apache MXNet[14]. A terceira geração de ferramentas, ou seja, ferramentas imperativas para *deep learning*, foi provavelmente liderada por Chainer[15], que usava uma sintaxe semelhante a Python NumPy para descrever modelos. Esta ideia foi adotada por ambos PyTorch[16], a Gluon API[17] da

[8] https://github.com/BVLC/caffe
[9] https://github.com/torch
[10] https://github.com/Theano/Theano
[11] https://github.com/tensorflow/tensorflow
[12] https://github.com/Microsoft/CNTK
[13] https://github.com/caffe2/caffe2
[14] https://github.com/apache/incubator-mxnet
[15] https://github.com/chainer/chainer
[16] https://github.com/pytorch/pytorch
[17] https://github.com/apache/incubator-mxnet

MXNet e Jax[18].

A divisão de trabalho entre os pesquisadores do sistema construindo melhores ferramentas e modeladores estatísticos construindo melhores redes neurais simplificou muito as coisas. Por exemplo, treinar um modelo de regressão linear logística costumava ser um problema de lição de casa não trivial, digno de dar aos novos alunos de Ph.D. em *machine learning* da *Carnegie Mellon University* em 2014. Agora, esta tarefa pode ser realizada com menos de 10 linhas de código, colocando-a firmemente nas mãos dos programadores.

1.6 Histórias de Sucesso

A IA tem uma longa história de entrega de resultados que seriam difíceis de realizar de outra forma. Por exemplo, os sistemas de classificação de correio usando reconhecimento óptico de caracteres foram implantados desde a década de 1990. Afinal, essa é a fonte do famoso *dataset* MNIST de dígitos manuscritos. O mesmo se aplica à leitura de cheques para depósitos bancários e pontuação de crédito para clientes. As transações financeiras são verificadas em busca de fraudes automaticamente. Isso forma a espinha dorsal de muitos sistemas de pagamento de comércio eletrônico, como PayPal, Stripe, AliPay, WeChat, Apple, Visa e MasterCard. Os programas de computador para xadrez são competitivos há décadas. O *machine learning* alimenta pesquisa, recomendação, personalização e classificação na Internet. Em outras palavras, o *machine learning* é difundido, embora muitas vezes oculto à vista.

Só recentemente é que a IA tem estado no centro das atenções, principalmente devido a soluções para problemas que foram considerados intratáveis anteriormente e que estão diretamente relacionados aos consumidores. Muitos desses avanços são atribuídos ao *machine learning*.

- Assistentes inteligentes, como Siri da Apple, Alexa da Amazon e do Google assistente, são capazes de responder a perguntas faladas com um grau razoável de precisão. Isso inclui tarefas servis, como ligar interruptores de luz (uma bênção para os deficientes), marcar compromissos com o barbeiro e oferecer suporte por telefone. Este é provavelmente o sinal mais perceptível de que a IA está afetando nossas vidas.
- Um ingrediente importante nos assistentes digitais é a capacidade de reconhecer a fala com precisão. Gradualmente, a precisão de tais sistemas aumentou ao ponto onde eles alcançam a paridade humana com certas aplicações (Xiong et al., 2018).
- O reconhecimento de objetos também percorreu um longo caminho. Estimando o objeto em uma imagem foi uma tarefa bastante desafiadora em 2010. No *benchmark* ImageNet, pesquisadores da NEC Labs e da Universidade de Illinois em Urbana-Champaign alcançaram uma taxa de erro de 28%, entre os 5 primeiros (Lin et al., 2010). Em 2017, esta taxa de erro foi reduzida para 2,25% (Hu et al., 2018). Da mesma forma, deslumbrante resultados foram alcançados para a identificação de aves ou diagnóstico de câncer de pele.
- Os jogos costumavam ser um bastião da inteligência humana. Começando com TD-Gammon, um programa para jogar gamão usando aprendizagem de reforço de diferença temporal, progresso algorítmico e computacional levou a algoritmos para uma ampla gama de aplicações. Ao contrário do gamão, o xadrez tem um espaço de estados e um conjunto de ações muito mais complexos. *DeepBlue* venceu Garry Kasparov usando paralelismo massivo, *hardware* para fins especiais e busca eficiente na árvore do jogo (Campbell et al., 2002). Go é ainda mais difícil, devido ao seu enorme espaço de estados. AlphaGo atingiu a paridade humana em 2015, usando *deep learning* combinado com amostragem de árvore Monte

[18] https://github.com/google/jax

Carlo (Silver et al., 2016). O desafio no Poker era que o espaço de estados é grande e não é totalmente observado (não conhecemos as cartas dos oponentes). Libratus excedeu o desempenho humano no Poker usando de forma eficiente estratégias estruturadas (Brown & Sandholm, 2017). Isso ilustra o progresso impressionante nos jogos e o fato de que algoritmos avançados desempenharam um papel crucial neles.

- Outra indicação de progresso na IA é o advento dos carros autônomos e caminhões. Embora a autonomia total ainda não esteja totalmente ao nosso alcance, excelente progresso foi feito nesta direção, com empresas como Tesla, NVIDIA, e Waymo enviando produtos que permitem pelo menos uma autonomia parcial. O que torna a autonomia total tão desafiadora é que a direção adequada requer a habilidade de perceber, raciocinar e incorporar regras em um sistema. No momento, o *deep learning* é usado principalmente no aspecto de visão computacional desses problemas. O resto é intensamente ajustado por engenheiros.

Novamente, a lista acima apenas arranha a superfície de onde o *machine learning* afetou os aplicativos práticos. Por exemplo, robótica, logística, biologia computacional, física de partículas e astronomia devem alguns de seus avanços recentes mais impressionantes, pelo menos em partes, ao *machine learning*. O *machine learning* está se tornando uma ferramenta onipresente para engenheiros e cientistas.

Freqüentemente, a questão do apocalipse da IA, ou a singularidade da IA foi levantado em artigos não técnicos sobre IA. O medo é que, de alguma forma, os sistemas de *machine learning* se tornarão sensíveis e decidirão independentemente de seus programadores (e mestres) sobre coisas que afetam diretamente o sustento dos humanos. Até certo ponto, a IA já afeta a vida dos humanos de forma imediata: a qualidade de crédito é avaliada automaticamente, os pilotos automáticos navegam veículos sozinhos, decisões sobre se deve conceder fiança, usam dados estatísticos como entrada. Mais levianamente, podemos pedir a Alexa que ligue a máquina de café.

Felizmente, estamos longe de um sistema inteligente de IA que esteja pronto para manipular seus criadores humanos (ou queimar seu café). Em primeiro lugar, os sistemas de IA são projetados, treinados e implantados de uma forma específica, maneira orientada para o objetivo. Embora seu comportamento possa dar a ilusão de inteligência geral, é uma combinação de regras, heurísticas e modelos estatísticos que fundamentam o design. Em segundo lugar, dentre as ferramentas para *inteligência artificial geral* atuais, simplesmente não existem aquelas que são capazes de se melhorar, raciocinar sobre si mesmos, e que são capazes de modificar, estender e melhorar sua própria arquitetura ao tentar resolver tarefas gerais.

Uma preocupação muito mais urgente é como a IA está sendo usada em nossas vidas diárias. É provável que muitas tarefas servis realizadas por motoristas de caminhão e os assistentes de loja podem e serão automatizados. Robôs agrícolas provavelmente reduzirão o custo da agricultura orgânica e também automatizarão as operações de colheita. Esta fase da revolução industrial pode ter consequências profundas em grandes segmentos da sociedade, já que motoristas de caminhão e assistentes de loja são alguns dos empregos mais comuns em muitos países. Além disso, os modelos estatísticos, quando aplicados sem cuidado pode levar a preconceitos raciais, de gênero ou idade e aumentar preocupações razoáveis sobre justiça processual se automatizados para conduzir decisões consequentes. É importante garantir que esses algoritmos sejam usados com cuidado. Com o que sabemos hoje, isso nos causa uma preocupação muito mais premente do que o potencial da superinteligência malévola para destruir a humanidade.

1.7 Características

Até agora, falamos sobre *machine learning* de maneira ampla, que é tanto um ramo da IA quanto uma abordagem da IA. Embora o *deep learning* seja um subconjunto do *machine learning*, o conjunto estonteante de algoritmos e aplicativos torna difícil avaliar quais podem ser os ingredientes específicos para o *deep learning*. Isso é tão difícil quanto tentar definir os ingredientes necessários para a pizza, uma vez que quase todos os componentes são substituíveis.

Como descrevemos, o *machine learning* pode usar dados para aprender as transformações entre entradas e saídas, como a transformação de áudio em texto no reconhecimento de fala. Ao fazer isso, muitas vezes é necessário representar os dados de uma forma adequada para que os algoritmos transformem essas representações na saída. *Deep learning* é *profundo* exatamente no sentido que seus modelos aprendem muitas *camadas* de transformações, onde cada camada oferece a representação em um nível. Por exemplo, camadas perto da entrada podem representar detalhes de baixo nível dos dados, enquanto as camadas mais próximas da saída de classificação pode representar conceitos mais abstratos usados para discriminação. Uma vez que *aprendizagem de representação* visa encontrar a própria representação, o *deep learning* pode ser referido como aprendizagem de representação multinível.

Os problemas que discutimos até agora, como aprendizagem do sinal de áudio bruto, os valores de pixel brutos das imagens, ou mapeamento entre sentenças de comprimentos arbitrários e suas contrapartes em línguas estrangeiras, são aqueles onde o *deep learning* se destaca e onde os métodos tradicionais de *machine learning* vacilam. Acontece que esses modelos de várias camadas são capazes de abordar dados perceptivos de baixo nível de uma forma que as ferramentas anteriores não conseguiam. Indiscutivelmente, a semelhança mais significativa em métodos de *deep learning* é o uso de *treinamento ponta a ponta*. Ou seja, ao invés de montar um sistema baseado em componentes que são ajustados individualmente, constrói-se o sistema e então ajusta seu desempenho em conjunto. Por exemplo, em visão computacional, os cientistas costumavam separar o processo de *engenharia de recursos* do processo de construção de modelos de aprendizado de máquina. O detector de bordas Canny (Canny, 1987) e o extrator de recursos SIFT de Lowe (Lowe, 2004) reinou supremo por mais de uma década como algoritmo para mapear imagens em vetores de recursos. No passado, a parte crucial de aplicar o *machine learning* a esses problemas consistia em criar métodos de engenharia manual para transformar os dados em alguma forma receptiva a modelos superficiais. Infelizmente, há muito pouco que os humanos possam realizar com engenhosidade em comparação com uma avaliação consistente de milhões de escolhas realizadas automaticamente por um algoritmo. Quando o *deep learning* surgiu, esses extratores de recursos foram substituídos por filtros ajustados automaticamente, produzindo uma precisão superior.

Por isso, uma das principais vantagens do *deep learning* é que ele não substitui apenas os modelos rasos no final dos canais de aprendizagem tradicionais, mas também o processo intensivo de mão-de-obra de engenharia de recursos. Além disso, ao substituir grande parte do pré-processamento específico do domínio, o *deep learning* eliminou muitos dos limites que antes separavam a visão computacional, o reconhecimento de fala, processamento de linguagem natural, informática médica e outras áreas de aplicação, oferecendo um conjunto unificado de ferramentas para lidar com diversos problemas.

Além do treinamento de ponta a ponta, estamos experimentando uma transição de descrições estatísticas paramétricas para modelos totalmente não paramétricos. Quando os dados são escassos, é necessário confiar na simplificação de suposições sobre a realidade para obter modelos úteis. Quando os dados são abundantes, eles podem ser substituídos por modelos não paramétricos que se ajustam à realidade com mais precisão. Até certo ponto, isso reflete o progresso que a física experimentou em meados do século anterior com a disponibilidade de computadores. Em

vez de resolver aproximações paramétricas de como os elétrons se comportam manualmente, pode-se agora recorrer a simulações numéricas das equações diferenciais parciais associadas. Isso levou a modelos muito mais precisos, embora muitas vezes às custas da explicabilidade.

Outra diferença em relação ao trabalho anterior é a aceitação de soluções sub-ótimas, lidar com problemas de otimização não-linear não convexa e a disposição de tentar coisas antes de prová-las. Esse empirismo recém-descoberto ao lidar com problemas estatísticos, combinado com um rápido influxo de talentos, levou a um rápido progresso de algoritmos práticos, embora em muitos casos às custas de modificar e reinventar ferramentas que existiram por décadas.

No final, a comunidade de aprendizagem profunda se orgulha de compartilhar ferramentas além das fronteiras acadêmicas e corporativas, lançando muitas bibliotecas excelentes, modelos estatísticos e redes treinadas como código aberto. É com esse espírito que os *notebooks* que compõem este livro estão disponíveis gratuitamente para distribuição e uso. Trabalhamos muito para reduzir as barreiras de acesso para que todos aprendam sobre o *deep learning* e esperamos que nossos leitores se beneficiem com isso.

1.8 Resumo

- O *machine learning* estuda como os sistemas de computador podem aproveitar a experiência (geralmente dados) para melhorar o desempenho em tarefas específicas. Ele combina ideias de estatísticas, mineração de dados e otimização. Frequentemente, é usado como meio de implementação de soluções de IA.
- Como uma classe de *machine learning*, o aprendizado representacional se concentra em como encontrar automaticamente a maneira apropriada de representar os dados. *Deep learning* é a aprendizagem de representação em vários níveis através do aprendizado de muitas camadas de transformações.
- O *deep learning* substitui não apenas os modelos superficiais no final dos canais de *machine learning* tradicionais, mas também o processo trabalhoso de engenharia de recursos.
- Muito do progresso recente no *deep learning* foi desencadeado por uma abundância de dados provenientes de sensores baratos e aplicativos em escala da Internet e por um progresso significativo na computação, principalmente por meio de GPUs.
- A otimização de todo o sistema é um componente chave para a obtenção de alto desempenho. A disponibilidade de estruturas eficientes de *deep learning* tornou o projeto e a implementação disso significativamente mais fáceis.

1.9 Exercícios

1. Quais partes do código que você está escrevendo atualmente podem ser "aprendidas", ou seja, aprimoradas ao aprender e determinando automaticamente as opções de design feitas em seu código? Seu código inclui opções de design heurístico?
2. Quais problemas você encontra têm muitos exemplos de como resolvê-los, mas nenhuma maneira específica de automatizá-los? Esses podem ser os principais candidatos para usar o *deep learning*.

3. Vendo o desenvolvimento da IA como uma nova revolução industrial, qual é a relação entre algoritmos e dados? É semelhante a motores a vapor e carvão? Qual é a diferença fundamental?
4. Onde mais você pode aplicar a abordagem de treinamento de ponta a ponta, como em Fig. 1.1.2, física, engenharia e econometria?

Discussions[19]

[19] https://discuss.d2l.ai/t/22

2 | Preliminares

Para iniciarmos o nosso aprendizado de *Deep Learning*, precisaremos desenvolver algumas habilidades básicas. Todo aprendizado de máquina está relacionado com a extração de informações dos dados. Portanto, começaremos aprendendo as habilidades práticas para armazenar, manipular e pré-processar dados.

Além disso, o aprendizado de máquina normalmente requer trabalhar com grandes conjuntos de dados, que podemos considerar como tabelas, onde as linhas correspondem a exemplos e as colunas correspondem aos atributos. A álgebra linear nos dá um poderoso conjunto de técnicas para trabalhar com dados tabulares. Não iremos muito longe nas teoria, mas sim nos concentraremos no básico das operações matriciais e sua implementação.

Além disso, o *Deep Learning* tem tudo a ver com otimização. Temos um modelo com alguns parâmetros e queremos encontrar aqueles que melhor se ajustam aos nossos dados. Determinar como alterar cada parâmetro em cada etapa de um algoritmo requer um pouco de cálculo, que será brevemente apresentado. Felizmente, o pacote `autograd` calcula automaticamente a diferenciação para nós, e vamos cobrir isso a seguir.

Em seguida, o aprendizado de máquina se preocupa em fazer previsões: qual é o valor provável de algum atributo desconhecido, dada a informação que observamos? Raciocinar rigorosamente sob a incerteza precisaremos invocar a linguagem da probabilidade.

No final, a documentação oficial fornece muitas descrições e exemplos que vão além deste livro. Para concluir o capítulo, mostraremos como procurar documentação para as informações necessárias.

Este livro manteve o conteúdo matemático no mínimo necessário para obter uma compreensão adequada de *Deep Learning* No entanto, isso não significa que este livro é livre de matemática. Assim, este capítulo fornece uma introdução rápida a matemática básica e frequentemente usada para permitir que qualquer pessoa entenda pelo menos *a maior parte* do conteúdo matemático do livro. Se você deseja entender *todo* o conteúdo matemático, uma revisão adicional do apêndice online sobre matemática[20] deve ser suficiente.

[20] https://d2l.ai/chapter_apencha-mathematics-for-deep-learning/index.html

2.1 Manipulação de Dados

Para fazer qualquer coisa, precisamos de alguma forma de armazenar e manipular dados. Geralmente, há duas coisas importantes que precisamos fazer com os dados: (i) adquirir eles; e (ii) processá-los assim que estiverem dentro do computador. Não há sentido em adquirir dados sem alguma forma de armazená-los, então vamos brincar com dados sintéticos. Para começar, apresentamos o *array* n-dimensional, também chamado de *tensor*.

Se você trabalhou com NumPy, o mais amplamente utilizado pacote de computação científica em Python, então você achará esta seção familiar. Não importa qual estrutura você usa, sua *classe de tensor* (`ndarray` em MXNet, `Tensor` em PyTorch e TensorFlow) é semelhante ao`ndarray` do NumPy com alguns recursos interessantes. Primeiro, a GPU é bem suportada para acelerar a computação enquanto o NumPy suporta apenas computação de CPU. Em segundo lugar, a classe tensor suporta diferenciação automática. Essas propriedades tornam a classe tensor adequada para aprendizado profundo. Ao longo do livro, quando dizemos tensores, estamos nos referindo a instâncias da classe tensorial, a menos que seja declarado de outra forma.

2.1.1 Iniciando

Nesta seção, nosso objetivo é colocá-lo em funcionamento, equipando você com as ferramentas básicas de matemática e computação numérica que você desenvolverá conforme progride no livro. Não se preocupe se você lutar para grocar alguns dos os conceitos matemáticos ou funções de biblioteca. As seções a seguir revisitarão este material no contexto de exemplos práticos e irá afundar. Por outro lado, se você já tem alguma experiência e quiser se aprofundar no conteúdo matemático, basta pular esta seção.

Para começar, importamos `torch`. Note que apesar de ser chamado PyTorch, devemos importar `torch` ao invés de `pytorch`.

```
import torch
```

Um tensor representa uma matriz (possivelmente multidimensional) de valores numéricos. Com uma dimensão, um tensor corresponde (em matemática) a um *vetor*. Com duas dimensões, um tensor corresponde a uma * matriz *. Tensores com mais de dois eixos não possuem nomes matemáticos.

Para começar, podemos usar `arange` para criar um vetor linha `x` contendo os primeiros 12 inteiros começando com 0, embora eles sejam criados como *float* por padrão. Cada um dos valores em um tensor é chamado de *elemento* do tensor. Por exemplo, existem 12 elementos no tensor `x`. A menos que especificado de outra forma, um novo tensor será armazenado na memória principal e designado para computação baseada em CPU.

```
x = torch.arange(12)
x
```

```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Podemos acessar o formato do tensor (o comprimento em cada coordenada) inspecionando sua propriedade `shape` .

```
x.shape
```

```
torch.Size([12])
```

Se quisermos apenas saber o número total de elementos em um tensor, ou seja, o produto de todos os *shapes*, podemos inspecionar seu tamanho. Porque estamos lidando com um vetor aqui, o único elemento de seu `shape` é idêntico ao seu tamanho.

```
x.numel()
```

```
12
```

Para mudar o *shape* de um tensor sem alterar o número de elementos ou seus valores, podemos invocar a função `reshape`. Por exemplo, podemos transformar nosso tensor, `x`, de um vetor linha com forma (12,) para uma matriz com forma (3, 4). Este novo tensor contém exatamente os mesmos valores, mas os vê como uma matriz organizada em 3 linhas e 4 colunas. Para reiterar, embora a forma tenha mudado, os elementos não. Observe que o tamanho não é alterado pela remodelagem.

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

A remodelação especificando manualmente todas as dimensões é desnecessária. Se nossa forma de destino for uma matriz com forma (altura, largura), então, depois de sabermos a largura, a altura é dada implicitamente. Por que devemos realizar a divisão nós mesmos? No exemplo acima, para obter uma matriz com 3 linhas, especificamos que deve ter 3 linhas e 4 colunas. Felizmente, os tensores podem calcular automaticamente uma dimensão considerando o resto. Invocamos esse recurso colocando `-1` para a dimensão que gostaríamos que os tensores inferissem automaticamente. No nosso caso, em vez de chamar `x.reshape (3, 4)`, poderíamos ter chamado equivalentemente `x.reshape (-1, 4)` ou `x.reshape (3, -1)`.

Normalmente, queremos que nossas matrizes sejam inicializadas seja com zeros, uns, algumas outras constantes, ou números amostrados aleatoriamente de uma distribuição específica. Podemos criar um tensor representando um tensor com todos os elementos definido como 0 e uma forma de (2, 3, 4) como a seguir:

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Da mesma forma, podemos criar tensores com cada elemento definido como 1 da seguinte maneira:

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

Frequentemente, queremos amostrar aleatoriamente os valores para cada elemento em um tensor de alguma distribuição de probabilidade. Por exemplo, quando construímos matrizes para servir como parâmetros em uma rede neural, vamos normalmente inicializar seus valores aleatoriamente. O fragmento a seguir cria um tensor com forma (3, 4). Cada um de seus elementos é amostrado aleatoriamente de uma distribuição gaussiana (normal) padrão com uma média de 0 e um desvio padrão de 1.

```
torch.randn(3, 4)
```

```
tensor([[-1.0383,  2.7221,  1.6101,  0.3270],
        [ 1.2290,  0.3447, -0.8467, -1.8943],
        [ 0.7013, -1.5338, -0.2593, -0.6438]])
```

Podemos também especificar os valores exatos para cada elemento no tensor desejado fornecendo uma lista Python (ou lista de listas) contendo os valores numéricos. Aqui, a lista externa corresponde ao eixo 0 e a lista interna ao eixo 1.

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

2.1.2 Operações

Este livro não é sobre engenharia de software. Nossos interesses não se limitam a simplesmente leitura e gravação de dados de/para matrizes. Queremos realizar operações matemáticas nessas matrizes. Algumas das operações mais simples e úteis são as operações elemento a elemento. Estes aplicam uma operação escalar padrão para cada elemento de uma matriz. Para funções que usam dois arrays como entradas, as operações elemento a elemento aplicam algum operador binário padrão em cada par de elementos correspondentes das duas matrizes. Podemos criar uma função elemento a elemento a partir de qualquer função que mapeia de um escalar para um escalar.

Em notação matemática, denotaríamos tal um operador escalar *unário* (tomando uma entrada) pela assinatura $f : \mathbb{R} \rightarrow \mathbb{R}$. Isso significa apenas que a função está mapeando de qualquer número real ($\mathbb{R}$) para outro. Da mesma forma, denotamos um operador escalar *binário* (pegando duas

entradas reais e produzindo uma saída) pela assinatura $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Dados quaisquer dois vetores $\mathbf{u}$ e $\mathbf{v}$ de mesmo *shape*, e um operador binário f, podemos produzir um vetor $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ definindo $c_i \leftarrow f(u_i, v_i)$ para todos i, onde c_i, u_i e v_i são os elementos i^{th} dos vetores $\mathbf{c}, \mathbf{u}$, e $\mathbf{v}$. Aqui, nós produzimos o valor vetorial $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ *transformando* a função escalar para uma operação de vetor elemento a elemento.

Os operadores aritméticos padrão comuns (+, -,*,/e**) foram todos transformados em operações elemento a elemento para quaisquer tensores de formato idêntico de forma arbitrária. Podemos chamar operações elemento a elemento em quaisquer dois tensores da mesma forma. No exemplo a seguir, usamos vírgulas para formular uma tupla de 5 elementos, onde cada elemento é o resultado de uma operação elemento a elemento.

Operações

Os operadores aritméticos padrão comuns (+, -,*,/e**) foram todos transformados em operações elemento a elemento.

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y  #  O ** é o operador exponenciação
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

Muitos mais operações podem ser aplicadas elemento a elemento, incluindo operadores unários como exponenciação.

```
torch.exp(x)
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

Além de cálculos elemento a elemento, também podemos realizar operações de álgebra linear, incluindo produtos escalar de vetor e multiplicação de matrizes. Explicaremos as partes cruciais da álgebra linear (sem nenhum conhecimento prévio assumido) em Section 2.3.

Também podemos *concatenar* vários tensores juntos, empilhando-os ponta a ponta para formar um tensor maior. Só precisamos fornecer uma lista de tensores e informar ao sistema ao longo de qual eixo concatenar. O exemplo abaixo mostra o que acontece quando concatenamos duas matrizes ao longo das linhas (eixo 0, o primeiro elemento da forma) vs. colunas (eixo 1, o segundo elemento da forma). Podemos ver que o comprimento do eixo 0 do primeiro tensor de saída (6) é a soma dos comprimentos do eixo 0 dos dois tensores de entrada ($3+3$); enquanto o comprimento do eixo 1 do segundo tensor de saída (8) é a soma dos comprimentos do eixo 1 dos dois tensores de entrada ($4+4$).

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

Às vezes, queremos construir um tensor binário por meio de *declarações lógicas*. Tome `X == Y` como exemplo. Para cada posição, se `X` e`Y` forem iguais nessa posição, a entrada correspondente no novo tensor assume o valor 1, o que significa que a declaração lógica `X == Y` é verdadeira nessa posição; caso contrário, essa posição assume 0.

```
X == Y
```

```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

Somando todos os elementos no tensor resulta em um tensor com apenas um elemento.

```
X.sum()
```

```
tensor(66.)
```

2.1.3 Mecanismo de *Broadcasting*

Na seção acima, vimos como realizar operações elemento a elemento em dois tensores da mesma forma. Sob certas condições, mesmo quando as formas são diferentes, ainda podemos realizar operações elementar invocando o mecanismo de *Broadcasting*. Esse mecanismo funciona da seguinte maneira: Primeiro, expanda um ou ambos os arrays copiando elementos de forma adequada de modo que após esta transformação, os dois tensores têm a mesma forma. Em segundo lugar, execute as operações elemento a elemento nas matrizes resultantes.

Na maioria dos casos, nós transmitimos ao longo de um eixo onde uma matriz inicialmente tem apenas o comprimento 1, como no exemplo a seguir:

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

Uma vez que `a` e`b` são matrizes 3×1 e 1×2 respectivamente, suas formas não correspondem se quisermos adicioná-los. Nós transmitimos as entradas de ambas as matrizes em uma matriz 3×2

maior da seguinte maneira: para a matriz a ele replica as colunas e para a matriz b ele replica as linhas antes de adicionar ambos os elementos.

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

2.1.4 Indexação e Fatiamento

Assim como em qualquer outro array Python, os elementos em um tensor podem ser acessados por índice. Como em qualquer matriz Python, o primeiro elemento tem índice 0 e os intervalos são especificados para incluir o primeiro, mas *antes* do último elemento. Como nas listas padrão do Python, podemos acessar os elementos de acordo com sua posição relativa ao final da lista usando índices negativos.

Assim, [-1] seleciona o último elemento e [1: 3] seleciona o segundo e o terceiro elementos da seguinte forma:

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

Além da leitura, também podemos escrever elementos de uma matriz especificando índices.

```
X[1, 2] = 9
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  9.,  7.],
        [ 8.,  9., 10., 11.]])
```

Se quisermos para atribuir a vários elementos o mesmo valor, simplesmente indexamos todos eles e, em seguida, atribuímos o valor a eles. Por exemplo, [0: 2,:] acessa a primeira e a segunda linhas, onde : leva todos os elementos ao longo do eixo 1 (coluna). Enquanto discutimos a indexação de matrizes, isso obviamente também funciona para vetores e para tensores de mais de 2 dimensões.

```
X[0:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

2.1.5 Economizando memória

As operações em execução podem fazer com que uma nova memória seja alocado aos resultados do host. Por exemplo, se escrevermos `Y = X + Y`, vamos desreferenciar o tensor que `Y` costumava apontar para e, em vez disso, aponte `Y` para a memória recém-alocada. No exemplo a seguir, demonstramos isso com a função `id ()` do Python, que nos dá o endereço exato do objeto referenciado na memória. Depois de executar `Y = Y + X`, descobriremos que`id (Y)`aponta para um local diferente. Isso ocorre porque o Python primeiro avalia `Y + X`, alocar nova memória para o resultado e, em seguida, torna `Y` aponte para este novo local na memória.

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

Isso pode ser indesejável por dois motivos. Em primeiro lugar, não queremos alocar memória desnecessariamente o tempo todo. No aprendizado de máquina, podemos ter centenas de megabytes de parâmetros e atualizar todos eles várias vezes por segundo. Normalmente, queremos realizar essas atualizações no local. Em segundo lugar, podemos apontar os mesmos parâmetros de várias variáveis. Se não atualizarmos no local, outras referências ainda apontarão para a localização da memória antiga, tornando possível para partes do nosso código para referenciar inadvertidamente parâmetros obsoletos.

Felizmente, executar operações no local é fácil. Podemos atribuir o resultado de uma operação para uma matriz previamente alocada com notação de fatia, por exemplo, `Y [:] = <expressão>`. Para ilustrar este conceito, primeiro criamos uma nova matriz `Z` com a mesma forma de outro `Y`, usando `zeros_like` para alocar um bloco de 0 entradas. : end_tab:

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140194801216832
id(Z): 140194801216832
```

Se o valor de `X` não for reutilizado em cálculos subsequentes, também podemos usar `X [:] = X + Y` ou`X + = Y` para reduzir a sobrecarga de memória da operação.

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

2.1.6 Conversão para outros objetos Python

Converter para um tensor NumPy, ou vice-versa, é fácil. O resultado convertido não compartilha memória. Este pequeno inconveniente é muito importante: quando você executa operações na CPU ou GPUs, você não quer interromper a computação, esperando para ver se o pacote NumPy do Python deseja fazer outra coisa com o mesmo pedaço de memória.

```
A = X.numpy()
B = torch.tensor(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

Para converter um tensor de tamanho 1 em um escalar Python, podemos invocar a função `item` ou as funções integradas do Python.

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

```
(tensor([3.5000]), 3.5, 3.5, 3)
```

2.1.7 Sumário

- A principal interface para armazenar e manipular dados para *Deep Learning* é o tensor (array n -dimensional). Ele fornece uma variedade de funcionalidades, incluindo operações matemáticas básicas, transmissão, indexação, divisão, economia de memória e conversão para outros objetos Python.

2.1.8 Exercícios

1. Execute o código nesta seção. Altere a declaração condicional `X == Y` nesta seção para`X < Y` ou `X > Y`, e então veja que tipo de tensor você pode obter.
2. Substitua os dois tensores que operam por elemento no mecanismo de transmissão por outras formas, por exemplo, tensores tridimensionais. O resultado é o mesmo que o esperado?

Discussions[21]

2.2 Pré-processamento de Dados

Até agora, introduzimos uma variedade de técnicas para manipular dados que já estão armazenados em tensores. Para aplicar o *Deep Learning* na solução de problemas do mundo real, frequentemente começamos com o pré-processamento de dados brutos, em vez daqueles dados bem preparados no formato tensor. Entre as ferramentas analíticas de dados populares em Python, o pacote `pandas` é comumente usado. Como muitos outros pacotes de extensão no vasto ecossistema

[21] https://discuss.d2l.ai/t/27

do Python, pandas podem trabalhar em conjunto com tensores. Então, vamos percorrer brevemente as etapas de pré-processamento de dados brutos com pandas e convertendo-os no formato tensor. Abordaremos mais técnicas de pré-processamento de dados em capítulos posteriores.

2.2.1 Lendo o *Dataset*

Como um exemplo, começamos criando um conjunto de dados artificial que é armazenado em um arquivo csv (valores separados por vírgula) `../ data / house_tiny.csv`. Dados armazenados em outro formatos podem ser processados de maneiras semelhantes.

Abaixo, escrevemos o conjunto de dados linha por linha em um arquivo csv.

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # Column names
    f.write('NA,Pave,127500\n')  # Each row represents a data example
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

Para carregar o conjunto de dados bruto do arquivo csv criado, importamos o pacote pandas e chamamos a funçãoread_csv. Este conjunto de dados tem quatro linhas e três colunas, onde cada linha descreve o número de quartos ("NumRooms"), o tipo de beco ("Alley") e o preço ("Price") de uma casa.

```
# Se o pandas ainda não estiver instalado descomente a linha abaixo:
# !pip install pandas
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2 Lidando com Dados Faltantes

Observe que as entradas "NaN" têm valores ausentes. Para lidar com dados perdidos, os métodos típicos incluem *imputação* e *exclusão*, onde a imputação substitui os valores ausentes por outros substituídos, enquanto a exclusão ignora os valores ausentes. Aqui, consideraremos a imputação.

Por indexação baseada em localização de inteiros (`iloc`), dividimos os dados em `entradas` e `saídas`, onde o primeiro leva as duas primeiras colunas, enquanto o último mantém apenas a última coluna. Para valores numéricos em `entradas` que estão faltando, nós substituímos as entradas "NaN" pelo valor médio da mesma coluna.

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN
```

Para valores categóricos ou discretos em `entradas`, consideramos "NaN" como uma categoria. Como a coluna "Alley" aceita apenas dois tipos de valores categóricos "Pave" e "NaN", O `pandas` pode converter automaticamente esta coluna em duas colunas "Alley_Pave" e "Alley_nan". Uma linha cujo tipo de beco é "Pave" definirá os valores de "Alley_Pave" e "Alley_nan" como 1 e 0. Uma linha com um tipo de beco ausente definirá seus valores para 0 e 1

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

2.2.3 Convertendo para o Formato Tensor

Agora que todas as entradas em `entradas` e `saídas` são numéricas, elas podem ser convertidas para o formato tensor. Uma vez que os dados estão neste formato, eles podem ser manipulados posteriormente com as funcionalidades de tensor que introduzimos em Section 2.1.

```
import torch

X, y = torch.tensor(inputs.values), torch.tensor(outputs.values)
X, y
```

```
(tensor([[3., 1., 0.],
         [2., 0., 1.],
         [4., 0., 1.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500, 106000, 178100, 140000]))
```

2.2.4 Sumário

- Como muitos outros pacotes de extensão no vasto ecossistema do Python, `pandas` pode trabalhar junto com tensores.
- Imputação e exclusão podem ser usadas para lidar com dados perdidos.

2.2.5 Exercícios

Crie um conjunto de dados bruto com mais linhas e colunas.

3. Exclua a coluna com a maioria dos valores ausentes.
4. Converta o conjunto de dados pré-processado para o formato tensor.

Discussions[22]

2.3 Álgebra Linear

Agora que você pode armazenar e manipular dados, vamos revisar brevemente o subconjunto da álgebra linear básica que você precisa para entender e implementar a maioria dos modelos cobertos neste livro. Abaixo, apresentamos os objetos matemáticos básicos, aritméticos, e operações em álgebra linear, expressar cada um deles por meio de notação matemática e a implementação correspondente em código.

2.3.1 Escalares

Se você nunca estudou álgebra linear ou aprendizado de máquina, então sua experiência anterior com matemática provavelmente consistia de pensar em um número de cada vez. E, se você já equilibrou um talão de cheques ou até mesmo pagou por um jantar em um restaurante então você já sabe como fazer coisas básicas como adicionar e multiplicar pares de números. Por exemplo, a temperatura em Palo Alto é de 52 graus Fahrenheit.

Formalmente, chamamos de valores que consistem de apenas uma quantidade numérica *escalar*. Se você quiser converter este valor para Celsius (escala de temperatura mais sensível do sistema métrico), você avaliaria a expressão $c = \frac{5}{9}(f - 32)$, definindo f para 52. Nesta equação, cada um dos termos—5, 9, e 32—são valores escalares. Os marcadores c e f são chamados de *variáveis* e eles representam valores escalares desconhecidos.

Neste livro, adotamos a notação matemática onde as variáveis escalares são denotadas por letras minúsculas comuns (por exemplo, x, y, and z). Denotamos o espaço de todos os escalares (contínuos) *com valor real* por $\mathbb{R}$. Por conveniência, vamos lançar mão de definições rigorosas do que exatamente é *espaço*, mas lembre-se por enquanto que a expressão $x \in \mathbb{R}$ é uma maneira formal de dizer que x é um escalar com valor real. O símbolo $\in$ pode ser pronunciado "em" e simplesmente denota associação em um conjunto. Analogamente, poderíamos escrever $x, y \in \{0, 1\}$ para afirmar que x e y são números cujo valor só pode ser 0 ou 1.

Um escalar é representado por um tensor com apenas um elemento. No próximo trecho de código, instanciamos dois escalares e realizar algumas operações aritméticas familiares com eles, a saber, adição, multiplicação, divisão e exponenciação.

[22] https://discuss.d2l.ai/t/29

```
import torch

x = torch.tensor([3.0])
y = torch.tensor([2.0])

x + y, x * y, x / y, x**y
```

```
(tensor([5.]), tensor([6.]), tensor([1.5000]), tensor([9.]))
```

2.3.2 Vetores

Você pode pensar em um vetor simplesmente como uma lista de valores escalares. Chamamos esses valores de *elementos* (*entradas* ou *componentes*) do vetor. Quando nossos vetores representam exemplos de nosso conjunto de dados, seus valores têm algum significado no mundo real. Por exemplo, se estivéssemos treinando um modelo para prever o risco de inadimplência de um empréstimo, podemos associar cada candidato a um vetor cujos componentes correspondem à sua receita, tempo de emprego, número de inadimplências anteriores e outros fatores. Se estivéssemos estudando o risco de ataques cardíacos que os pacientes de hospitais potencialmente enfrentam, podemos representar cada paciente por um vetor cujos componentes capturam seus sinais vitais mais recentes, níveis de colesterol, minutos de exercício por dia, etc. Em notação matemática, geralmente denotamos os vetores em negrito, letras minúsculas (por exemplo,, $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).

Trabalhamos com vetores via tensores unidimensionais. Em geral, os tensores podem ter comprimentos arbitrários, sujeito aos limites de memória de sua máquina.

```
x = torch.arange(4)
x
```

```
tensor([0, 1, 2, 3])
```

Podemos nos referir a qualquer elemento de um vetor usando um subscrito. Por exemplo, podemos nos referir ao elemento i^th element of $\mathbf{x}$ por x_i. Observe que o elemento x_i é um escalar, portanto, não colocamos a fonte em negrito quando nos referimos a ela. A literatura extensa considera os vetores de coluna como o padrão orientação de vetores, este livro também. Em matemática, um vetor $\mathbf{x}$ pode ser escrito como

$$\mathbf{x} =\begin{bmatrix}x_{1} \\x_{2} \\ \vdots \\x_{n}\end{bmatrix}, \tag{2.3.1}$$

onde $x_1, \ldots, x_n$ são elementos do vetor. No código acessamos qualquer elemento indexando no tensor

```
x[3]
```

```
tensor(3)
```

Comprimento, Dimensionalidade e Forma

Vamos revisitar alguns conceitos de: numref: `sec_ndarray`. Um vetor é apenas uma matriz de números. E assim como todo array tem um comprimento, todo vetor também. Em notação matemática, se quisermos dizer que um vetor $\mathbf{x}$ consiste em n escalares com valor real, podemos expressar isso como $\mathbf{x} \in \mathbb{R}^n$. O comprimento de um vetor é comumente chamado de *dimensão* do vetor.

Tal como acontece com uma matriz Python comum, nós podemos acessar o comprimento de um tensor chamando a função `len ()` embutida do Python.

```
len(x)
```

```
4
```

Quando um tensor representa um vetor (com precisamente um eixo), também podemos acessar seu comprimento por meio do atributo `.shape`. A forma é uma tupla que lista o comprimento (dimensionalidade) ao longo de cada eixo do tensor. Para tensores com apenas um eixo, a forma tem apenas um elemento.

```
x.shape
```

```
torch.Size([4])
```

Observe que a palavra "dimensão" tende a ficar sobrecarregada nesses contextos e isso tende a confundir as pessoas. Para esclarecer, usamos a dimensionalidade de um *vetor* ou um *eixo* para se referir ao seu comprimento, ou seja, o número de elementos de um vetor ou eixo. No entanto, usamos a dimensionalidade de um tensor para se referir ao número de eixos que um tensor possui. Nesse sentido, a dimensionalidade de algum eixo de um tensor será o comprimento desse eixo.

2.3.3 Matrizes

Assim como os vetores generalizam escalares de ordem zero para ordem um, matrizes generalizam vetores de ordem um para ordem dois. Matrizes, que normalmente denotamos com letras maiúsculas em negrito (por exemplo, $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$), são representados no código como tensores com dois eixos.

Em notação matemática, usamos $\mathbf{A} \in \mathbb{R}^{m \times n}$ para expressar que a matriz $\mathbf{A}$ consiste em m linhas e n colunas de escalares com valor real. Visualmente, podemos ilustrar qualquer matriz $\mathbf{A} \in \mathbb{R}^{m \times n}$ como uma tabela, onde cada elemento a_{ij} pertence à linha i^{th} e coluna j^{th}:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{2.3.2}$$

Para qualquer $\mathbf{A} \in \mathbb{R}^{m \times n}$, a forma de $\mathbf{A}$ é (m, n) ou $m \times n$. Especificamente, quando uma matriz tem o mesmo número de linhas e colunas, sua forma se torna um quadrado; portanto, é chamada de *matriz quadrada*.

Podemos criar uma matriz $m \times n$ especificando uma forma com dois componentes m e n ao chamar qualquer uma de nossas funções favoritas para instanciar um tensor.

```
A = torch.arange(20).reshape(5, 4)
A
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
```

Podemos acessar o elemento escalar a_{ij} de uma matriz $\mathbf{A}$ em: eqref: `eq_matrix_def` especificando os índices para a linha (i) e coluna (j), como $[\mathbf{A}]_{ij}$. Quando os elementos escalares de uma matriz $\mathbf{A}$, como em: eqref: `eq_matrix_def`, não são fornecidos, podemos simplesmente usar a letra minúscula da matriz $\mathbf{A}$ com o subscrito do índice, a_{ij}, para se referir a $[\mathbf{A}]_{ij}$. Para manter a notação simples, as vírgulas são inseridas para separar os índices apenas quando necessário, como $a_{2,3j}$ e $[\mathbf{A}]_{2i-1,3}$.

Às vezes, queremos inverter os eixos. Quando trocamos as linhas e colunas de uma matriz, o resultado é chamado de *transposição* da matriz. Formalmente, significamos uma matriz $\mathbf{A}$ transposta por $\mathbf{A}^\top$ e se $\mathbf{B} = \mathbf{A}^\top$, então $b_{ij} = a_{ji}$ para qualquer i e j. Assim, a transposição de $\mathbf{A}$ em: eqref: `eq_matrix_def` é uma matriz $n \times m$:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{2.3.3}$$

Agora acessamoas a matriz transposta via código.

```
A.T
```

```
tensor([[ 0,  4,  8, 12, 16],
        [ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19]])
```

Como um tipo especial de matriz quadrada, a *matriz simétrica* $\mathbf{A}$ é igual à sua transposta: $\mathbf{A} = \mathbf{A}^\top$. Aqui definimos uma matriz simétrica `B`.

```
B = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

```
tensor([[1, 2, 3],
        [2, 0, 4],
        [3, 4, 5]])
```

Agora comparamos `B` com sua transposta.

```
B == B.T
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

Matrizes são estruturas de dados úteis: eles nos permitem organizar dados que têm diferentes modalidades de variação. Por exemplo, as linhas em nossa matriz podem corresponder a diferentes casas (exemplos de dados), enquanto as colunas podem corresponder a diferentes atributos. Isso deve soar familiar se você já usou um software de planilha ou leu: numref: `sec_pandas`. Assim, embora a orientação padrão de um único vetor seja um vetor coluna, em uma matriz que representa um conjunto de dados tabular, é mais convencional tratar cada exemplo de dados como um vetor linha na matriz. E, como veremos em capítulos posteriores, esta convenção permitirá práticas comuns de aprendizado profundo. Por exemplo, ao longo do eixo mais externo de um tensor, podemos acessar ou enumerar *minibatches* de exemplos de dados, ou apenas exemplos de dados se não houver *minibatch*.

2.3.4 Tensores

Assim como vetores generalizam escalares e matrizes generalizam vetores, podemos construir estruturas de dados com ainda mais eixos. Tensores ("tensores" nesta subseção referem-se a objetos algébricos) nos dê uma maneira genérica de descrever matrizes n -dimensionais com um número arbitrário de eixos. Vetores, por exemplo, são tensores de primeira ordem e matrizes são tensores de segunda ordem. Tensores são indicados com letras maiúsculas de uma fonte especial (por exemplo, X, Y, e Z) e seu mecanismo de indexação (por exemplo, x_{ijk} e $[\mathsf{X}]_{1,2i-1,3}$) é semelhante ao de matrizes.

Os tensores se tornarão mais importantes quando começarmos a trabalhar com imagens, que chegam como matrizes n -dimensionais com 3 eixos correspondentes à altura, largura e um eixo de *canal* para empilhar os canais de cores (vermelho, verde e azul). Por enquanto, vamos pular tensores de ordem superior e nos concentrar no básico.

```
X = torch.arange(24).reshape(2, 3, 4)
X
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

2.3.5 Propriedades Básicas de Aritmética de Tensores

Escalares, vetores, matrizes e tensores ("tensores" nesta subseção referem-se a objetos algébricos) de um número arbitrário de eixos têm algumas propriedades interessantes que muitas vezes são úteis. Por exemplo, você deve ter notado da definição de uma operação elemento a elemento que qualquer operação unária elementar não altera a forma de seu operando. Similarmente, dados quaisquer dois tensores com a mesma forma, o resultado de qualquer operação binária elementar será um tensor da mesma forma. Por exemplo, adicionar duas matrizes da mesma forma realiza a adição elemento a elemento sobre essas duas matrizes.

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # Cria uma cópia de `A` em `B` alocando nova memória
A, A + B
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.],
         [16., 17., 18., 19.]]),
 tensor([[ 0.,  2.,  4.,  6.],
         [ 8., 10., 12., 14.],
         [16., 18., 20., 22.],
         [24., 26., 28., 30.],
         [32., 34., 36., 38.]]))
```

Especificamente, a multiplicação elemento a elemento de duas matrizes é chamada de *produto Hadamard* (notação matemática $\odot$). Considere a matriz $\mathbf{B} \in \mathbb{R}^{m\times n}$ cujo elemento da linha i e coluna j é b_{ij}. O produto Hadamard das matrizes $\mathbf{A}$ (definido em (2.3.2)) e $\mathbf{B}$

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{2.3.4}$$

```
A * B
```

```
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
```

Multiplicar ou adicionar um tensor por um escalar também não muda a forma do tensor, onde cada elemento do tensor operando será adicionado ou multiplicado pelo escalar.

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

2.3.6 Redução

Uma operação útil que podemos realizar com tensores arbitrários é para calcule a soma de seus elementos. Em notação matemática, expressamos somas usando o símbolo $\sum$. Para expressar a soma dos elementos em um vetor $\mathbf{x}$ de comprimento d, escrevemos $\sum_{i=1}^{d} x_i$. No código, podemos apenas chamar a função para calcular a soma.

```
x = torch.arange(4, dtype=torch.float32)
x, x.sum()
```

```
(tensor([0., 1., 2., 3.]), tensor(6.))
```

Podemos expressar somas sobre os elementos de tensores de forma arbitrária. Por exemplo, a soma dos elementos de uma matriz $m \times n$ $\mathbf{A}$ poderia ser escrita como $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
A.shape, A.sum()
```

```
(torch.Size([5, 4]), tensor(190.))
```

Por padrão, invocar a função para calcular a soma *reduz* um tensor ao longo de todos os seus eixos a um escalar. Também podemos especificar os eixos ao longo dos quais o tensor é reduzido por meio da soma Pegue as matrizes como exemplo. Para reduzir a dimensão da linha (eixo 0) somando os elementos de todas as linhas, especificamos `axis = 0` ao invocar a função. Uma vez que a matriz de entrada reduz ao longo do eixo 0 para gerar o vetor de saída, a dimensão do eixo 0 da entrada é perdida na forma de saída.

```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

```
(tensor([40., 45., 50., 55.]), torch.Size([4]))
```

Especificando `eixo = 1` irá reduzir a dimensão da coluna (eixo 1) ao somar os elementos de todas as colunas. Assim, a dimensão do eixo 1 da entrada é perdida na forma da saída.

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(tensor([ 6., 22., 38., 54., 70.]), torch.Size([5]))
```

Reduzindo uma matriz ao longo de ambas as linhas e colunas por meio da soma é equivalente a somar todos os elementos da matriz.

```
A.sum(axis=[0, 1])  # O mesmo que `A.sum()`
```

```
tensor(190.)
```

Uma quantidade relacionada é a *média*, que também é chamada de *média*. Calculamos a média dividindo a soma pelo número total de elementos. No código, poderíamos apenas chamar a função para calcular a média em tensores de forma arbitrária.

```
A.mean(), A.sum() / A.numel()
```

```
(tensor(9.5000), tensor(9.5000))
```

Da mesma forma, a função de cálculo da média também pode reduzir um tensor ao longo dos eixos especificados.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(tensor([ 8.,  9., 10., 11.]), tensor([ 8.,  9., 10., 11.]))
```

Soma não reducional

Contudo, às vezes pode ser útil manter o número de eixos inalterado ao invocar a função para calcular a soma ou média.

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A
```

```
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
```

Por exemplo, uma vez que `sum_A` ainda mantém seus dois eixos após somar cada linha, podemos dividir`A` por `sum_A` com *broadcasting*.

```
A / sum_A
```

```
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.1818, 0.2273, 0.2727, 0.3182],
        [0.2105, 0.2368, 0.2632, 0.2895],
        [0.2222, 0.2407, 0.2593, 0.2778],
        [0.2286, 0.2429, 0.2571, 0.2714]])
```

Se quisermos calcular a soma cumulativa dos elementos de `A` ao longo de algum eixo, diga`eixo = 0` (linha por linha), podemos chamar a função `cumsum`. Esta função não reduzirá o tensor de entrada ao longo de nenhum eixo.

```
A.cumsum(axis=0)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.],
        [24., 28., 32., 36.],
        [40., 45., 50., 55.]])
```

2.3.7 Produto Escalar

Até agora, realizamos apenas operações elementares, somas e médias. E se isso fosse tudo que pudéssemos fazer, a álgebra linear provavelmente não mereceria sua própria seção. No entanto, uma das operações mais fundamentais é o produto escalar. Dados dois vetores $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, seu *produto escalar* $\mathbf{x}^\top \mathbf{y}$ (ou $\langle \mathbf{x}, \mathbf{y} \rangle$) é uma soma sobre os produtos dos elementos na mesma posição: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

```
y = torch.ones(4, dtype = torch.float32)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2., 3.]), tensor([1., 1., 1., 1.]), tensor(6.))
```

Observe que podemos expressar o produto escalar de dois vetores de forma equivalente, realizando uma multiplicação elemento a elemento e, em seguida, uma soma:

```
torch.sum(x * y)
```

```
tensor(6.)
```

Os produtos escalares são úteis em uma ampla variedade de contextos. Por exemplo, dado algum conjunto de valores, denotado por um vetor $\mathbf{x} \in \mathbb{R}^d$ e um conjunto de pesos denotado por $\mathbf{w} \in \mathbb{R}^d$,, a soma ponderada dos valores em $\mathbf{x}$ de acordo com os pesos $\mathbf{w}$ pode ser expresso como o produto escalar $\mathbf{x}^\top \mathbf{w}$. Quando os pesos não são negativos e soma a um (ou seja, $\left(\sum_{i=1}^{d} w_i = 1\right)$), o produto escalar expressa uma *média ponderada*. Depois de normalizar dois vetores para ter o comprimento unitário, os produtos escalares expressam o cosseno do ângulo entre eles. Apresentaremos formalmente essa noção de *comprimento* posteriormente nesta seção.

2.3.8 Produtos Matriz-Vetor

Agora que sabemos como calcular produtos escalares, podemos começar a entender *produtos vetoriais de matriz*. Lembre-se da matriz $\mathbf{A} \in \mathbb{R}^{m \times n}$ e o vetor $\mathbf{x} \in \mathbb{R}^n$ definido e visualizado em (2.3.2) e (2.3.1) respectivamente. Vamos começar visualizando a matriz $\mathbf{A}$ em termos de seus vetores linha

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{2.3.5}$$

onde cada $\mathbf{a}_i^\top \in \mathbb{R}^n$ é uma linha vetor representando a ith linha da matriz $\mathbf{A}$.

O produto vetor-matriz $\mathbf{A}\mathbf{x}$ é simplesmente um vetor coluna de comprimento m, cujo elemento i^th é o produto escalar $\mathbf{a}^\top_i \mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{2.3.6}$$

Podemos pensar na multiplicação por uma matriz $\mathbf{A} \in \mathbb{R}^{m \times n}$ como uma transformação que projeta vetores de $\mathbb{R}^n$ a $\mathbb{R}^m$. Essas transformações revelaram-se extremamente úteis. Por exemplo, podemos representar rotações como multiplicações por uma matriz quadrada. Como veremos nos capítulos subsequentes, também podemos usar produtos vetoriais de matriz para descrever os cálculos mais intensivos necessário ao calcular cada camada em uma rede neural dados os valores da camada anterior.

Expressando produtos de vetor-matriz em código com tensores, usamos a mesma função `dot` que para produtos de ponto. Quando chamamos `np.dot (A, x)` com uma matriz `A` e um vetor`x`, o produto matriz-vetor é realizado. Observe que a dimensão da coluna de `A` (seu comprimento ao longo do eixo 1) deve ser igual à dimensão de `x` (seu comprimento).

```
A.shape, x.shape, torch.mv(A, x)
```

```
(torch.Size([5, 4]), torch.Size([4]), tensor([ 14.,  38.,  62.,  86., 110.]))
```

2.3.9 Multiplicação Matriz Matriz

Se você já pegou o jeito dos produtos escalares e produtos matriciais, então a *multiplicação matriz-matriz* deve ser direta. Digamos que temos duas matrizes $\mathbf{A} \in \mathbb{R}^{n \times k}$ e $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix},\quad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{2.3.7}$$

Denotada por $\mathbf{a}^\top_{i} \in \mathbb{R}^k$ o vetor linha representando a ith linha da matriz $\mathbf{A}$, e $\mathbf{b}_{j} \in \mathbb{R}^k$ seja o vetor coluna da j^th coluna matriz $\mathbf{B}$. Para produzir o produto de matrizes $\mathbf{C} = \mathbf{A}\mathbf{B}$, é mais facil pensar $\mathbf{A}$

em termos de seus vetores linha $\mathbf{B}$ em termos de seus vetores coluna:

$$\mathbf{A}=\begin{bmatrix}\mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \end{bmatrix}, \quad \mathbf{B}=\begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \end{bmatrix}. \tag{2.3.8}$$

Então, o produto da matriz $\mathbf{C} \in \mathbb{R}^{n \times m}$ é produzido, pois simplesmente calculamos cada elemento c_{ij} como o produto escalar $\mathbf{a}^\top_i \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix}\mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \end{bmatrix} \begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \end{bmatrix} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{b}_1 & \mathbf{a}^\top_{1}\mathbf{b}_2& \cdots & \mathbf{a}^\top_{1} \mathbf{b}_m \\ \mathbf{a}^\top_{2}\mathbf{b}_1 & \mathbf{a}^\top_{2} \mathbf{b}_2 & \cdots & \mathbf{a}^\top_{2} \mathbf{b}_m \\ \vdots & \vdots & \ddots &\vdots\\ \mathbf{a}^\top_{n} \mathbf{b}_1 & \mathbf{a}^\top_{n}\mathbf{b}_2& \cdots& \mathbf{a}^\top_{n} \mathbf{b}_m \end{bmatrix}. \tag{2.3.9}$$

Podemos pensar na multiplicação matriz-matriz $\mathbf{AB}$ simplesmente realizando m produtos vetoriais de matriz e juntando os resultados para formar uma matriz $n \times m$. No trecho a seguir, realizamos a multiplicação da matriz em `A` e`B`. Aqui, `A` é uma matriz com 5 linhas e 4 colunas, e `B` é uma matriz com 4 linhas e 3 colunas. Após a multiplicação, obtemos uma matriz com 5 linhas e 3 colunas.

```
B = torch.ones(4, 3)
torch.mm(A, B)
```

```
tensor([[ 6.,  6.,  6.],
        [22., 22., 22.],
        [38., 38., 38.],
        [54., 54., 54.],
        [70., 70., 70.]])
```

A multiplicação de matriz-matriz pode ser simplesmente chamada de *multiplicação de matrizes* e não deve ser confundida com o produto Hadamard.

2.3.10 Normas

Alguns dos operadores mais úteis em álgebra linear são *normas*. Informalmente, a norma de um vetor nos diz o quão *grande* é um vetor. A noção de *tamanho* em consideração aqui preocupa-se não em dimensionalidade ,mas sim a magnitude dos componentes.

Na álgebra linear, uma norma vetorial é uma função f que mapeia um vetor para um escalar, satisfazendo um punhado de propriedades. Dado qualquer vetor $\mathbf{x}$, a primeira propriedade diz que se escalarmos todos os elementos de um vetor por um fator constante α, sua norma também escala pelo *valor absoluto* do mesmo fator constante:

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{2.3.10}$$

A segunda propriedade é a familiar desigualdade do triângulo:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{2.3.11}$$

A terceira propriedade simplesmente diz que a norma deve ser não negativa:

$$f(\mathbf{x}) \geq 0. \tag{2.3.12}$$

Isso faz sentido, pois na maioria dos contextos, o menor *tamanho* para qualquer coisa é 0. A propriedade final requer que a menor norma seja alcançada e somente alcançada por um vetor que consiste em todos os zeros.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{2.3.13}$$

Você pode notar que as normas se parecem muito com medidas de distância. E se você se lembra das distâncias euclidianas (pense no teorema de Pitágoras) da escola primária, então, os conceitos de não negatividade e a desigualdade do triângulo podem ser familiares. Na verdade, a distância euclidiana é uma norma: especificamente, é a norma L_2. Suponha que os elementos no vetor n -dimensional **x** são $x_1, \ldots, x_n$.

A L_2 *norma* de **x** é a raiz quadrada da soma dos quadrados dos elementos do vetor:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \tag{2.3.14}$$

onde o subscrito 2 é frequentemente omitido nas normas L_2, ou seja, $\|\mathbf{x}\|$ é equivalente a $\|\mathbf{x}\|_2$. Em código, podemos calcular a norma L_2 de um vetor da seguinte maneira.

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
tensor(5.)
```

Em *Deep Learning*, trabalhamos com mais frequência com a norma L_2 ao quadrado.

Você também encontrará frequentemente a norma L_1, que é expresso como a soma dos valores absolutos dos elementos do vetor:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n \left|x_i \right|. \tag{2.3.15}$$

Em comparação com a norma L_2, é menos influenciado por outliers. Para calcular a norma L_1, nós compomos a função de valor absoluto com uma soma sobre os elementos.

```
torch.abs(u).sum()
```

```
tensor(7.)
```

Tanto a norma L_2 quanto a norma L_1 são casos especiais da norma mais geral L_p:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n \left|x_i \right|^p \right)^{1/p}. \tag{2.3.16}$$

Análogo a L_2 normas de vetores, a *norma de Frobenius* de uma matriz $\mathbf{X} \in \mathbb{R}^{m \times n}$ é a raiz quadrada da soma dos quadrados dos elementos da matriz:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}. \tag{2.3.17}$$

A norma Frobenius satisfaz todas as propriedades das normas vetoriais. Ele se comporta como se fosse uma norma L_2 de um vetor em forma de matriz. Invocar a função a seguir calculará a norma de Frobenius de uma matriz.

```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

Normas e Objetivos

Embora não queiramos nos adiantar muito, já podemos ter uma intuição sobre por que esses conceitos são úteis. No *Deep Learning*, muitas vezes tentamos resolver problemas de otimização: *maximizar* a probabilidade atribuída aos dados observados; *minimizar* a distância entre as previsões e as observações de verdade. Atribuir representações vetoriais a itens (como palavras, produtos ou artigos de notícias) de modo que a distância entre itens semelhantes seja minimizada, e a distância entre itens diferentes é maximizada. Muitas vezes, os objetivos, talvez os componentes mais importantes de algoritmos de *Deep Learning* (além dos dados), são expressos como normas.

2.3.11 Mais sobre Algebra Linear

Apenas nesta seção, nós ensinamos a vocês toda a álgebra linear que você precisa entender um pedaço notável do aprendizado profundo moderno. Há muito mais coisas na álgebra linear e muito dessa matemática é útil para o *Deep Learning*. Por exemplo, as matrizes podem ser decompostas em fatores, e essas decomposições podem revelar estrutura de baixa dimensão em conjuntos de dados do mundo real. Existem subcampos inteiros de *Deep Learning* que se concentram no uso de decomposições de matriz e suas generalizações para tensores de alta ordem para descobrir a estrutura em conjuntos de dados e resolver problemas de previsão. Mas este livro se concentra no *Deep Learning*. E acreditamos que você estará muito mais inclinado a aprender mais matemática depois de sujar as mãos implantar modelos úteis de aprendizado de máquina em conjuntos de dados reais. Portanto, embora nos reservemos o direito de introduzir mais matemática muito mais tarde, vamos encerrar esta seção aqui.

Se voce gostaria de aprender mais sobre Algebra Linear, pode procurar em apêndice online sobre operações de algebra linear[23] ou outra excelente fonte (Strang, 1993; Kolter, 2008; Petersen et al., 2008).

2.3.12 Sumário

- Escalares, vetores, matrizes e tensores são objetos matemáticos básicos em álgebra linear.
- Vetores generalizam escalares e matrizes generalizam vetores.
- Escalares, vetores, matrizes e tensores têm zero, um, dois e um número arbitrário de eixos, respectivamente.
- Um tensor pode ser reduzido ao longo dos eixos especificados por `soma` e `média`.
- A multiplicação elementar de duas matrizes é chamada de produto Hadamard. É diferente da multiplicação de matrizes.
- No aprendizado profundo, geralmente trabalhamos com normas como a norma L_1, a norma L_2 e a norma Frobenius.

[23] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/geometry-linear-algebraic-ops.html

- Podemos realizar uma variedade de operações sobre escalares, vetores, matrizes e tensores.

2.3.13 Exercícios

1. Prove que a transposta de uma matriz **A** transposta é **A**: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Dadas duas matrizes **A** e **B**, mostre que a soma das transpostas é igual à transposta de uma soma: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Dada qualquer matriz quadrada **A**, $\mathbf{A} + \mathbf{A}^\top$ é sempre simétrica? Porque?
4. Definimos o tensor `X` de forma (2, 3, 4) nesta seção. Qual é a saída de `len (X)`?
5. Para um tensor `X` de forma arbitrária, `len (X)`sempre corresponde ao comprimento de um certo eixo de `X`? Qual é esse eixo?
6. Execute `A / A.sum (eixo = 1)` e veja o que acontece. Você pode analisar o motivo?
7. Ao viajar entre dois pontos em Manhattan, qual é a distância que você precisa percorrer em termos de coordenadas, ou seja, em termos de avenidas e ruas? Você pode viajar na diagonal?
8. Considere um tensor com forma (2, 3, 4). Quais são as formas das saídas de soma ao longo dos eixos 0, 1 e 2?
9. Alimente um tensor com 3 ou mais eixos para a função `linalg.norm` e observe sua saída. O que essa função calcula para tensores de forma arbitrária?

Discussions[24]

2.4 Cálculo

Encontrar a área de um polígono permaneceu um mistério até pelo menos 2.500 anos atrás, quando os gregos antigos dividiam um polígono em triângulos e somavam suas áreas. Para encontrar a área de formas curvas, como um círculo, os gregos antigos inscreviam polígonos nessas formas. Conforme mostrado em :numref: `fig_circle_area`, um polígono inscrito com mais lados de igual comprimento se aproxima melhor o circulo. Este processo também é conhecido como *método de exaustão*.

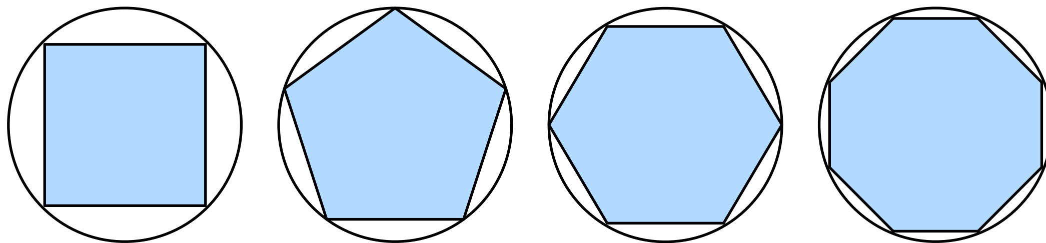


Fig. 2.4.1: Buscando a área de um círculo através do método de exaustão.

Na verdade, o método de exaustão é de onde o *cálculo integral* (será descrito em :numref: `sec_integral_calculus`) se origina. Mais de 2.000 anos depois, o outro ramo do cálculo, *cálculo diferencial*, foi inventado. Entre as aplicações mais críticas do cálculo diferencial, problemas de

[24] https://discuss.d2l.ai/t/31

otimização consideram como fazer algo *o melhor*. Conforme discutido em Section 2.3.10, tais problemas são onipresentes em *Deep Learning*.

Em *Deep Learning*, nós *treinamos* modelos, atualizando-os sucessivamente para que fiquem cada vez melhores à medida que veem mais e mais dados. Normalmente, melhorar significa minimizar uma *função de perda*, uma pontuação que responde à pergunta "quão *ruim* é o nosso modelo?" Esta pergunta é mais sutil do que parece. Em última análise, o que realmente nos preocupa está produzindo um modelo com bom desempenho em dados que nunca vimos antes. Mas só podemos ajustar o modelo aos dados que podemos realmente ver. Assim, podemos decompor a tarefa de ajustar os modelos em duas preocupações principais: i) *otimização*: processo de adequação de nossos modelos aos dados observados; ii) *generalização*: os princípios matemáticos e a sabedoria dos profissionais que guia sobre como produzir modelos cuja validade se estende além do conjunto exato de exemplos de dados usados para treiná-los.

Para te ajudar a entender problemas e métodos de otimização em capítulos posteriores, aqui, damos uma breve introdução ao cálculo diferencial que é comumente usado no *Deep Learning*.

2.4.1 Derivadas e Diferenciação

Começamos abordando o cálculo de derivadas, uma etapa crucial em quase todos os algoritmos de otimização de *Deep Learning*. No *Deep Learning*, normalmente escolhemos funções de perda que são diferenciáveis em relação aos parâmetros do nosso modelo. Simplificando, isso significa que para cada parâmetro, podemos determinar a rapidez com que a perda aumentaria ou diminuiria, deveríamos *aumentar* ou *diminuir* esse parâmetro por uma quantidade infinitesimalmente pequena.

Suponha que temos uma função $f : \mathbb{R} \rightarrow \mathbb{R}$, cuja entrada e saída são escalares. A *derivada* de f é definida como

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \tag{2.4.1}$$

se este limite existe. Se $f'(a)$ existe, Diz-se que f é *diferenciável* em a. Se f é diferenciável a cada número de um intervalo, então esta função é diferenciável neste intervalo. Podemos interpretar a derivada $f'(x)$ em (2.4.1) como a taxa de variação *instantânea* de $f(x)$ em relação a x. A chamada taxa instantânea de mudança é baseada em a variação h em x, que se aproxima de 0.

Para ilustrar derivadas, vamos experimentar com um exemplo. Define-se $u = f(x) = 3x^2 - 4x$.

```
%matplotlib inline
import numpy as np
from IPython import display
from d2l import torch as d2l


def f(x):
    return 3 * x ** 2 - 4 * x
```

Definindo $x = 1$ e deixando h se aproximar de 0, o resultado numérico de $\frac{f(x+h)-f(x)}{h}$ in: eqref: `eq_derivative` aproxima-se de 2. Embora este experimento não seja uma prova matemática, veremos mais tarde que a derivada u' é 2 quando $x = 1$.

```python
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
    print(f'h={h:.5f}, numerical limit={numerical_lim(f, 1, h):.5f}')
    h *= 0.1
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Vamos nos familiarizar com algumas notações equivalentes para derivadas. Dado $y = f(x)$, onde x e y são a variável independente e a variável dependente da função f, respectivamente. As seguintes expressões são equivalentes:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = D_x f(x), \tag{2.4.2}$$

onde os símbolos $\frac{d}{dx}$ e D são *operadores de diferenciação* que indicam operação de *diferenciação*. Podemos usar as seguintes regras para diferenciar funções comuns:

- $DC = 0$ (C é uma constante),
- $Dx^n = nx^{n-1}$ (uma exponenciação, n é qualquer valor real),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

Para diferenciar uma função que é formada de algumas funções mais simples, como as funções comuns acima, as regras a seguir podem ser úteis para nós. Suponha que as funções f e g sejam diferenciáveis e C seja uma constante, temos a *regra múltipla constante*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx} f(x), \tag{2.4.3}$$

a *regra da soma*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx} f(x) + \frac{d}{dx} g(x), \tag{2.4.4}$$

a *regra do produto*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

e a *regra do quociente*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

Agora podemos aplicar algumas das regras acima para encontrar $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x-4$. Assim, definindo $x = 1$, temos $u' = 2$: isso é apoiado por nosso experimento anterior nesta seção onde o resultado numérico se aproxima de 2. Esta derivada também é a inclinação da linha tangente para a curva $u = f(x)$ quando $x = 1$.

Para visualizar tal interpretação das derivadas, usaremos `matplotlib`, uma biblioteca de plotagem popular em Python. Para configurar as propriedades das figuras produzidas por `matplotlib`, precisamos definir algumas funções. Na sequência, a função `use_svg_display` especifica o pacote`matplotlib` para produzir os números SVG para imagens mais nítidas. Observe que o comentário `# @ save` é uma marca especial onde a seguinte função, classe, ou instruções são salvas no pacote `d2l` então, mais tarde, eles podem ser chamados diretamente (por exemplo, `d2l.use_svg_display ()`) sem serem redefinidos.

```
def use_svg_display():  #@save
    """Use the svg format to display a plot in Jupyter."""
    display.set_matplotlib_formats('svg')
```

Definimos a função `set_figsize` para especificar o tamanho das figuras. Observe que aqui usamos diretamente `d2l.plt`, uma vez que a instrução import`from matplotlib import pyplot as plt` foi marcada para ser salva no pacote `d2l` no prefácio.

```
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

A seguinte função `set_axes` define as propriedades dos eixos das figuras produzidas por`matplotlib`.

```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

Com essas três funções para configurações de figura, nós definimos a função `plot` para traçar várias curvas sucintamente uma vez que precisaremos visualizar muitas curvas ao longo do livro.

```
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if `X` (tensor or list) has 1 axis
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
```

(continues on next page)

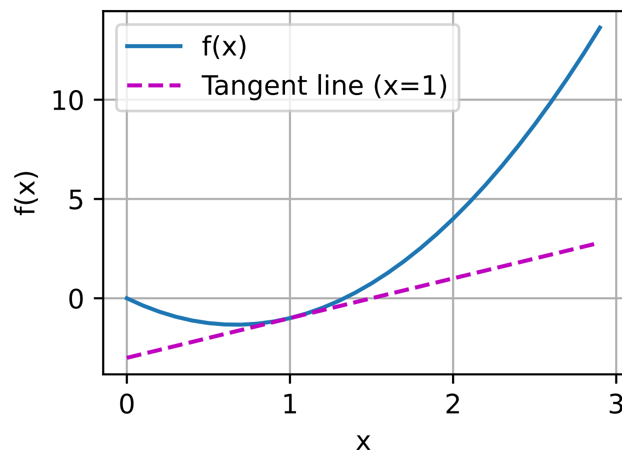
(continued from previous page)

```
                 and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

Agora podemos plotar a função $u = f(x)$ e sua linha tangente $y = 2x - 3$ em $x = 1$, onde o coeficiente 2 é a inclinação da linha tangente .

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```

```
/tmp/ipykernel_104167/77894834.py:3: DeprecationWarning: set_matplotlib_formats is␣
↪deprecated since IPython 7.23, directly use matplotlib_inline.backend_inline.set_
↪matplotlib_formats()
  display.set_matplotlib_formats('svg')
```



2.4.2 Derivadas Parciais

Até agora, lidamos com a diferenciação de funções de apenas uma variável. No *Deep Learning*, as funções geralmente dependem de *muitas* variáveis. Portanto, precisamos estender as idéias de diferenciação para essas funções *multivariadas*.

Seja $y = f(x_1, x_2, \ldots, x_n)$ uma função com n variáveis. A *derivada parcial* de y em relação ao seu i^th parâmetro x_i é

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \ldots, x_{i-1}, x_i+h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

Para calcular $\frac{\partial y}{\partial x_i}$, podemos simplesmente tratar $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ como constantes e calcular a derivada de y com respeito a x_i. Para notação de derivadas parciais, os seguintes são equivalentes:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.8}$$

2.4.3 Gradientes

Podemos concatenar derivadas parciais de uma função multivariada com respeito a todas as suas variáveis para obter o vetor *gradiente* da função. Suponha que a entrada da função $f: \mathbb{R}^n \rightarrow \mathbb{R}$ seja um vetor n -dimensional $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ e a saída é um escalar. O gradiente da função $f(\mathbf{x})$ em relação a $\mathbf{x}$ é um vetor de n derivadas parciais:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \bigg[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\bigg]^\top, \tag{2.4.9}$$

onde $\nabla_{\mathbf{x}} f(\mathbf{x})$ é frequentemente substituído por $\nabla f(\mathbf{x})$ quando não há ambiguidade.

Seja $\mathbf{x}$ um vetor n -dimensional, as seguintes regras são freqüentemente usadas ao diferenciar funções multivariadas:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A} \mathbf{x} = \mathbf{A}^\top$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x} \|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Da mesma forma, para qualquer matriz $\mathbf{X}$, temos $\nabla_{\mathbf{X}} \|\mathbf{X} \|_F^2 = 2\mathbf{X}$. Como veremos mais tarde, os gradientes são úteis para projetar algoritmos de otimização em *deep learning*.

2.4.4 Regra da Cadeia

No entanto, esses gradientes podem ser difíceis de encontrar. Isso ocorre porque as funções multivariadas no *deep learning* são frequentemente *compostas*, portanto, não podemos aplicar nenhuma das regras mencionadas acima para diferenciar essas funções. Felizmente, a *regra da cadeia* nos permite diferenciar funções compostas.

Vamos primeiro considerar as funções de uma única variável. Suponha que as funções $y = f(u)$ e $u = g(x)$ sejam diferenciáveis, então a regra da cadeia afirma que

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}. \tag{2.4.10}$$

Agora, vamos voltar nossa atenção para um cenário mais geral onde as funções têm um número arbitrário de variáveis. Suponha que a função diferenciável y tenha variáveis $u_1, u_2, \ldots, u_m$, onde cada função diferenciável u_i tem variáveis $x_1, x_2, \ldots, x_n$. Observe que y é uma função de $x_1, x_2, \ldots, x_n$. Então a regra da cadeia será

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{2.4.11}$$

para qualquer $i = 1, 2, \ldots, n$.

2.4.5 Sumário

- Cálculo diferencial e cálculo integral são dois ramos do cálculo, onde o primeiro pode ser aplicado aos problemas de otimização onipresentes no *deep learning*.
- Uma derivada pode ser interpretada como a taxa instantânea de mudança de uma função em relação à sua variável. É também a inclinação da linha tangente à curva da função.
- Um gradiente é um vetor cujos componentes são as derivadas parciais de uma função multivariada com respeito a todas as suas variáveis.
- A regra da cadeia nos permite diferenciar funções compostas.

2.4.6 Exercícios

1. Trace a função $y = f(x) = x^3 - \frac{1}{x}$ e sua linha tangente quando $x = 1$.
2. Encontre o gradiente da função $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
3. Qual é o gradiente da função $f(\mathbf{x}) = \|\mathbf{x}\|_2$?
4. Você pode escrever a regra da cadeia para o caso em que $u = f(x, y, z)$ e $x = x(a, b)$, $y = y(a, b)$, e $z = z(a, b)$?

Discussions[25]

2.5 Diferenciação automática

Como já explicado em Section 2.4, a diferenciação é uma etapa crucial em quase todos os algoritmos de otimização de *Deep Learning*. Embora os cálculos para obter esses derivados sejam diretos, exigindo apenas alguns cálculos básicos, para modelos complexos, trabalhando as atualizações manualmente pode ser uma tarefa difícil (e muitas vezes sujeita a erros). *Frameworks* de *Deep learning* aceleram este trabalho calculando automaticamente as derivadas, ou seja, *diferenciação automática*. Na prática, com base em nosso modelo projetado o sistema constrói um *grafo computacional*, rastreando quais dados combinados por meio de quais operações produzem a saída. A diferenciação automática permite que o sistema propague gradientes posteriormente. Aqui, propagar(do Inglês *backpropagate*) significa simplesmente traçar o gráfico computacional, preencher as derivadas parciais em relação a cada parâmetro.

[25] https://discuss.d2l.ai/t/33

2.5.1 Um exemplo simples

Como exemplo, digamos que estamos interessados em derivar a função $y = 2\mathbf{x}^\top\mathbf{x}$ com respeito ao vetor coluna $\mathbf{x}$. Inicialmente criamos a variável x e atribuimos a ela um valor inicial.

```
import torch

x = torch.arange(4.0)
x
```

```
tensor([0., 1., 2., 3.])
```

Antes de calcularmos o gradiente de$y$ em relação a $\mathbf{x}$, precisamos armazena-lo. É importante que não aloquemos nova memória cada vez que tomamos uma derivada em relação a um parâmetro porque costumamos atualizar os mesmos parâmetros milhares ou milhões de vezes e podemos rapidamente ficar sem memória. Observe que um gradiente de uma função com valor escalar com respeito a um vetor $\mathbf{x}$ tem valor vetorial e tem a mesma forma de $\mathbf{x}$.

```
x.requires_grad_(True)  # Same as `x = torch.arange(4.0, requires_grad=True)`
x.grad  # O valor padrão é None
```

Então calcularemos y.

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

Uma vez que x é um vetor de comprimento 4, um produto interno de x ex é realizado, produzindo a saída escalar que atribuímos a y. Em seguida, podemos calcular automaticamente o gradiente de y com relação a cada componente de x chamando a função de retropropagação e imprimindo o gradiente.

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

O gradiente da função $y = 2\mathbf{x}^\top\mathbf{x}$ em relação a $\mathbf{x}$ should be $4\mathbf{x}$. Vamos verificar rapidamente se nosso gradiente desejado foi calculado corretamente.

```
x.grad == 4 * x
```

```
tensor([True, True, True, True])
```

Agora calculamos outra função de x.

```
# O PyTorch acumula os gradientes por padrão, precisamos
# apagar os valores anteriores
```

(continues on next page)

(continued from previous page)

```
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

```
tensor([1., 1., 1., 1.])
```

2.5.2 Retroceder para variáveis não escalares

Tecnicamente, quando y não é um escalar, a interpretação mais natural da diferenciação de um vetor y em relação a um vetor, x é uma matriz. Para y ex de ordem superior e dimensão superior, o resultado da diferenciação pode ser um tensor de ordem alta.

No entanto, embora esses objetos mais exóticos apareçam em aprendizado de máquina avançado (incluindo em *Deep Learning*), com mais frequência quando estamos retrocedendo um vetor, estamos tentando calcular as derivadas das funções de perda para cada constituinte de um *lote* de exemplos de treinamento. Aqui, nossa intenção é não calcular a matriz de diferenciação mas sim a soma das derivadas parciais calculado individualmente para cada exemplo no lote.

```
# Invocar `backward` em um não escalar requer passar um argumento `gradient`
# que especifica o gradiente da função diferenciada w.r.t `self`.
# Em nosso caso, simplesmente queremos somar as derivadas parciais, assim passando
# em um gradiente de uns é apropriado
x.grad.zero_()
y = x * x
# y.backward(torch.ones(len(x))) equivalente a:
y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

2.5.3 Computação *Detaching*

Às vezes, desejamos mover alguns cálculos fora do gráfico computacional registrado. Por exemplo, digamos que y foi calculado como uma função dex, e que subsequentemente z foi calculado como uma função dey e x. Agora, imagine que quiséssemos calcular o gradiente de z em relação ax, mas queria, por algum motivo, tratar y como uma constante, e apenas leve em consideração a função que x jogou apósy foi calculado. Aqui, podemos desanexar y para retornar uma nova variável u que tem o mesmo valor que y, mas descarta qualquer informação sobre como y foi calculado no grafo computacional. Em outras palavras, o gradiente não fluirá de volta de u para x. Assim, a seguinte função de retropropagação calcula a derivada parcial de `z = u * x` com respeito ax enquanto trata u como uma constante, em vez da derivada parcial de `z = x * x * x` em relação ax.

```
x.grad.zero_()
y = x * x
```

(continues on next page)

(continued from previous page)

```
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

```
tensor([True, True, True, True])
```

Uma vez que o cálculo de y foi registrado, podemos subsequentemente invocar a retropropagação em y para obter a derivada de `y = x * x` com respeito a x, que é `2 * x`.

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

```
tensor([True, True, True, True])
```

2.5.4 Computando o Gradiente do *Python Control Flow*

Uma vantagem de usar a diferenciação automática é que mesmo se construir o gráfo computacional de uma função requer muito trabalho com o uso do *Python Control Flow* (por exemplo, condicionais, loops e chamadas de função arbitrárias), ainda podemos calcular o gradiente da variável resultante. No trecho a seguir, observe que o número de iterações do loop `while` e a avaliação da instrução `if` ambos dependem do valor da entrada `a`.

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Vamos computar o gradiente:

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Agora podemos analisar a função `f` definida acima. Observe que é linear por partes em sua entrada `a`. Em outras palavras, para qualquer `a` existe algum escalar constante `k` tal que `f (a) = k * a`, onde o valor de `k` depende da entrada `a`. Consequentemente, `d / a` nos permite verificar se o gradiente está correto.

```
a.grad == d / a
```

```
tensor(True)
```

2.5.5 Sumário

- *Frameworks* de *Deep learning* podem automatizar o cálculo de derivadas. Para usá-lo, primeiro anexamos gradientes às variáveis em relação às quais desejamos as derivadas parciais. Em seguida, registramos o cálculo de nosso valor alvo, executamos sua função para retropropagação e acessamos o gradiente resultante.

2.5.6 Exercícios

1. Por que a segunda derivada é muito mais computacionalmente cara de se calcular do que a primeira derivada?
2. Depois de executar a função de retropropagação, execute-a imediatamente novamente e veja o que acontece.
3. No exemplo de fluxo de controle onde calculamos a derivada de `d` com respeito a `a`, o que aconteceria se mudássemos a variável `a` para um vetor ou matriz aleatória. Neste ponto, o resultado do cálculo `f (a)` não é mais um escalar. O que acontece com o resultado? Como analisamos isso?
4. Redesenhe um exemplo para encontrar o gradiente do *Control Flow*. Execute e analise o resultado.
5. Seja $f(x) = sin(x)$. Plote $f(x)$ e $fracdf(x)dx$, onde o último é calculado sem explorar que $f'(x) = cos(x)$.

Discussions[26]

2.6 Probabilidade

De uma forma ou de outra, o aprendizado de máquina envolve fazer previsões. Podemos querer prever a *probabilidade* de um paciente sofrer um ataque cardíaco no próximo ano, considerando seu histórico clínico. Na detecção de anomalias, podemos avaliar quão *provável* seria um conjunto de leituras do motor a jato de um avião, se ele estivesse operando normalmente. Na aprendizagem por reforço, queremos que um agente aja de forma inteligente em um ambiente. Isso significa que precisamos pensar sobre a probabilidade de obter uma alta recompensa em cada uma das ações disponíveis. E quando construímos sistemas de recomendação, também precisamos pensar sobre probabilidade. Por exemplo, diga *hipoteticamente* que trabalhamos para uma grande livraria online. Podemos querer estimar a probabilidade de um determinado usuário comprar um determinado livro. Para isso, precisamos usar a linguagem da probabilidade. Cursos inteiros, graduações, teses, carreiras e até departamentos são dedicados à probabilidade. Então, naturalmente, nosso objetivo nesta seção não é ensinar todo o assunto. Em vez disso, esperamos fazer você decolar, ensinar apenas o suficiente para que você possa começar a construir seus primeiros modelos de *Deep Learning* e dar-lhe um sabor suficiente para o assunto que você pode começar a explorá-lo por conta própria, se desejar.

[26] https://discuss.d2l.ai/t/35

Já invocamos as probabilidades nas seções anteriores, sem articular o que são precisamente ou dar um exemplo concreto. Vamos ser mais sérios agora, considerando o primeiro caso: distinguir cães e gatos com base em fotografias. Isso pode parecer simples, mas na verdade é um desafio formidável. Para começar, a dificuldade do problema pode depender da resolução da imagem.

Fig. 2.6.1: Imagens de diferentes resoluções (10×10, 20×20, 40×40, 80×80, e 160×160 pixels).

Conforme mostrado em Fig. 2.6.1, embora seja fácil para os humanos reconhecerem cães e gatos na resolução de 160×160 pixels, torna-se um desafio em 40×40 pixels e quase impossível em 10×10 pixels. Em outras palavras, nossa capacidade de distinguir cães e gatos a uma grande distância (e, portanto, em baixa resolução) pode se aproximar de uma suposição desinformada. A probabilidade nos dá um maneira formal de raciocinar sobre nosso nível de certeza. Se tivermos certeza absoluta que a imagem representa um gato, dizemos que a *probabilidade* de que o rótulo y correspondente seja "cat", denotado $P(y = \text{“cat”})$ é igual a 1. Se não tivéssemos nenhuma evidência para sugerir que $y = \text{“cat”}$ ou que $y = \text{“dog”}$, então poderíamos dizer que as duas possibilidades eram igualmente *provavelmente* expressando isso como $P(y = \text{“cat”}) = P(y = \text{“dog”}) = 0.5$. Se estivéssemos razoavelmente confiantes, mas não temos certeza de que a imagem representava um gato, podemos atribuir um probabilidade $0,5 <P (y = "$cat$") <1$.

Agora considere o segundo caso: dados alguns dados de monitoramento do tempo, queremos prever a probabilidade de que choverá em Taipei amanhã. Se for verão, a chuva pode vir com probabilidade 0,5.

Em ambos os casos, temos algum valor de interesse. E em ambos os casos não temos certeza sobre o resultado. Mas existe uma diferença fundamental entre os dois casos. Neste primeiro caso, a imagem é de fato um cachorro ou um gato, e simplesmente não sabemos qual. No segundo caso, o resultado pode realmente ser um evento aleatório, se você acredita em tais coisas (e a maioria dos físicos acredita). Portanto, probabilidade é uma linguagem flexível para raciocinar sobre nosso nível de certeza e pode ser aplicada com eficácia em um amplo conjunto de contextos.

2.6.1 Teoria Básica de Probabilidade

Digamos que lançamos um dado e queremos saber qual é a chance de ver um 1 em vez de outro dígito. Se o dado for justo, todos os seis resultados $\{1, \ldots, 6\}$ têm a mesma probabilidade de ocorrer e, portanto, veríamos 1 em um dos seis casos. Formalmente afirmamos que 1 ocorre com probabilidade $\frac{1}{6}$.

Para um dado real que recebemos de uma fábrica, podemos não saber essas proporções e precisaríamos verificar se ele está contaminado. A única maneira de investigar o dado é lançando-o várias vezes e registrando os resultados. Para cada lançamento do dado, observaremos um valor em $\{1, \ldots, 6\}$. Dados esses resultados, queremos investigar a probabilidade de observar cada resultado.

Uma abordagem natural para cada valor é pegar o contagem individual para aquele valor e dividi-lo pelo número total de jogadas. Isso nos dá uma *estimativa* da probabilidade de um determinado *evento*. A *lei de grandes números* nos dizem que, conforme o número de lançamentos aumenta, essa estimativa se aproxima cada vez mais da verdadeira probabilidade subjacente. Antes de entrar em detalhes sobre o que está acontecendo aqui, vamos experimentar.

Para começar, importemos os pacotes necessários.

```
%matplotlib inline
import torch
from torch.distributions import multinomial
from d2l import torch as d2l
```

Em seguida, queremos ser capazes de lançar o dado. Nas estatísticas, chamamos este processo de colher exemplos de *amostragem* de distribuições de probabilidade . A distribuição que atribui probabilidades a uma série de escolhas discretas é chamado de *distribuição multinomial*. Daremos uma definição mais formal de *distribuição* mais tarde, mas em um alto nível, pense nisso como apenas uma atribuição de probabilidades para eventos.

Para obter uma única amostra, simplesmente passamos um vetor de probabilidades. A saída é outro vetor do mesmo comprimento: seu valor no índice i é o número de vezes que o resultado da amostragem corresponde a i.

```
fair_probs = torch.ones([6]) / 6
multinomial.Multinomial(1, fair_probs).sample()
```

```
tensor([0., 0., 0., 0., 1., 0.])
```

Se você executar o amostrador várias vezes, descobrirá que sai aleatoriamente valores de cada vez. Tal como acontece com a estimativa da justiça de um dado, muitas vezes queremos gerar muitas amostras da mesma distribuição. Seria insuportavelmente lento para fazer isso com um loop Python `for`, então a função que estamos usando suporta gerar várias amostras de uma vez, retornando uma matriz de amostras independentes em qualquer forma podemos desejar.

```
multinomial.Multinomial(10, fair_probs).sample()
```

```
tensor([0., 4., 1., 1., 2., 2.])
```

Agora que sabemos como obter amostras de um dado, podemos simular 1000 execuções. Podemos

então passar e contar, após cada um dos 1000 lançamentos, quantas vezes cada número foi rolado. Especificamente, calculamos a frequência relativa como a estimativa da probabilidade verdadeira.

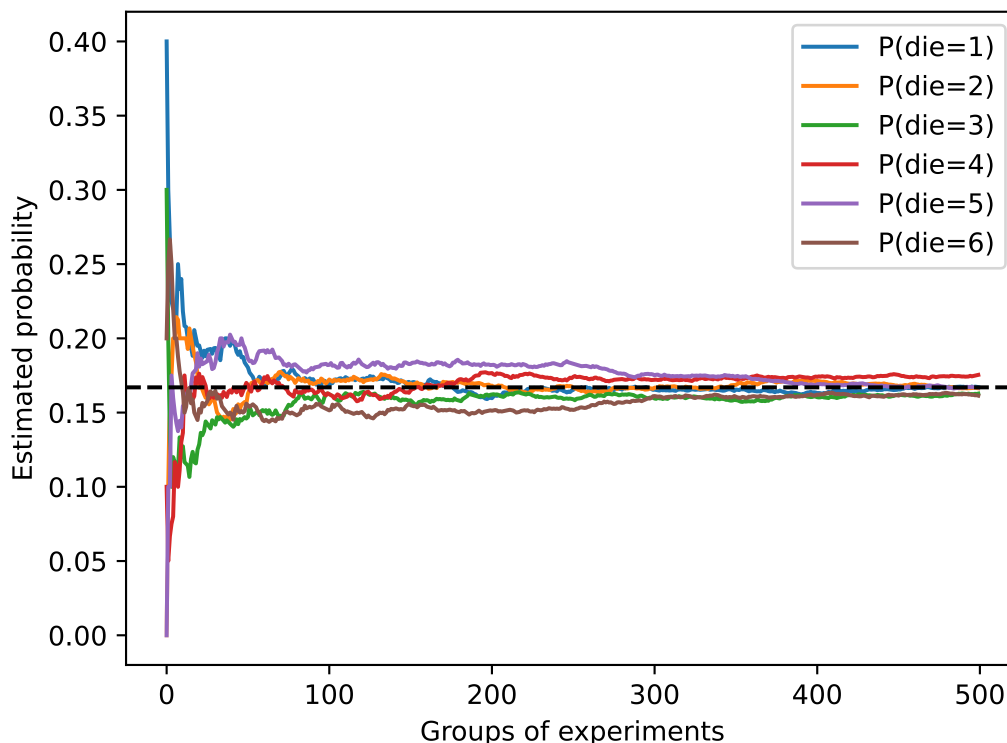
```
# Store the results as 32-bit floats for division
counts = multinomial.Multinomial(1000, fair_probs).sample()
counts / 1000  # Relative frequency as the estimate
```

```
tensor([0.1830, 0.1600, 0.1660, 0.1380, 0.1570, 0.1960])
```

Como geramos os dados de um dado justo, sabemos que cada resultado tem probabilidade real $\frac{1}{6}$, cerca de $0,167$, portanto, as estimativas de saída acima parecem boas.

Também podemos visualizar como essas probabilidades convergem ao longo do tempo para a probabilidade verdadeira. Vamos conduzir 500 grupos de experimentos onde cada grupo extrai 10 amostras.

```
counts = multinomial.Multinomial(10, fair_probs).sample((500,))
cum_counts = counts.cumsum(dim=0)
estimates = cum_counts / cum_counts.sum(dim=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].numpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Groups of experiments')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```



Cada curva sólida corresponde a um dos seis valores do dado e dá nossa probabilidade estimada de

que o dado aumente esse valor conforme avaliado após cada grupo de experimentos. A linha preta tracejada fornece a verdadeira probabilidade subjacente. À medida que obtemos mais dados conduzindo mais experimentos, as curvas sólidas de 6 convergem para a probabilidade verdadeira.

Axiomas da Teoria de Probabilidade

Ao lidar com as jogadas de um dado, chamamos o conjunto $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ o *espaço de amostra* ou *espaço de resultado*, onde cada elemento é um *resultado*. Um *evento* é um conjunto de resultados de um determinado espaço amostral. Por exemplo, "ver 5" ($\{5\}$) e "ver um número ímpar" ($\{1, 3, 5\}$) são eventos válidos de lançar um dado. Observe que se o resultado de um experimento aleatório estiver no evento $\mathcal{A}$, então o evento $\mathcal{A}$ ocorreu. Ou seja, se 3 pontos virados para cima após rolar um dado, uma vez que $3 \in \{1, 3, 5\}$, podemos dizer que o evento "ver um número ímpar" ocorreu.

Formalmente, *probabilidade* pode ser pensada como uma função que mapeia um conjunto para um valor real. A probabilidade de um evento $\mathcal{A}$ no espaço amostral dado $\mathcal{S}$, denotado como $P(\mathcal{A})$, satisfaz as seguintes propriedades:

- Para qualquer evento $\mathcal{A}$, sua probabilidade nunca é negativa, ou seja, $P(\mathcal{A}) \geq 0$;
- A probabilidade de todo o espaço amostral é 1, ou seja, $P(\mathcal{S}) = 1$;
- Para qualquer sequência contável de eventos $\mathcal{A}_1, \mathcal{A}_2, \ldots$ que são *mutuamente exclusivos* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ para todo $i \neq j$), a probabilidade de que aconteça é igual à soma de suas probabilidades individuais, ou seja, $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

Esses também são os axiomas da teoria das probabilidades, propostos por Kolmogorov em 1933. Graças a este sistema de axiomas, podemos evitar qualquer disputa filosófica sobre aleatoriedade; em vez disso, podemos raciocinar rigorosamente com uma linguagem matemática. Por exemplo, permitindo que o evento $\mathcal{A}_1$ seja todo o espaço da amostra e $\mathcal{A}_i = \emptyset$ para todos $i > 1$, podemos provar que $P(\emptyset) = 0$, ou seja, a probabilidade de um evento impossível é 0.

Variáveis Aleatórias

Em nosso experimento aleatório de lançar um dado, introduzimos a noção de uma *variável aleatória*. Uma variável aleatória pode ser praticamente qualquer quantidade e não é determinística. Pode assumir um valor entre um conjunto de possibilidades em um experimento aleatório. Considere uma variável aleatória X cujo valor está no espaço amostral $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ do lançamento de um dado. Podemos denotar o evento "vendo 5" como $\{X = 5\}$ ou $X = 5$, e sua probabilidade como $P(\{X = 5\})$ ou $P(X = 5)$. Por $P(X = a)$, fazemos uma distinção entre a variável aleatória X e os valores (por exemplo, a) que X pode assumir. No entanto, esse pedantismo resulta em uma notação complicada. Para uma notação compacta, por um lado, podemos apenas denotar $P(X)$ como a *distribuição* sobre a variável aleatória X: a distribuição nos diz a probabilidade de que X assuma qualquer valor. Por outro lado, podemos simplesmente escrever $P(a)$ para denotar a probabilidade de uma variável aleatória assumir o valor a. Uma vez que um evento na teoria da probabilidade é um conjunto de resultados do espaço amostral, podemos especificar um intervalo de valores para uma variável aleatória assumir. Por exemplo, $P(1 \leq X \leq 3)$ denota a probabilidade do evento $\{1 \leq X \leq 3\}$, o que significa $\{X = 1, 2, \text{or}, 3\}$. De forma equivalente, $\{X = 1, 2, \text{or}, 3\}$ representa a probabilidade de que a variável aleatória X possa assumir um valor de $\{1, 2, 3\}$.

Observe que há uma diferença sutil entre variáveis aleatórias *discretas*, como os lados de um dado, e *contínuas*, como o peso e a altura de uma pessoa. Não adianta perguntar se duas pes-

soas têm exatamente a mesma altura. Se tomarmos medidas precisas o suficiente, você descobrirá que duas pessoas no planeta não têm exatamente a mesma altura. Na verdade, se fizermos uma medição suficientemente precisa, você não terá a mesma altura ao acordar e ao dormir. Portanto, não há nenhum propósito em perguntar sobre a probabilidade que alguém tem 1,80139278291028719210196740527486202 metros de altura. Dada a população mundial de humanos, a probabilidade é virtualmente 0. Faz mais sentido, neste caso, perguntar se a altura de alguém cai em um determinado intervalo, digamos entre 1,79 e 1,81 metros. Nesses casos, quantificamos a probabilidade de vermos um valor como uma *densidade*. A altura de exatamente 1,80 metros não tem probabilidade, mas densidade diferente de zero. No intervalo entre quaisquer duas alturas diferentes, temos probabilidade diferente de zero. No restante desta seção, consideramos a probabilidade no espaço discreto. Para probabilidade sobre variáveis aleatórias contínuas, você pode consultar Section 18.6.

2.6.2 Lidando com Múltiplas Variáveis Aleatórias

Muitas vezes, queremos considerar mais de uma variável aleatória de cada vez. Por exemplo, podemos querer modelar a relação entre doenças e sintomas. Dados uma doença e um sintoma, digamos "gripe" e "tosse", podem ou não ocorrer em um paciente com alguma probabilidade. Embora esperemos que a probabilidade de ambos seja próxima de zero, podemos estimar essas probabilidades e suas relações entre si para que possamos aplicar nossas inferências para obter um melhor atendimento médico.

Como um exemplo mais complicado, as imagens contêm milhões de pixels, portanto, milhões de variáveis aleatórias. E, em muitos casos, as imagens vêm com um rótulo, identificando objetos na imagem. Também podemos pensar no rótulo como um variável aleatória. Podemos até pensar em todos os metadados como variáveis aleatórias como local, tempo, abertura, comprimento focal, ISO, distância de foco e tipo de câmera. Todas essas são variáveis aleatórias que ocorrem em conjunto. Quando lidamos com múltiplas variáveis aleatórias, existem várias quantidades de interesse.

Probabilidade Conjunta

O primeiro é chamado de *probabilidade conjunta* $P(A = a, B = b)$. Dados quaisquer valores a e b, a probabilidade conjunta nos permite responder, qual é a probabilidade de que $A = a$ e $B = b$ simultaneamente? Observe que, para quaisquer valores a e b, $P(A = a, B = b) \leq P(A = a)$. Tem de ser este o caso, visto que para $A = a$ e $B = b$ acontecer, $A = a$ tem que acontecer *e* $B = b$ também tem que acontecer (e vice-versa). Assim, $A = a$ e $B = b$ não podem ser mais prováveis do que $A = a$ ou $B = b$ individualmente.

Probabilidade Condicional

Isso nos leva a uma razão interessante: $0 \leq \frac{P(A=a,B=b)}{P(A=a)} \leq 1$. Chamamos essa proporção de *probabilidade condicional* e denotá-lo por $P(B = b \mid A = a)$: é a probabilidade de $B = b$, desde que $A = a$ ocorreu.

Teorema de Bayes

Usando a definição de probabilidades condicionais, podemos derivar uma das equações mais úteis e celebradas em estatística: *Teorema de Bayes*. Por construção, temos a *regra de multiplicação* que $P(A,B) = P(B \mid A)P(A)$. Por simetria, isso também é válido para $P(A,B) = P(A \mid B)P(B)$. Suponha que $P(B) > 0$. Resolvendo para uma das variáveis condicionais, obtemos

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{2.6.1}$$

Observe que aqui usamos a notação mais compacta em que $P(A,B)$ é uma *distribuição conjunta* e $P(A \mid B)$ é uma *distribuição condicional*. Essas distribuições podem ser avaliadas para valores particulares $A = a, B = b$.

Marginalização

O teorema de Bayes é muito útil se quisermos inferir uma coisa da outra, digamos causa e efeito, mas só conhecemos as propriedades na direção reversa, como veremos mais adiante nesta seção. Uma operação importante de que precisamos para fazer esse trabalho é a *marginalização*. É a operação de determinar $P(B)$ de $P(A,B)$. Podemos ver que a probabilidade de B equivale a contabilizar todas as escolhas possíveis de A e agregar as probabilidades conjuntas de todas elas:

$$P(B) = \sum_{A} P(A,B), \tag{2.6.2}$$

que também é conhecida como *regra da soma*. A probabilidade ou distribuição como resultado da marginalização é chamada de *probabilidade marginal* ou *distribuição marginal*.

Independencia

Outra propriedade útil para verificar é *dependência* vs. *independência*. Duas variáveis aleatórias A e B sendo independentes significa que a ocorrência de um evento de A não revela nenhuma informação sobre a ocorrência de um evento de B. Neste caso $P(B \mid A) = P(B)$. Os estatísticos normalmente expressam isso como $A \perp B$. Do teorema de Bayes, segue imediatamente que também $P(A \mid B) = P(A)$. Em todos os outros casos, chamamos A e B de dependente. Por exemplo, duas jogadas sucessivas de um dado são independentes. Em contraste, a posição de um interruptor de luz e a luminosidade da sala não são (eles não são perfeitamente determinísticos, pois podemos sempre ter uma lâmpada quebrada, falha de energia ou um interruptor quebrado).

Dado que $P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ é equivalente a $P(A,B) = P(A)P(B)$, duas variáveis aleatórias são independentes se e somente se sua distribuição conjunta é o produto de suas distribuições individuais. Da mesma forma, duas variáveis aleatórias A e B são *condicionalmente independentes* dada outra variável aleatória C se e somente se $P(A,B \mid C) = P(A \mid C)P(B \mid C)$. Isso é expresso como $A \perp B \mid C$.

Aplicação

Vamos colocar nossas habilidades à prova. Suponha que um médico administre um teste de HIV a um paciente. Este teste é bastante preciso e falha apenas com 1% de probabilidade se o paciente for saudável, mas relatá-lo como doente. Além disso, nunca deixa de detectar o HIV se o paciente realmente o tiver. Usamos D_1 para indicar o diagnóstico (1 se positivo e 0 se negativo) e H para denotar o estado de HIV (1 se positivo e 0 se negativo). `conditional_prob_D1` lista tais probabilidades condicionais. : Probabilidade condicional de $P(D_1 \mid H)$.

Table 2.6.1: label:`conditional_prob_D1`

Probabilidade Condicional	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

Observe que as somas das colunas são todas 1 (mas as somas das linhas não), uma vez que a probabilidade condicional precisa somar 1, assim como a probabilidade. Vamos calcular a probabilidade de o paciente ter HIV se o teste der positivo, ou seja, $P(H = 1 \mid D_1 = 1)$. Obviamente, isso vai depender de quão comum é a doença, já que afeta o número de alarmes falsos. Suponha que a população seja bastante saudável, por exemplo, $P(H = 1) = 0.0015$. Para aplicar o teorema de Bayes, precisamos aplicar a marginalização e a regra de multiplicação para determinar

$$\begin{aligned}
&P(D_1 = 1) \\
=&P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\
=&P(D_1 = 1 \mid H = 0) P(H = 0) + P(D_1 = 1 \mid H = 1) P(H = 1) \\
=&0.011485.
\end{aligned} \tag{2.6.3}$$

Portanto, obtemos:

$$\begin{aligned}
&P(H = 1 \mid D_1 = 1)\\
=& \frac{P(D_1=1 \mid H=1) P(H=1)}{P(D_1=1)}. \\
=&0.1306
\end{aligned} \tag{2.6.4}$$

Em outras palavras, há apenas 13,06% de chance de que o paciente realmente tem HIV, apesar de usar um teste muito preciso. Como podemos ver, a probabilidade pode ser contra-intuitiva.

O que o paciente deve fazer ao receber notícias tão terríveis? Provavelmente, o paciente pediria ao médico para administrar outro teste para obter clareza. O segundo teste tem características diferentes e não é tão bom quanto o primeiro, como mostrado em :numref:conditional_prob_D2`.

Table 2.6.2: Probabilidade Condicional de $P(D_2 \mid H)$.

Probabilidade Condicional	$H = 1$	$H = 0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

Infelizmente, o segundo teste também deu positivo. Vamos trabalhar as probabilidades necessárias para invocar o teorema de Bayes assumindo a independência condicional:

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1 \mid H = 0) \\
=& P(D_1 = 1 \mid H = 0) P(D_2 = 1 \mid H = 0) \\
=& 0.0003,
\end{aligned} \tag{2.6.5}$$

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1 \mid H = 1)\\
=&P(D_1 = 1 \mid H = 1) P(D_2 = 1 \mid H = 1)\\
=&0.98.
\end{aligned} \tag{2.6.6}$$

Agora podemos aplicar a marginalização e a regra de multiplicação:

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1)\\
=&P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\
=&P(D_1 = 1, D_2 = 1 \mid H = 0)P(H=0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H=1)\\
=&0.00176955.
\end{aligned} \tag{2.6.7}$$

No final, a probabilidade de o paciente ter HIV, dado ambos os testes positivos, é

$$\begin{aligned}
&P(H = 1 \mid D_1 = 1, D_2 = 1)\\
=&\frac{P(D_1 = 1, D_2 = 1 \mid H=1) P(H=1)}{P(D_1 = 1, D_2 = 1)}\\
=&0.8307.
\end{aligned} \tag{2.6.8}$$

Ou seja, o segundo teste nos permitiu ganhar uma confiança muito maior de que nem tudo está bem. Apesar do segundo teste ser consideravelmente menos preciso do que o primeiro, ele ainda melhorou significativamente nossa estimativa.

2.6.3 Expectativa e Variância

Para resumir as principais características das distribuições de probabilidade, precisamos de algumas medidas. A *expectativa* (ou média) da variável aleatória X é denotada como

$$E[X] = \sum_{x} x P(X = x). \tag{2.6.9}$$

Quando a entrada de uma função $f(x)$ é uma variável aleatória retirada da distribuição P com valores diferentes x, a expectativa de $f(x)$ é calculada como

$$E_{x \sim P} [f(x)] = \sum_x f(x) P(x). \tag{2.6.10}$$

Em muitos casos, queremos medir o quanto a variável aleatória X se desvia de sua expectativa. Isso pode ser quantificado pela variação

$$\mathrm{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{2.6.11}$$

Sua raiz quadrada é chamada de *desvio padrão*. A variância de uma função de uma variável aleatória mede pelo quanto a função se desvia da expectativa da função, como diferentes valores x da variável aleatória são amostrados de sua distribuição:

$$\mathrm{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{2.6.12}$$

2.6.4 Sumário

- Podemos obter amostras de distribuições de probabilidade.
- Podemos analisar múltiplas variáveis aleatórias usando distribuição conjunta, distribuição condicional, teorema de Bayes, marginalização e suposições de independência.
- A expectativa e a variância oferecem medidas úteis para resumir as principais características das distribuições de probabilidade.

2.6.5 Exercícios

1. Conduzimos $m = 500$ grupos de experimentos onde cada grupo extrai $n = 10$ amostras. Varie m e n. Observe e analise os resultados experimentais.
2. Dados dois eventos com probabilidade $P(\mathcal{A})$ e $P(\mathcal{B})$, calcule os limites superior e inferior em $P(\mathcal{A} \cup \mathcal{B})$ e $P(\mathcal{A} \cap \mathcal{B})$. (Dica: exiba a situação usando um Diagrama de Venn[27].)
3. Suponha que temos uma sequência de variáveis aleatórias, digamos A, B e C, onde B depende apenas de A e C depende apenas de B, você pode simplificar a probabilidade conjunta $P(A, B, C)$? (Dica: esta é uma Cadeia de Markov[28].)
4. Em Section 2.6.2, o primeiro teste é mais preciso. Por que não executar o primeiro teste duas vezes em vez de executar o primeiro e o segundo testes?

Discussão[29]

2.7 Documentação

Devido a restrições na extensão deste livro, não podemos apresentar todas as funções e classes do PyTorch (e você provavelmente não gostaria que o fizéssemos). A documentação da API e os tutoriais e exemplos adicionais fornecem muita documentação além do livro. Nesta seção, fornecemos algumas orientações para explorar a API PyTorch.

2.7.1 Encontrando Todas as Funções e Classes em um Módulo

Para saber quais funções e classes podem ser chamadas em um módulo, nós invoque a função `dir`. Por exemplo, podemos consultar todas as propriedades no módulo para gerar números aleatórios:

```
import torch

print(dir(torch.distributions))
```

```
['AbsTransform', 'AffineTransform', 'Bernoulli', 'Beta', 'Binomial', 'CatTransform
↪', 'Categorical', 'Cauchy', 'Chi2', 'ComposeTransform', 'ContinuousBernoulli',
↪'CorrCholeskyTransform', 'Dirichlet', 'Distribution', 'ExpTransform', 'Exponential',
↪'ExponentialFamily', 'FisherSnedecor', 'Gamma', 'Geometric', 'Gumbel', 'HalfCauchy',
↪'HalfNormal', 'Independent', 'IndependentTransform', 'Kumaraswamy', 'LKJCholesky', 'Laplace
↪', 'LogNormal', 'LogisticNormal', 'LowRankMultivariateNormal', 'LowerCholeskyTransform
↪', 'MixtureSameFamily', 'Multinomial', 'MultivariateNormal', 'NegativeBinomial',
↪'Normal', 'OneHotCategorical', 'OneHotCategoricalStraightThrough', 'Pareto', 'Poisson
↪', 'PowerTransform', 'RelaxedBernoulli', 'RelaxedOneHotCategorical', 'ReshapeTransform
↪', 'SigmoidTransform', 'SoftmaxTransform', 'StackTransform', 'StickBreakingTransform',
↪'StudentT', 'TanhTransform', 'Transform', 'TransformedDistribution', 'Uniform', 'VonMises
↪', 'Weibull', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader_
↪_', '__name__', '__package__', '__path__', '__spec__', 'bernoulli', 'beta', 'biject_
↪to', 'binomial', 'categorical', 'cauchy', 'chi2', 'constraint_registry', 'constraints
↪', 'continuous_bernoulli', 'dirichlet', 'distribution', 'exp_family', 'exponential',
```

(continues on next page)

[27] https://en.wikipedia.org/wiki/Venn_diagram
[28] https://en.wikipedia.org/wiki/Markov_chain
[29] https://discuss.d2l.ai/t/37

(continued from previous page)

Geralmente, podemos ignorar funções que começam e terminam com __ (objetos especiais em Python) ou funções que começam com um único _ (normalmente funções internas). Com base nos nomes de funções ou atributos restantes, podemos arriscar um palpite de que este módulo oferece vários métodos para gerar números aleatórios, incluindo amostragem da distribuição uniforme (`uniforme`), distribuição normal (`normal`) e distribuição multinomial (`multinomial`).

2.7.2 Buscando o Uso de Funções e Classes Específicas

Para obter instruções mais específicas sobre como usar uma determinada função ou classe, podemos invocar a função `help`. Como um exemplo, vamos explorar as instruções de uso para a função ones dos tensores.

```
help(torch.ones)
```

```
Help on built-in function ones:

ones(...)
    ones(*size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_
↪grad=False) -> Tensor

    Returns a tensor filled with the scalar value 1, with the shape defined
    by the variable argument size.

    Args:
        size (int...): a sequence of integers defining the shape of the output␣
↪tensor.
            Can be a variable number of arguments or a collection like a list or␣
↪tuple.

    Keyword arguments:
        out (Tensor, optional): the output tensor.
        dtype (torch.dtype, optional): the desired data type of returned tensor.
            Default: if None, uses a global default (see torch.set_default_tensor_
↪type()).
        layout (torch.layout, optional): the desired layout of returned Tensor.
            Default: torch.strided.
        device (torch.device, optional): the desired device of returned tensor.
            Default: if None, uses the current device for the default tensor type
            (see torch.set_default_tensor_type()). device will be the CPU
            for CPU tensor types and the current CUDA device for CUDA tensor types.
        requires_grad (bool, optional): If autograd should record operations on the
            returned tensor. Default: False.

    Example::

        >>> torch.ones(2, 3)
        tensor([[ 1.,  1.,  1.],
```

```
        [ 1.,  1.,  1.]])

>>> torch.ones(5)
tensor([ 1.,  1.,  1.,  1.,  1.])
```

A partir da documentação, podemos ver que a função ones cria um novo tensor com a forma especificada e define todos os elementos com o valor de 1. Sempre que possível, você deve executar um teste rápido para confirmar seu interpretação:

```
torch.ones(4)
```

```
tensor([1., 1., 1., 1.])
```

No bloco de notas Jupyter, podemos usar ? para exibir o documento em outra janela. Por exemplo, `list?` criará conteúdo que é quase idêntico a `help(list)`, exibindo-o em um novo navegador janela. Além disso, se usarmos dois pontos de interrogação, como `list??`, o código Python que implementa a função também será exibido.

2.7.3 Sumário

- A documentação oficial fornece muitas descrições e exemplos que vão além deste livro.
- Podemos consultar a documentação para o uso de uma API chamando as funções `dir` e`help`, ou ? E ?? em blocos de notas Jupyter.

2.7.4 Exercícios

1. Procure a documentação de qualquer função ou classe na estrutura de *Deep Learning*. Você também pode encontrar a documentação no site oficial do framework?

Discussions[30]

[30] https://discuss.d2l.ai/t/39

3 | Linear Neural NetworkRedes Neurais Lineares

Antes de entrarmos nos detalhes das redes neurais profundas, precisamos cobrir os fundamentos do treinamento de redes neurais. Neste capítulo, cobriremos todo o processo de treinamento, incluindo a definição de arquiteturas de redes neurais simples, manipulação de dados, especificação de uma função de perda e treinamento do modelo. Para tornar as coisas mais fáceis de entender, começamos com os conceitos mais simples. Felizmente, técnicas clássicas de aprendizagem estatística, como regressão linear e *softmax* podem ser lançadas como redes neurais *lineares*. Partindo desses algoritmos clássicos, apresentaremos o básico, fornecendo a base para técnicas mais complexas no restante do livro.

3.1 Linear Regression

Regression refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable. In the natural sciences and social sciences, the purpose of regression is most often to *characterize* the relationship between the inputs and outputs. Machine learning, on the other hand, is most often concerned with *prediction*.

Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting length of stay (for patients in the hospital), demand forecasting (for retail sales), among countless others. Not every prediction problem is a classic regression problem. In subsequent sections, we will introduce classification problems, where the goal is to predict membership among a set of categories.

3.1.1 Elementos Básicos de Regressão Linear

Regressão linear pode ser a mais simples e mais popular entre as ferramentas padrão para regressão. Datado do início do século 19, A regressão linear flui a partir de algumas suposições simples. Primeiro, assumimos que a relação entre as variáveis independentes $\mathbf{x}$ e a variável dependente y é linear, ou seja, esse y pode ser expresso como uma soma ponderada dos elementos em $\mathbf{x}$, dado algum ruído nas observações. Em segundo lugar, assumimos que qualquer ruído é bem comportado (seguindo uma distribuição gaussiana).

Para motivar a abordagem, vamos começar com um exemplo de execução. Suponha que desejamos estimar os preços das casas (em dólares) com base em sua área (em pés quadrados) e idade (em anos). Para realmente desenvolver um modelo para prever os preços das casas, precisaríamos colocar as mãos em um conjunto de dados consistindo em vendas para as quais sabemos o preço de venda, área e idade de cada casa. Na terminologia de *machine learning*, o conjunto de dados é

chamado de *dataset de treinamento* ou *conjunto de treinamento*, e cada linha (aqui os dados correspondentes a uma venda) é chamado de *exemplo* (ou *tupla*, *instância de dados*, * amostra *). O que estamos tentando prever (preço) é chamado de* label* (ou *rótulo*). As variáveis independentes (idade e área) em que as previsões são baseadas são chamadas de *features* (ou *covariáveis*).

Normalmente, usaremos n para denotar o número de exemplos em nosso conjunto de dados. Nós indexamos os exemplos de dados por i, denotando cada entrada como $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}]^\top$ e o *label* correspondente como $y^{(i)}$.

Modelo Linear

A suposição de linearidade apenas diz que o alvo (preço) pode ser expresso como uma soma ponderada das características (área e idade):

$$\mathrm{preço} = w_{\mathrm{área}} \cdot \mathrm{área} + w_{\mathrm{idade}} \cdot \mathrm{idade} + b. \tag{3.1.1}$$

In (3.1.1), $ w_{**:raw-latex:`\mathrm{area}`**}$ e w_{age} são chamados de *pesos* e b é chamado de *bias* (também chamado de *deslocamento* ou *offset*). Os pesos determinam a influência de cada *feature* em nossa previsão e o *bias* apenas diz qual valor o preço previsto deve assumir quando todos os *features* assumem o valor 0. Mesmo que nunca vejamos nenhuma casa com área zero, ou que têm exatamente zero anos de idade, ainda precisamos do *bias* ou então vamos limitar a expressividade do nosso modelo. Estritamente falando, (3.1.1) é uma *transformação afim* de *features* de entrada, que é caracterizada por uma *transformação linear* de *features* via soma ponderada, combinada com uma *tradução* por meio do *bias* adicionado.

Dado um *dataset*, nosso objetivo é escolher os pesos $\mathbf{w}$ e o *bias* b de modo que, em média, as previsões feitas de acordo com nosso modelo se ajustem o melhor possível aos preços reais observados nos dados. Modelos cuja previsão de saída é determinada pela transformação afim de *features* de entrada são *modelos lineares*, onde a transformação afim é especificada pelos pesos e *bias* escolhidos.

Em disciplinas onde é comum se concentrar em conjuntos de dados com apenas alguns *features*, expressar explicitamente modelos de formato longo como esse é comum. No *machine learning*, geralmente trabalhamos com *datasets* de alta dimensão, portanto, é mais conveniente empregar a notação de álgebra linear. Quando nossas entradas consistem em d *features*, expressamos nossa previsão $\hat{y}$ (em geral, o símbolo "chapéu" ou "acento circunflexo" denota estimativas) como

$$\hat{y} = w_1 x_1 + ... + w_d x_d + b. \tag{3.1.2}$$

Coletando todas as *features* em um vetor $\mathbf{x} \in \mathbb{R}^d$ e todos os pesos em um vetor $\mathbf{w} \in \mathbb{R}^d$, podemos expressar nosso modelo compactamente usando um produto escalar:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{3.1.3}$$

Em (3.1.3), o vetor $\mathbf{x}$ corresponde às *features* de um único exemplo de dados. Frequentemente acharemos conveniente para se referir a recursos de todo o nosso *dataset* de n exemplos através da *matriz de design* $\mathbf{X} \in \mathbb{R}^{n \times d}$. Aqui, $\mathbf{X}$ contém uma linha para cada exemplo e uma coluna para cada *feature*.

Para uma coleção de *features* $\mathbf{X}$, as previsões $\hat{\mathbf{y}} \in \mathbb{R}^n$ pode ser expresso por meio do produto matriz-vetor:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{3.1.4}$$

onde a transmissão (veja Section 2.1.3) é aplicada durante o somatório. Dadas as *features* de um *dataset* de treinamento $\mathbf{X}$ e *labels* correspondentes (conhecidos) $\mathbf{y}$, o objetivo da regressão linear é encontrar o vetor de pesos $\mathbf{w}$ e o termo de polarização b que dadas as *features* de um novo exemplo de dados amostrado da mesma distribuição de $\mathbf{X}$, o *label* do novo exemplo será (na expectativa) previsto com o menor erro.

Mesmo se acreditarmos que o melhor modelo para predizer y dado $\mathbf{x}$ é linear, não esperaríamos encontrar um *dataset* do mundo real de n exemplos onde $y^{(i)}$ é exatamente igual a $\mathbf{w}^\top \mathbf{x}^{(i)} + b$ para todos $1 \leq i \leq n$. Por exemplo, quaisquer instrumentos que usarmos para observar as *features* $\mathbf{X}$ e os *labels* $\mathbf{y}$ podem sofrer uma pequena quantidade de erro de medição. Assim, mesmo quando estamos confiantes que a relação subjacente é linear, vamos incorporar um termo de ruído para contabilizar esses erros.

Antes de começarmos a pesquisar os melhores *parâmetros* (ou *parâmetros do modelo*) $\mathbf{w}$ e b, precisaremos de mais duas coisas: (i) uma medida de qualidade para algum modelo dado; e (ii) um procedimento de atualização do modelo para melhorar sua qualidade.

Função de Perda

Antes de começarmos a pensar sobre como *ajustar* os dados ao nosso modelo, precisamos determinar uma medida de *aptidão*. A *função de perda* quantifica a distância entre o valor *real* e *previsto* do *target*. A perda geralmente será um número não negativo onde valores menores são melhores e previsões perfeitas incorrem em uma perda de 0. A função de perda mais popular em problemas de regressão é o erro quadrático. Quando nossa previsão para um exemplo i é $\hat{y}^{(i)}$ e o *label verdadeiro correspondente é $y^{(i)}$, o quadrado do erro é dado por:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{3.1.5}$$

A constante $\frac{1}{2}$ não faz diferença real mas será notacionalmente conveniente, cancelando quando tomamos a derivada da perda. Como o conjunto de dados de treinamento é fornecido a nós e, portanto, está fora de nosso controle, o erro empírico é apenas função dos parâmetros do modelo. Para tornar as coisas mais concretas, considere o exemplo abaixo onde traçamos um problema de regressão para um caso unidimensional como mostrado em Fig. 3.1.1.

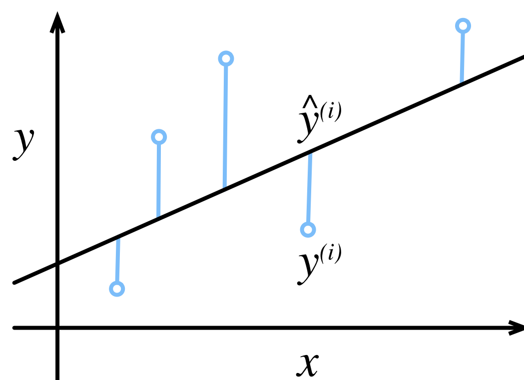


Fig. 3.1.1: Fit data with a linear model.

Observe que grandes diferenças entre estimativas $\hat{y}^{(i)}$ e observações $y^{(i)}$ levam a contribuições ainda maiores para a perda, devido à dependência quadrática. Para medir a qualidade de um

modelo em todo o conjunto de dados de n exemplos, nós simplesmente calculamos a média (ou equivalentemente, somamos) as perdas no conjunto de treinamento.

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.1.6}$$

Ao treinar o modelo, queremos encontrar os parâmetros $(\mathbf{w}^*, b^*)$ que minimizam a perda total em todos os exemplos de treinamento:

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b). \tag{3.1.7}$$

Solução Analítica

A regressão linear passa a ser um problema de otimização incomumente simples. Ao contrário da maioria dos outros modelos que encontraremos neste livro, a regressão linear pode ser resolvida analiticamente aplicando uma fórmula simples. Para começar, podemos incluir o *bias* b no parâmetro $\mathbf{w}$ anexando uma coluna à matriz de design que consiste em todas as unidades. Então nosso problema de previsão é minimizar $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$. Há apenas um ponto crítico na superfície de perda e corresponde ao mínimo de perda em todo o domínio. Tirando a derivada da perda em relação a $\mathbf{w}$ e defini-lo igual a zero produz a solução analítica (de forma fechada):

$$\mathbf{w}^* = (\mathbf X^\top \mathbf X)^{-1}\mathbf X^\top \mathbf{y}. \tag{3.1.8}$$

Embora problemas simples como regressão linear podem admitir soluções analíticas, você não deve se acostumar com essa boa sorte. Embora as soluções analíticas permitam uma boa análise matemática, o requisito de uma solução analítica é tão restritivo que isso excluiria todo o *deep learning*.

Gradiente Descendente Estocástico com *Minibatch*

Mesmo nos casos em que não podemos resolver os modelos analiticamente, acontece que ainda podemos treinar modelos efetivamente na prática. Além disso, para muitas tarefas, aqueles modelos difíceis de otimizar acabam sendo muito melhores do que descobrir como treiná-los acaba valendo a pena.

A principal técnica para otimizar quase qualquer modelo de *deep learning*, e que recorreremos ao longo deste livro, consiste em reduzir iterativamente o erro atualizando os parâmetros na direção que diminui gradativamente a função de perda. Este algoritmo é denominado *gradiente descendente*.

A aplicação mais ingênua de gradiente descendente consiste em obter a derivada da função de perda, que é uma média das perdas calculadas em cada exemplo no *dataset*. Na prática, isso pode ser extremamente lento: devemos passar por todo o conjunto de dados antes de fazer uma única atualização. Assim, frequentemente nos contentaremos em amostrar um *minibatch* aleatório de exemplos toda vez que precisamos calcular a atualização, uma variante chamada *gradiente descendente estocástico de minibatch*.

Em cada iteração, primeiro amostramos aleatoriamente um *minibatch* $\mathcal{B}$ consistindo em um número fixo de exemplos de treinamento. Em seguida, calculamos a derivada (gradiente) da perda média no *minibatch* em relação aos parâmetros do modelo. Finalmente, multiplicamos o gradiente por um valor positivo predeterminado η e subtraimos o termo resultante dos valores dos parâmetros atuais.

Podemos expressar a atualização matematicamente da seguinte forma (∂ denota a derivada parcial):

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b). \tag{3.1.9}$$

Para resumir, as etapas do algoritmo são as seguintes: (i) inicializamos os valores dos parâmetros do modelo, normalmente de forma aleatória; (ii) amostramos iterativamente *minibatches* aleatórios dos dados, atualizando os parâmetros na direção do gradiente negativo. Para perdas quadráticas e transformações afins, podemos escrever isso explicitamente da seguinte maneira:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right),\\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \end{aligned} \tag{3.1.10}$$

Observe que $\mathbf{w}$ e $\mathbf{x}$ são vetores em (3.1.10). Aqui, a notação vetorial mais elegante torna a matemática muito mais legível do que expressar coisas em termos de coeficientes, diga $w_1, w_2, \ldots, w_d$. A cardinalidade definida $|\mathcal{B}|$ representa o número de exemplos em cada *minibatch* (o *tamanho do lote*) e η denota a *taxa de aprendizagem*. Enfatizamos que os valores do tamanho do lote e da taxa de aprendizagem são pré-especificados manualmente e normalmente não aprendidos por meio do treinamento do modelo. Esses parâmetros são ajustáveis, mas não atualizados no loop de treinamento são chamados de *hiperparâmetros*. *Ajuste de hiperparâmetros* é o processo pelo qual os hiperparâmetros são escolhidos, e normalmente requer que os ajustemos com base nos resultados do ciclo de treinamento conforme avaliado em um *dataset de validação* separado (ou *conjunto de validação*).

Após o treinamento para algum número predeterminado de iterações (ou até que algum outro critério de parada seja atendido), registramos os parâmetros estimados do modelo, denotado $\hat{\mathbf{w}}, \hat{b}$. Observe que mesmo que nossa função seja verdadeiramente linear e sem ruídos, esses parâmetros não serão os minimizadores exatos da perda porque, embora o algoritmo convirja lentamente para os minimizadores, não pode alcançá-los exatamente em um número finito de etapas.

A regressão linear passa a ser um problema de aprendizagem onde há apenas um mínimo em todo o domínio. No entanto, para modelos mais complicados, como redes profundas, as superfícies de perda contêm muitos mínimos. Felizmente, por razões que ainda não são totalmente compreendidas, praticantes de *deep learning* raramente se esforçam para encontrar parâmetros que minimizem a perda *em conjuntos de treinamento*. A tarefa mais formidável é encontrar parâmetros que irão atingir baixa perda de dados que não vimos antes, um desafio chamado *generalização*. Retornamos a esses tópicos ao longo do livro.

Fazendo Predições com o Modelo Aprendido

Dado o modelo de regressão linear aprendido $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, agora podemos estimar o preço de uma nova casa (não contido nos dados de treinamento) dada sua área x_1 e idade x_2. Estimar *labels* dadas as características é comumente chamado de *predição* ou *inferência*.

Tentaremos manter o termo *predição* porque chamando esta etapa de *inferência*, apesar de emergir como jargão padrão no *deep learning*, é um nome impróprio. Em estatísticas, *inferência* denota mais frequentemente estimar parâmetros com base em um conjunto de dados. Este uso indevido de terminologia é uma fonte comum de confusão quando os profissionais de *machine learning* conversam com os estatísticos.

3.1.2 Vetorização para Velocidade

Ao treinar nossos modelos, normalmente queremos processar *minibatches* inteiros de exemplos simultaneamente. Fazer isso de forma eficiente requer que nós vetorizar os cálculos e aproveitar as bibliotecas de álgebra linear rápida em vez de escrever *loops for* custosos em Python.

```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

Para ilustrar por que isso é tão importante, podemos considerer dois métodos para adicionar vetores. Para começar, instanciamos dois vetores de 10000 dimensões contendo todos os outros. Em um método, faremos um loop sobre os vetores com um *loop for* Python. No outro método, contaremos com uma única chamada para +.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

Uma vez que iremos comparar o tempo de execução com freqüência neste livro, vamos definir um cronômetro.

```
class Timer:  #@save
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """Start the timer."""
        self.tik = time.time()

    def stop(self):
        """Stop the timer and record the time in a list."""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """Return the average time."""
        return sum(self.times) / len(self.times)

    def sum(self):
        """Return the sum of time."""
        return sum(self.times)

    def cumsum(self):
        """Return the accumulated time."""
        return np.array(self.times).cumsum().tolist()
```

Agora podemos avaliar as cargas de trabalho. Primeiro, nós os adicionamos, uma coordenada por vez, usando um *loop for*.

```
c = torch.zeros(n)
timer = Timer()
for i in range(n):
    c[i] = a[i] + b[i]
f'{timer.stop():.5f} sec'
```

```
'0.09436 sec'
```

Alternativamente, contamos com o operador recarregado + para calcular a soma elemento a elemento.

```
timer.start()
d = a + b
f'{timer.stop():.5f} sec'
```

```
'0.00022 sec'
```

Você provavelmente percebeu que o segundo método é dramaticamente mais rápido que o primeiro. A vetorização do código geralmente produz acelerações da ordem de magnitude. Além disso, colocamos mais matemática na biblioteca e não precisamos escrever tantos cálculos nós mesmos, reduzindo o potencial de erros.

3.1.3 A Distribuição Normal e Perda Quadrada

Embora você já possa sujar as mãos usando apenas as informações acima, a seguir, podemos motivar mais formalmente o objetivo de perda quadrado através de suposições sobre a distribuição do ruído.

A regressão linear foi inventada por Gauss em 1795, que também descobriu a distribuição normal (também chamada de *Gaussiana*). Acontece que a conexão entre a distribuição normal e regressão linear é mais profunda do que o parentesco comum. Para refrescar sua memória, a densidade de probabilidade de uma distribuição normal com média μ e variância σ^2 (desvio padrão σ) é dada como

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.1.11}$$

Abaixo definimos uma função Python para calcular a distribuição normal.

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 / sigma**2 * (x - mu)**2)
```

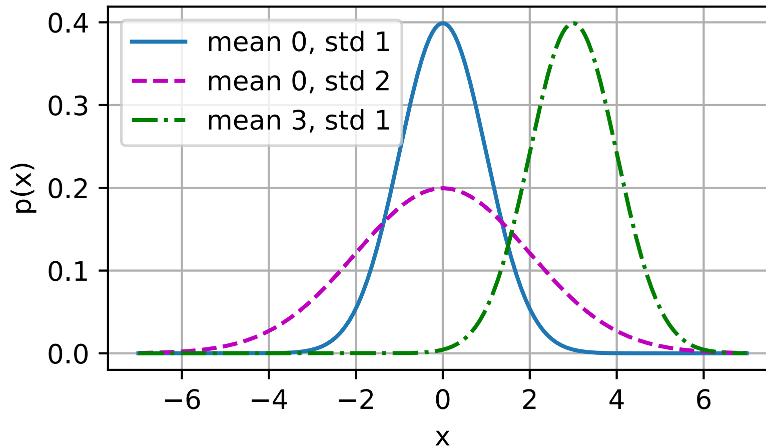
Podemos agora visualizar as distribuições normais.

```
# Use numpy again for visualization
x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
```

(continues on next page)

(continued from previous page)

```
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



Como podemos ver, mudar a média corresponde a uma mudança ao longo do eixo x, e aumentar a variância espalha a distribuição, diminuindo seu pico.

Uma maneira de motivar a regressão linear com a função de perda de erro quadrático médio (ou simplesmente perda quadrada) é assumir formalmente que as observações surgem de observações ruidosas, onde o ruído é normalmente distribuído da seguinte forma:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ onde } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.1.12}$$

Assim, podemos agora escrever a *probabilidade* de ver um determinado y para um determinado $\mathbf{x}$ via

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{3.1.13}$$

Agora, de acordo com o princípio da máxima verossimilhança (*likelihood*), os melhores valores dos parâmetros $\mathbf{w}$ e b são os que maximizam a *probabilidade* de todo o conjunto de dados:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{3.1.14}$$

Estimadores escolhidos de acordo com o princípio da máxima verossimilhança são chamados de *estimadores de máxima verossimilhança*. Enquanto, maximizando o produto de muitas funções exponenciais, pode parecer difícil, podemos simplificar as coisas significativamente, sem alterar o objetivo, maximizando o log da probabilidade em vez disso. Por razões históricas, as otimizações são expressas com mais frequência como minimização em vez de maximização. Portanto, sem alterar nada, podemos minimizar a *probabilidade de log negativo* $-\log P(\mathbf{y} \mid \mathbf{X})$. Trabalhando a matemática nos dá:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{3.1.15}$$

Agora, precisamos apenas mais uma suposição de que σ é alguma constante fixa. Assim, podemos ignorar o primeiro termo porque não depende de $\mathbf{w}$ ou b. Agora, o segundo termo é idêntico à perda de erro quadrada introduzida anteriormente, exceto para a constante multiplicativa

$\frac{1}{\sigma^2}$. Felizmente, a solução não depende de σ. Segue-se que minimizar o erro quadrático médio é equivalente a estimar a máxima verossimilhança de um modelo linear sob a suposição de ruído gaussiano aditivo.

3.1.4 Da Regressão Linear às Redes Profundas

Até agora, falamos apenas sobre modelos lineares. Enquanto as redes neurais cobrem uma família muito mais rica de modelos, podemos começar a pensar no modelo linear como uma rede neural, expressando-a na linguagem das redes neurais. Para começar, vamos começar reescrevendo as coisas em uma notação de "camada".

Diagrama de Rede Neural

Praticantes de *deep learning* gostam de desenhar diagramas para visualizar o que está acontecendo em seus modelos. Em Fig. 3.1.2, retratamos nosso modelo de regressão linear como uma rede neural. Observe que esses diagramas destacam o padrão de conectividade como cada entrada é conectada à saída, mas não os valores tomados pelos pesos ou *bias*.

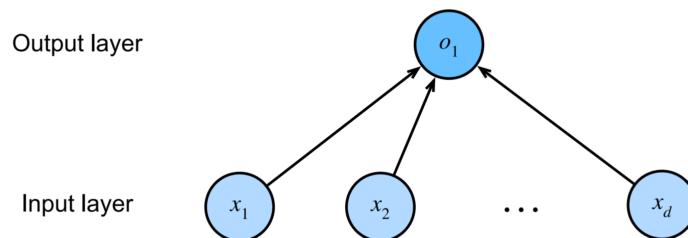


Fig. 3.1.2: Linear regression is a single-layer neural network.

Para a rede neural mostrada em Fig. 3.1.2, as entradas são $x_1, \ldots, x_d$, portanto, o *número de entradas* (ou *dimensionalidade do recurso*) na camada de entrada é d. A saída da rede em Fig. 3.1.2 é o_1, portanto, o *número de saídas* na camada de saída é 1. Observe que os valores de entrada são todos *fornecidos* e há apenas um único neurônio *calculado*. Concentrando-se em onde a computação ocorre, convencionalmente, não consideramos a camada de entrada ao contar camadas. Quer dizer, o *número de camadas* para a rede neural em Fig. 3.1.2 é 1. Podemos pensar em modelos de regressão linear como redes neurais consistindo em apenas um único neurônio artificial, ou como redes neurais de camada única.

Já que para a regressão linear, cada entrada é conectada para cada saída (neste caso, há apenas uma saída), podemos considerar esta transformação (a camada de saída em Fig. 3.1.2) como uma *camada totalmente conectada* ou *camada densa*. Falaremos muito mais sobre redes compostas por tais camadas no próximo capítulo.

Biologia

Como a regressão linear (inventada em 1795) antecede a neurociência computacional, pode parecer anacrônico descrever regressão linear como uma rede neural. Para ver por que os modelos lineares eram um lugar natural para começar quando os ciberneticistas/neurofisiologistas Warren McCulloch e Walter Pitts começaram a desenvolver modelos de neurônios artificiais, considere a imagem de desenho animado de um neurônio biológico em Fig. 3.1.3, consistindo em *dendritos* (terminais de entrada), o *núcleo* (CPU), o *axônio* (fio de saída), e os * terminais de axônio* (terminais de saída), permitindo conexões com outros neurônios por meio de *sinapses*.

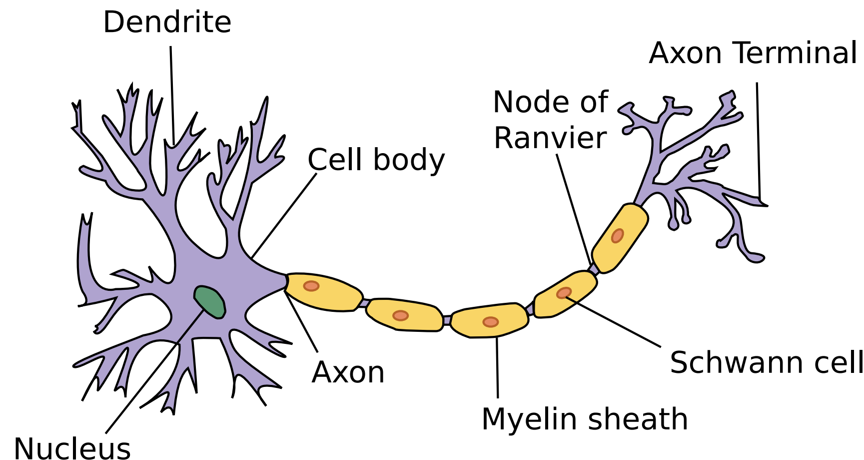


Fig. 3.1.3: The real neuron.

Informação x_i vinda de outros neurônios (ou sensores ambientais, como a retina) é recebida nos dendritos. Em particular, essa informação é ponderada por *pesos sinápticos* w_i determinando o efeito das entradas (por exemplo, ativação ou inibição por meio do produto $x_i w_i$). As entradas ponderadas que chegam de várias fontes são agregadas no núcleo como uma soma ponderada $y = \sum_i x_i w_i + b$, e esta informação é então enviada para processamento adicional no axônio y, normalmente após algum processamento não linear via $\sigma(y)$. De lá, ela chega ao seu destino (por exemplo, um músculo) ou é alimentado em outro neurônio por meio de seus dendritos.

Certamente, a ideia de alto nível de que muitas dessas unidades poderiam ser remendadas com a conectividade certa e algoritmo de aprendizado correto, para produzir um comportamento muito mais interessante e complexo do que qualquer neurônio sozinho poderia expressar, se deve ao nosso estudo de sistemas neurais biológicos reais.

Ao mesmo tempo, a maioria das pesquisas em *deep learning* hoje atrai pouca inspiração direta na neurociência. Invocamos Stuart Russell e Peter Norvig que, em seu livro clássico de IA *Inteligência Artificial: Uma Abordagem Moderna* (Russell & Norvig, 2016), apontoam que, embora os aviões possam ter sido *inspirados* por pássaros, ornitologia não tem sido o principal motivador de inovação aeronáutica por alguns séculos. Da mesma forma, a inspiração no *deep learning* nos dias de hoje vem em igual ou maior medida da matemática, estatísticas e ciência da computação.

Summary

- Os principais ingredientes em um modelo de *machine learning* são dados de treinamento, uma função de perda, um algoritmo de otimização e, obviamente, o próprio modelo.
- A vetorização torna tudo melhor (principalmente matemática) e mais rápido (principalmente código).
- Minimizar uma função objetivo e realizar a estimativa de máxima verossimilhança pode significar a mesma coisa.
- Os modelos de regressão linear também são redes neurais.

Exercises

1. Suponha que temos alguns dados $x_1, \ldots, x_n \in \mathbb{R}$. Nosso objetivo é encontrar uma constante b tal que $ **:raw-latex:`\sum`**_i(x_i - b)^2$ seja minimizado.
    1. Encontre uma solução analítica para o valor ideal de b.
    2. Como esse problema e sua solução se relacionam com a distribuição normal?
2. Derive a solução analítica para o problema de otimização para regressão linear com erro quadrático. Para manter as coisas simples, você pode omitir o *bias* b do problema (podemos fazer isso com base em princípios, adicionando uma coluna a $\mathbf{X}$ consistindo em todas as colunas).
    1. Escreva o problema de otimização em notação de matriz e vetor (trate todos os dados como uma única matriz e todos os valores de *label* esperados como um único vetor).
    2. Calcule o gradiente da perda em relação a w.
    3. Encontre a solução analítica definindo o gradiente igual a zero e resolvendo a equação da matriz.
    4. Quando isso pode ser melhor do que usar o gradiente descendente estocástico? Quando esse método pode falhar?
3. Suponha que o modelo de ruído que governa o ruído aditivo ϵ é a distribuição exponencial. Ou seja, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
    1. Escreva a *log-likelihood* negativa dos dados no modelo $-\log P(\mathbf{y} \mid \mathbf{X})$.
    2. Você pode encontrar uma solução de forma fechada?
    3. Sugira um algoritmo de gradiente descendente estocástico para resolver este problema. O que pode dar errado (dica: o que acontece perto do ponto estacionário à medida que atualizamos os parâmetros)? Você pode consertar isso?

Discussions[31]

[31] https://discuss.d2l.ai/t/258

3.2 Linear Regression Implementation from Scratch

Agora que você entende as principais ideias por trás da regressão linear, podemos começar a trabalhar por meio de uma implementação prática no código. Nesta seção, vamos implementar todo o método do zero, incluindo o pipeline de dados, o modelo, a função de perda e o otimizador de descida gradiente estocástico do minibatch. Embora as estruturas modernas de *deep learning* possam automatizar quase todo esse trabalho, implementar coisas do zero é a única maneira para ter certeza de que você realmente sabe o que está fazendo. Além disso, quando chega a hora de personalizar modelos, definindo nossas próprias camadas ou funções de perda, entender como as coisas funcionam nos bastidores será útil. Nesta seção, contaremos apenas com tensores e diferenciação automática. Posteriormente, apresentaremos uma implementação mais concisa, aproveitando sinos e assobios de *frameworks* de *deep learning*.

```
%matplotlib inline
import random
import torch
from d2l import torch as d2l
```

3.2.1 Gerando o Dataset

Para manter as coisas simples, iremos construir um conjunto de dados artificial de acordo com um modelo linear com ruído aditivo. Nossa tarefa será recuperar os parâmetros deste modelo usando o conjunto finito de exemplos contidos em nosso conjunto de dados. Manteremos os dados em baixa dimensão para que possamos visualizá-los facilmente. No seguinte *snippet* de código, geramos um conjunto de dados contendo 1000 exemplos, cada um consistindo em 2 *features* amostrado a partir de uma distribuição normal padrão. Assim, nosso conjunto de dados sintético será uma matriz $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

Os verdadeiros parâmetros que geram nosso conjunto de dados serão $\mathbf{w} = [2, -3, 4]^\top$ e $b = 4, 2$, e nossos rótulos sintéticos serão atribuídos de acordo ao seguinte modelo linear com o termo de ruído ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{3.2.1}$$

Você pode pensar em ϵ como um potencial de captura erros de medição nos recursos e rótulos. Vamos assumir que as premissas padrão são válidas e, portanto, que ϵ obedece a uma distribuição normal com média 0. Para tornar nosso problema mais fácil, definiremos seu desvio padrão em 0,01. O código a seguir gera nosso conjunto de dados sintético.

```
def synthetic_data(w, b, num_examples):  #@save
    """Generate y = Xw + b + noise."""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))
```

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

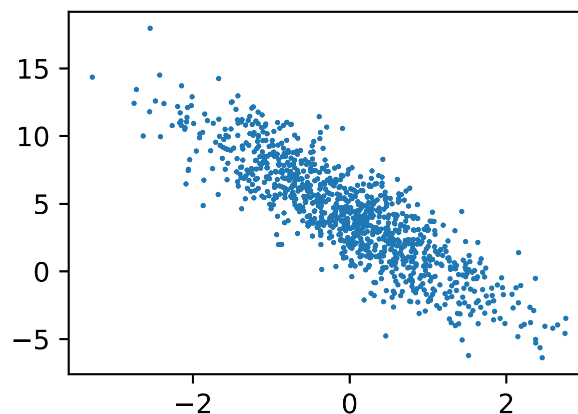
Observe que cada linha em `features` consiste em um exemplo de dados bidimensionais e que cada linha em `labels` consiste em um valor de rótulo unidimensional (um escalar).

```
print('features:', features[0],'\nlabel:', labels[0])
```

```
features: tensor([-0.9563,  1.2447])
label: tensor([-1.9621])
```

Ao gerar um gráfico de dispersão usando o segundo recurso `features [:, 1]` e `labels`, podemos observar claramente a correlação linear entre os dois.

```
d2l.set_figsize()
# The semicolon is for displaying the plot only
d2l.plt.scatter(features[:, (1)].detach().numpy(), labels.detach().numpy(), 1);
```



3.2.2 Lendo o *Dataset*

Lembre-se de que os modelos de treinamento consistem em fazer várias passagens sobre o *dataset*, pegando um *minibatch* de exemplos por vez, e usando-los para atualizar nosso modelo. Uma vez que este processo é tão fundamental para treinar algoritmos de a*machine learning*, vale a pena definir uma função de utilidade para embaralhar o conjunto de dados e acessá-lo em *minibatches*.

No código a seguir, nós definimos a função `data_iter` para demonstrar uma possível implementação dessa funcionalidade. A função leva um tamanho de amostra, uma matriz de *features*, e um vetor de *labels*, produzindo *minibatches* do tamanho `batch_size`. Cada *minibatch* consiste em uma tupla de *features* e *labels*.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

Em geral, queremos usar *minibatches* de tamanhos razoáveis para aproveitar as vantagens do hardware da GPU, que se destaca em operações de paralelização. Porque cada exemplo pode ser alimentado por meio de nossos modelos em paralelo e o gradiente da função de perda para cada exemplo também pode ser tomado em paralelo, GPUs nos permitem processar centenas de exemplos em pouco mais tempo do que pode demorar para processar apenas um único exemplo.

Para construir alguma intuição, vamos ler e imprimir o primeiro pequeno lote de exemplos de dados. A forma dos recursos em cada *minibatch* nos diz o tamanho do *minibatch* e o número de recursos de entrada. Da mesma forma, nosso *minibatch* de rótulos terá uma forma dada por `batch_size`.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
tensor([[-0.0649,  0.4390],
        [-0.2518, -0.4019],
        [-0.1489,  0.2960],
        [ 1.6701, -0.8914],
        [ 0.6946,  0.2719],
        [-1.4623,  0.5890],
        [ 0.1270,  0.7019],
        [-1.2410,  0.1549],
        [-0.3620, -0.1373],
        [-0.2483, -1.6446]])
 tensor([[ 2.5737],
        [ 5.0419],
        [ 2.8981],
        [10.5841],
        [ 4.6693],
        [-0.7264],
        [ 2.0609],
        [ 1.1768],
        [ 3.9539],
        [ 9.2774]])
```

Conforme executamos a iteração, obtemos *minibatches* distintos sucessivamente até que todo o conjunto de dados se esgote (tente isto). Embora a iteração implementada acima seja boa para fins didáticos, é ineficiente de maneiras que podem nos colocar em apuros em problemas reais. Por exemplo, requer que carreguemos todos os dados na memória e que realizamos muitos acessos aleatórios à memória. Os iteradores integrados implementados em uma estrutura de *deep learning* são consideravelmente mais eficientes e podem lidar com dados armazenados em arquivos e dados alimentados por meio de fluxos de dados.

3.2.3 Initializing Model Parameters

Antes de começarmos a otimizar os parâmetros do nosso modelo por gradiente descendente estocástico de *minibatch*, precisamos ter alguns parâmetros em primeiro lugar. No código a seguir, inicializamos os pesos por amostragem números aleatórios de uma distribuição normal com média 0 e um desvio padrão de 0,01, e definindo a tendência para 0.

```
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

Depois de inicializar nossos parâmetros, nossa próxima tarefa é atualizá-los até eles se ajustam aos nossos dados suficientemente bem. Cada atualização requer a obtenção do gradiente da nossa função de perda no que diz respeito aos parâmetros. Dado este gradiente, podemos atualizar cada parâmetro na direção que pode reduzir a perda.

Uma vez que ninguém quer calcular gradientes explicitamente (isso é entediante e sujeito a erros), usamos diferenciação automática, conforme apresentado em Section 2.5, para calcular o gradiente.

3.2.4 Definindo o Modelo

Em seguida, devemos definir nosso modelo, relacionando suas entradas e parâmetros com suas saídas. Lembre-se de que, para calcular a saída do modelo linear, simplesmente pegamos o produto escalar vetor-matriz dos recursos de entrada $\mathbf{X}$ e os pesos do modelo $\mathbf{w}$, e adicione o *offset* b a cada exemplo. Observe que abaixo de $\mathbf{Xw}$ está um vetor e b é um escalar. Lembre-se do mecanismo de transmissão conforme descrito em Section 2.1.3. Quando adicionamos um vetor e um escalar, o escalar é adicionado a cada componente do vetor.

```
def linreg(X, w, b):  #@save
    """The linear regression model."""
    return torch.matmul(X, w) + b
```

3.2.5 Definindo a Função de Perda

Uma vez que atualizar nosso modelo requer tomar o gradiente de nossa função de perda, devemos definir a função de perda primeiro. Aqui vamos usar a função de perda quadrada conforme descrito em Section 3.1. Na implementação, precisamos transformar o valor verdadeiro `y` na forma do valor previsto `y_hat`. O resultado retornado pela seguinte função também terá a mesma forma de `y_hat`.

```
def squared_loss(y_hat, y):  #@save
    """Squared loss."""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

3.2.6 Definindo o Algoritmo de Otimização

Como discutimos em Section 3.1, a regressão linear tem uma solução de forma fechada. No entanto, este não é um livro sobre regressão linear: é um livro sobre *deep learning*. Uma vez que nenhum dos outros modelos que este livro apresenta pode ser resolvido analiticamente, aproveitaremos esta oportunidade para apresentar seu primeiro exemplo de trabalho de gradiente descendente estocástico de *minibatch*.

Em cada etapa, usando um *minibatch* retirado aleatoriamente de nosso conjunto de dados, vamos estimar o gradiente da perda em relação aos nossos parâmetros. A seguir, vamos atualizar nossos parâmetros na direção que pode reduzir a perda. O código a seguir aplica a atualização da descida gradiente estocástica do *minibatch*, dado um conjunto de parâmetros, uma taxa de aprendizagem e um tamanho de *batch*. O tamanho da etapa de atualização é determinado pela taxa de aprendizagem `lr`. Como nossa perda é calculada como a soma do *minibatch* de exemplos, normalizamos o tamanho do nosso passo pelo tamanho do *batch* (`batch_size`), de modo que a magnitude de um tamanho de passo típico não dependa muito de nossa escolha do tamanho do lote.

```
def sgd(params, lr, batch_size):  #@save
    """Minibatch stochastic gradient descent."""
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

3.2.7 Treinamento

Agora que temos todas as peças no lugar, estamos prontos para implementar o *loop* de treinamento principal. É crucial que você entenda este código porque você verá loops de treinamento quase idênticos repetidamente ao longo de sua carreira de *deep learning*.

Em cada iteração, pegaremos um *minibatch* de exemplos de treinamento, e os passamos por nosso modelo para obter um conjunto de previsões. Depois de calcular a perda, iniciamos a passagem para trás pela rede, armazenando os gradientes em relação a cada parâmetro. Finalmente, chamaremos o algoritmo de otimização de `sgd` para atualizar os parâmetros do modelo.

Em resumo, vamos executar o seguinte loop:

- Inicializar parâmetros $(\mathbf{w}, b)$
- Repetir até terminar
  - Computar gradiente $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - Atualizar parâmetros $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Em cada *época*, iremos iterar por todo o conjunto de dados (usando a função `data_iter`) uma vez passando por todos os exemplos no conjunto de dados de treinamento (assumindo que o número de exemplos seja divisível pelo tamanho do lote). O número de épocas `num_epochs` e a taxa de aprendizagem`lr` são hiperparâmetros, que definimos aqui como 3 e 0,03, respectivamente. Infelizmente, definir hiperparâmetros é complicado e requer alguns ajustes por tentativa e erro. Excluímos esses detalhes por enquanto, mas os revisamos mais tarde em Chapter 11.

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Compute gradient on `l` with respect to [`w`, `b`]
        l.sum().backward()
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.044391
epoch 2, loss 0.000181
epoch 3, loss 0.000046
```

Neste caso, porque nós mesmos sintetizamos o conjunto de dados, sabemos exatamente quais são os verdadeiros parâmetros. Assim, podemos avaliar nosso sucesso no treinamento comparando os parâmetros verdadeiros com aqueles que aprendemos através de nosso ciclo de treinamento. Na verdade, eles acabam sendo muito próximos um do outro.

```
print(f'error in estimating w: {true_w - w.reshape(true_w.shape)}')
print(f'error in estimating b: {true_b - b}')
```

```
error in estimating w: tensor([ 0.0006, -0.0003], grad_fn=<SubBackward0>)
error in estimating b: tensor([0.0008], grad_fn=<RsubBackward1>)
```

Observe que não devemos tomar isso como garantido que somos capazes de recuperar os parâmetros perfeitamente. No entanto, no *machine learning*, normalmente estamos menos preocupados com a recuperação de verdadeiros parâmetros subjacentes, e mais preocupados com parâmetros que levam a previsões altamente precisas. Felizmente, mesmo em problemas de otimização difíceis, o gradiente descendente estocástico pode muitas vezes encontrar soluções notavelmente boas, devido em parte ao fato de que, para redes profundas, existem muitas configurações dos parâmetros que levam a uma previsão altamente precisa.

3.2.8 Resumo

- Vimos como uma rede profunda pode ser implementada e otimizada do zero, usando apenas tensores e diferenciação automática, sem a necessidade de definir camadas ou otimizadores sofisticados.
- Esta seção apenas arranha a superfície do que é possível. Nas seções a seguir, descreveremos modelos adicionais com base nos conceitos que acabamos de apresentar e aprenderemos como implementá-los de forma mais concisa.

3.2.9 Exercícios

1. O que aconteceria se inicializássemos os pesos para zero. O algoritmo ainda funcionaria?
2. Suponha que você seja Georg Simon Ohm[32] tentando inventar um modelo entre tensão e corrente. Você poderia usar a diferenciação automática para aprender os parâmetros do seu modelo?
3. Você pode usar a Lei de Planck[33] para determinar a temperatura de um objeto usando densidade de energia espectral?
4. Quais são os problemas que você pode encontrar se quiser calcular as derivadas secundárias? Como você os consertaria?
5. Por que a função `reshape` é necessária na função`squared_loss`?
6. Experimente usar diferentes taxas de aprendizagem para descobrir a rapidez com que o valor da função de perda diminui.
7. Se o número de exemplos não pode ser dividido pelo tamanho do lote, o que acontece com o comportamento da função `data_iter`?

Discussions[34]

3.3 Implementação Concisa de Regressão Linear

Amplo e intenso interesse em *deep learning* nos últimos anos inspiraram empresas, acadêmicos e amadores para desenvolver uma variedade de estruturas de código aberto maduras para automatizar o trabalho repetitivo de implementação algoritmos de aprendizagem baseados em gradiente. Em Section 3.2, contamos apenas com (i) tensores para armazenamento de dados e álgebra linear; e (ii) auto diferenciação para cálculo de gradientes. Na prática, porque iteradores de dados, funções de perda, otimizadores, e camadas de rede neural são tão comuns que as bibliotecas modernas também implementam esses componentes para nós.

Nesta seção, mostraremos como implementar o modelo de regressão linear de:numref:`sec_linear_scratch` de forma concisa, usando APIs de alto nível de estruturas de *deep learning*.

3.3.1 Gerando the Dataset

Para começar, vamos gerar o mesmo conjunto de dados como em Section 3.2.

```
import numpy as np
import torch
from torch.utils import data
from d2l import torch as d2l
```

[32] https://en.wikipedia.org/wiki/Georg_Ohm
[33] https://en.wikipedia.org/wiki/Planck%27s_law
[34] https://discuss.d2l.ai/t/43

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

3.3.2 Lendo o Dataset

Em vez de usar nosso próprio iterador, podemos chamar a API existente em uma estrutura para ler os dados. Passamos ``*features*`` e ``*labels*`` como argumentos e especificamos ``*batch_size*`` ao instanciar um objeto iterador de dados. Além disso, o valor booleano `is_train` indica se ou não queremos que o objeto iterador de dados embaralhe os dados em cada época (passe pelo conjunto de dados).

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Construct a PyTorch data iterator."""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Now we can use `data_iter` in much the same way as we called the `data_iter` function in Section 3.2. To verify that it is working, we can read and print the first minibatch of examples. Comparing with Section 3.2, here we use `iter` to construct a Python iterator and use `next` to obtain the first item from the iterator.

```
next(iter(data_iter))
```

```
[tensor([[ 0.4291,  0.1270],
         [ 1.7995, -1.2012],
         [ 0.9239, -0.7505],
         [ 1.3561, -0.4303],
         [ 0.6144, -0.5138],
         [-1.0876, -0.8626],
         [ 1.1090,  3.4219],
         [-1.6905,  0.1326],
         [ 0.6009,  0.9365],
         [ 0.1519, -1.1885]]),
 tensor([[ 4.6351],
         [11.8713],
         [ 8.5979],
         [ 8.3702],
         [ 7.1819],
         [ 4.9432],
         [-5.2131],
         [ 0.3592],
         [ 2.2248],
         [ 8.5418]])]
```

3.3.3 Definindo o Modelo

Quando implementamos a regressão linear do zero em Section 3.2, definimos nossos parâmetros de modelo explicitamente e codificamos os cálculos para produzir saída usando operações básicas de álgebra linear. Você *deveria* saber como fazer isso. Mas quando seus modelos ficam mais complexos, e uma vez que você tem que fazer isso quase todos os dias, você ficará feliz com a ajuda. A situação é semelhante a codificar seu próprio blog do zero. Fazer uma ou duas vezes é gratificante e instrutivo, mas você seria um péssimo desenvolvedor da web se toda vez que você precisava de um blog você passava um mês reinventando tudo.

Para operações padrão, podemos usar as camadas predefinidas de uma estrutura, o que nos permite focar especialmente nas camadas usadas para construir o modelo em vez de ter que se concentrar na implementação. Vamos primeiro definir uma variável de modelo `net`, que se refere a uma instância da classe `Sequential`. A classe `Sequential` define um contêiner para várias camadas que serão encadeadas. Dados dados de entrada, uma instância `Sequential` passa por a primeira camada, por sua vez passando a saída como entrada da segunda camada e assim por diante. No exemplo a seguir, nosso modelo consiste em apenas uma camada, portanto, não precisamos realmente de `Sequencial`. Mas como quase todos os nossos modelos futuros envolverão várias camadas, vamos usá-lo de qualquer maneira apenas para familiarizá-lo com o fluxo de trabalho mais padrão.

Lembre-se da arquitetura de uma rede de camada única, conforme mostrado em Fig. 3.1.2. Diz-se que a camada está *totalmente conectada* porque cada uma de suas entradas está conectada a cada uma de suas saídas por meio de uma multiplicação de matriz-vetor.

: begin_tab: `pytorch` No PyTorch, a camada totalmente conectada é definida na classe `Linear`. Observe que passamos dois argumentos para `nn.Linear`. O primeiro especifica a dimensão do recurso de entrada, que é 2, e o segundo é a dimensão do recurso de saída, que é um escalar único e, portanto, 1.

```
# `nn` is an abbreviation for neural networks
from torch import nn

net = nn.Sequential(nn.Linear(2, 1))
```

3.3.4 Inicializando os Parâmetros do Modelo

Antes de usar `net`, precisamos inicializar os parâmetros do modelo, como os pesos e *bias* no modelo de regressão linear. As estruturas de *deep learning* geralmente têm uma maneira predefinida de inicializar os parâmetros. Aqui especificamos que cada parâmetro de peso deve ser amostrado aleatoriamente a partir de uma distribuição normal com média 0 e desvio padrão 0,01. O parâmetro bias será inicializado em zero.

As we have specified the input and output dimensions when constructing `nn.Linear`. Now we access the parameters directly to specify their initial values. We first locate the layer by `net[0]`, which is the first layer in the network, and then use the `weight.data` and `bias.data` methods to access the parameters. Next we use the replace methods `normal_` and `fill_` to overwrite parameter values.

```
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
```

```
tensor([0.])
```

3.3.5 Definindo a Função de Perda

A classe MSELoss calcula o erro quadrático médio, também conhecido como norma $ L_2 $ quadrada. Por padrão, ela retorna a perda média sobre os exemplos.

```
loss = nn.MSELoss()
```

3.3.6 Definindo o Algoritmo de Otimização

O gradiente descendente estocástico de *minibatch* é uma ferramenta padrão para otimizar redes neurais e, portanto, PyTorch o suporta ao lado de uma série de variações deste algoritmo no módulo `optim`. Quando nós instanciamos uma instância `SGD`, iremos especificar os parâmetros para otimizar (podem ser obtidos de nossa rede via `net.parameters ()`), com um dicionário de hiperparâmetros exigido por nosso algoritmo de otimização. O gradiente descendente estocástico de *minibatch* requer apenas que definamos o valor `lr`, que é definido como 0,03 aqui.

```
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

3.3.7 Treinamento

Você deve ter notado que expressar nosso modelo por meio APIs de alto nível de uma estrutura de *deep learning* requer comparativamente poucas linhas de código. Não tivemos que alocar parâmetros individualmente, definir nossa função de perda ou implementar o gradiente descendente estocástico de *minibatch*. Assim que começarmos a trabalhar com modelos muito mais complexos, as vantagens das APIs de alto nível aumentarão consideravelmente. No entanto, uma vez que temos todas as peças básicas no lugar, o loop de treinamento em si é surpreendentemente semelhante ao que fizemos ao implementar tudo do zero.

Para refrescar sua memória: para anguns números de épocas, faremos uma passagem completa sobre o conjunto de dados (``*train_data*``), pegando iterativamente um *minibatch* de entradas e os *labels* de verdade fundamental correspondentes. Para cada *minibatch*, passamos pelo seguinte ritual:

- Gerar previsões chamando `net (X)` e calcular a perda `l` (a propagação direta).
- Calcular gradientes executando a retropropagação.
- Atualizar os parâmetros do modelo invocando nosso otimizador.

Para uma boa medida, calculamos a perda após cada época e a imprimimos para monitorar o progresso.

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X) ,y)
        trainer.zero_grad()
```

(continues on next page)

(continued from previous page)

```
        l.backward()
        trainer.step()
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')
```

```
epoch 1, loss 0.000239
epoch 2, loss 0.000098
epoch 3, loss 0.000099
```

Abaixo, nós comparamos os parâmetros do modelo aprendidos pelo treinamento em dados finitos e os parâmetros reais que geraram nosso *dataset*. Para acessar os parâmetros, primeiro acessamos a camada que precisamos de `net` e, em seguida, acessamos os pesos e a polarização dessa camada. Como em nossa implementação do zero, observe que nossos parâmetros estimados são perto de suas contrapartes verdadeiras.

```
w = net[0].weight.data
print('error in estimating w:', true_w - w.reshape(true_w.shape))
b = net[0].bias.data
print('error in estimating b:', true_b - b)
```

```
error in estimating w: tensor([0.0004, 0.0001])
error in estimating b: tensor([-0.0005])
```

3.3.8 Resumo

- Usando as APIs de alto nível do PyTorch, podemos implementar modelos de forma muito mais concisa.
- No PyTorch, o módulo `data` fornece ferramentas para processamento de dados, o módulo`nn` define um grande número de camadas de rede neural e funções de perda comuns.
- Podemos inicializar os parâmetros substituindo seus valores por métodos que terminam com `_`.

3.3.9 Exercícios

1. Se substituirmos `nn.MSELoss (*reduction* = 'sum')` por `nn.MSELoss ()`, como podemos alterar a taxa de aprendizagem para que o código se comporte de forma idêntica. Por quê?
2. Revise a documentação do PyTorch para ver quais funções de perda e métodos de inicialização são fornecidos. Substitua a perda pela perda de Huber.
3. Como você acessa o gradiente de `net[0].weight`?

Discussions[35]

[35] https://discuss.d2l.ai/t/45

3.4 Regressão *Softmax*

Em Section 3.1, introduzimos a regressão linear, trabalhando através de implementações do zero em Section 3.2 e novamente usando APIs de alto nível de uma estrutura de *deep learning* em Section 3.3 para fazer o trabalho pesado.

A regressão é o martelo que procuramos quando queremos responder a perguntas*quanto?* ou *quantas?*. Se você deseja prever o número de dólares (preço) a que uma casa será vendida, ou o número de vitórias que um time de beisebol pode ter, ou o número de dias que um paciente permanecerá hospitalizado antes de receber alta, então provavelmente você está procurando um modelo de regressão.

Na prática, estamos mais frequentemente interessados na *classificação*: perguntando não "quanto", mas "qual":

- Este e-mail pertence à pasta de spam ou à caixa de entrada?
- É mais provável que este cliente *se inscreva* ou *não se inscreva* em um serviço de assinatura?
- Esta imagem retrata um burro, um cachorro, um gato ou um galo?
- Qual filme Aston tem mais probabilidade de assistir a seguir?

Coloquialmente, praticantes de *machine learning* sobrecarregam a palavra *classificação* para descrever dois problemas sutilmente diferentes: (i) aqueles em que estamos interessados apenas em atribuições difíceis de exemplos a categorias (classes); e (ii) aqueles em que desejamos fazer atribuições leves, ou seja, para avaliar a probabilidade de que cada categoria se aplica. A distinção tende a ficar confusa, em parte, porque muitas vezes, mesmo quando nos preocupamos apenas com tarefas difíceis, ainda usamos modelos que fazem atribuições suaves.

3.4.1 Problema de Classificação

Para molhar nossos pés, vamos começar com um problema simples de classificação de imagens. Aqui, cada entrada consiste em uma imagem em tons de cinza 2×2. Podemos representar cada valor de pixel com um único escalar, dando-nos quatro características x_1, x_2, x_3, x_4. Além disso, vamos supor que cada imagem pertence a uma entre as categorias "gato", "frango" e "cachorro".

A seguir, temos que escolher como representar os *labels*. Temos duas escolhas óbvias. Talvez o impulso mais natural seja escolher $y \in \{1, 2, 3\}$, onde os inteiros representam $\{\text{cachorro}, \text{gato}, \text{frango}\}$ respectivamente. Esta é uma ótima maneira de *armazenar* essas informações em um computador. Se as categorias tivessem alguma ordem natural entre elas, digamos se estivéssemos tentando prever $\{\text{bebê}, \text{criança}, \text{adolescente}, \text{jovem adulto}, \text{adulto}, \text{idoso}\}$, então pode até fazer sentido lançar este problema como uma regressão e manter os rótulos neste formato.

Mas os problemas gerais de classificação não vêm com ordenações naturais entre as classes. Felizmente, os estatísticos há muito tempo inventaram uma maneira simples para representar dados categóricos: a *codificação one-hot*. Uma codificação *one-hot* é um vetor com tantos componentes quantas categorias temos. O componente correspondente à categoria da instância em particular é definido como 1 e todos os outros componentes são definidos como 0. Em nosso caso, um rótulo y seria um vetor tridimensional, com $(1, 0, 0)$ correspondendo a "gato", $(0, 1, 0)$ a "galinha", e $(0, 0, 1)$ para "cachorro":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{3.4.1}$$

3.4.2 Arquitetura de Rede

A fim de estimar as probabilidades condicionais associadas a todas as classes possíveis, precisamos de um modelo com várias saídas, uma por classe. Para abordar a classificação com modelos lineares, precisaremos de tantas funções afins quantas forem as saídas. Cada saída corresponderá a sua própria função afim. No nosso caso, uma vez que temos 4 *features* e 3 categorias de saída possíveis, precisaremos de 12 escalares para representar os pesos (w com subscritos), e 3 escalares para representar os *offsets* (b com subscritos). Calculamos esses três *logits*, o_1, o_2, and o_3, para cada entrada:

$$\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,\\
o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,\\
o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.
\end{aligned} \tag{3.4.2}$$

Podemos representar esse cálculo com o diagrama da rede neural mostrado em Fig. 3.4.1. Assim como na regressão linear, a regressão *softmax* também é uma rede neural de camada única. E desde o cálculo de cada saída, o_1, o_2, e o_3, depende de todas as entradas, x_1, x_2, x_3, e x_4, a camada de saída da regressão *softmax* também pode ser descrita como uma camada totalmente conectada.

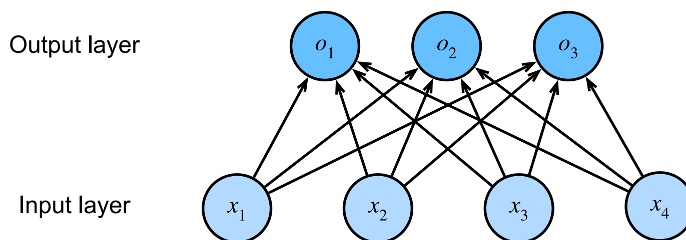


Fig. 3.4.1: Softmax regression is a single-layer neural network.

Para expressar o modelo de forma mais compacta, podemos usar a notação de álgebra linear. Na forma vetorial, chegamos a $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, uma forma mais adequada tanto para matemática quanto para escrever código. Observe que reunimos todos os nossos pesos em uma matriz 3×4 e que para características de um dado exemplo de dados $\mathbf{x}$, nossas saídas são dadas por um produto vetor-matriz de nossos pesos por nossos recursos de entrada mais nossos *offsets* $\mathbf{b}$.

3.4.3 Custo de Parametrização de Camadas Totalmente Conectadas

Como veremos nos capítulos subsequentes, camadas totalmente conectadas são onipresentes no *deep learning*. No entanto, como o nome sugere, camadas totalmente conectadas são *totalmente* conectadas com muitos parâmetros potencialmente aprendíveis. Especificamente, para qualquer camada totalmente conectada com d entradas e q saídas, o custo de parametrização é $\mathcal{O}(dq)$, que pode ser proibitivamente alto na prática. Felizmente, este custo de transformar d entradas em q saídas pode ser reduzido a $\mathcal{O}(\frac{dq}{n})$, onde o hiperparâmetro n pode ser especificado de maneira flexível por nós para equilibrar entre o salvamento de parâmetros e a eficácia do modelo em aplicações do mundo real (Zhang et al., 2021).

3.4.4 Operação do *Softmax*

A abordagem principal que vamos adotar aqui é interpretar as saídas de nosso modelo como probabilidades. Vamos otimizar nossos parâmetros para produzir probabilidades que maximizam a probabilidade dos dados observados. Então, para gerar previsões, vamos definir um limite, por exemplo, escolhendo o *label* com as probabilidades máximas previstas.

Colocado formalmente, gostaríamos de qualquer saída $\hat{y}_j$ fosse interpretada como a probabilidade que um determinado item pertence à classe j. Então podemos escolher a classe com o maior valor de saída como nossa previsão $\operatorname{argmax}_j y_j$. Por exemplo, se $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ são 0,1, 0,8 e 0,1, respectivamente, então, prevemos a categoria 2, que (em nosso exemplo) representa "frango".

Você pode ficar tentado a sugerir que interpretemos os *logits* o diretamente como nossas saídas de interesse. No entanto, existem alguns problemas com interpretação direta da saída da camada linear como uma probabilidade. Por um lado, nada restringe esses números a somarem 1. Por outro lado, dependendo das entradas, podem assumir valores negativos. Estes violam axiomas básicos de probabilidade apresentados em Section 2.6

Para interpretar nossos resultados como probabilidades, devemos garantir que (mesmo em novos dados), eles serão não negativos e somam 1. Além disso, precisamos de um objetivo de treinamento que incentive o modelo para estimar probabilidades com fidelidade. De todas as instâncias quando um classificador produz 0,5, esperamos que metade desses exemplos realmente pertenceram à classe prevista. Esta é uma propriedade chamada *calibração*.

A *função softmax*, inventada em 1959 pelo cientista social R. Duncan Luce no contexto de *modelos de escolha*, faz exatamente isso. Para transformar nossos *logits* de modo que eles se tornem não negativos e somem 1, ao mesmo tempo em que exigimos que o modelo permaneça diferenciável, primeiro exponenciamos cada *logit* (garantindo a não negatividade) e, em seguida, dividimos pela soma (garantindo que somem 1):

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}. \tag{3.4.3}$$

É fácil ver $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \leq \hat{y}_j \leq 1$ para todo j. Assim, $\hat{\mathbf{y}}$ é uma distribuição de probabilidade adequada cujos valores de elementos podem ser interpretados em conformidade. Observe que a operação *softmax* não muda a ordem entre os *logits* $\mathbf{o}$, que são simplesmente os valores pré-*softmax* que determinam as probabilidades atribuídas a cada classe. Portanto, durante a previsão, ainda podemos escolher a classe mais provável por

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{3.4.4}$$

Embora *softmax* seja uma função não linear, as saídas da regressão *softmax* ainda são *determinadas* por uma transformação afim de recursos de entrada; portanto, a regressão *softmax* é um modelo linear.

3.4.5 Vetorização para *Minibatches*

Para melhorar a eficiência computacional e aproveitar as vantagens das GPUs, normalmente realizamos cálculos vetoriais para *minibatches* de dados. Suponha que recebemos um *minibatch* $\mathbf{X}$ de exemplos com dimensionalidade do recurso (número de entradas) d e tamanho do lote n. Além disso, suponha que temos q categorias na saída. Então os *features* de *minibatch* $\mathbf{X}$ estão em $\mathbb{R}^{n \times d}$, pesos $\mathbf{W} \in \mathbb{R}^{d \times q}$, e o *bias* satisfaz $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned} \tag{3.4.5}$$

Isso acelera a operação dominante em um produto matriz-matriz $\mathbf{X}\mathbf{W}$ vs. os produtos de vetor-matriz que estaríamos executando se processamos um exemplo de cada vez. Uma vez que cada linha em $\mathbf{X}$ representa um exemplo de dados, a própria operação *softmax* pode ser calculada *rowwise* (através das colunas): para cada linha de $\mathbf{O}$, exponenciando todas as entradas e depois normalizando-as pela soma. Disparando a transmissão durante a soma $\mathbf{X}\mathbf{W} + \mathbf{b}$ in (3.4.5), o *minibatch* registra $\mathbf{O}$ e as probabilidades de saída $\hat{\mathbf{Y}}$ são matrizes $n \times q$.

3.4.6 Função de Perda

Em seguida, precisamos de uma função de perda para medir a qualidade de nossas probabilidades previstas. Contaremos com a estimativa de probabilidade máxima, o mesmo conceito que encontramos ao fornecer uma justificativa probabilística para o objetivo de erro quadrático médio na regressão linear (Section 3.1.3).

Log-Likelihood

A função *softmax* nos dá um vetor $\hat{\mathbf{y}}$, que podemos interpretar como probabilidades condicionais estimadas de cada classe dada qualquer entrada $\mathbf{x}$,, por exemplo, $\hat{y}_1$ = $P(y = \text{cat} \mid \mathbf{x})$. Suponha que todo o conjunto de dados $\{\mathbf{X}, \mathbf{Y}\}$ tenha n exemplos, onde o exemplo indexado por i consiste em um vetor de característica $\mathbf{x}^{(i)}$ e um vetor de rótulo único $\mathbf{y}^{(i)}$. Podemos comparar as estimativas com a realidade verificando quão prováveis as classes reais são de acordo com nosso modelo, dadas as características:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{3.4.6}$$

De acordo com a estimativa de máxima *likelihood*, maximizamos $P(\mathbf{Y} \mid \mathbf{X})$, que é equivalente a minimizar a probabilidade de *log-likelihood* negativo:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{3.4.7}$$

onde para qualquer par de rótulo $\mathbf{y}$ e predição de modelo $\hat{\mathbf{y}}$ sobre q classes, a função de perda l é

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{3.4.8}$$

Por razões explicadas mais tarde, a função de perda em (3.4.8) é comumente chamada de *perda de entropia cruzada*. Uma vez que $\mathbf{y}$ é um vetor *one-hot* de comprimento q, a soma de todas as suas coordenadas j desaparece para todos, exceto um termo. Uma vez que todos os $\hat{y}_j$ são probabilidades

previstas, seu logaritmo nunca é maior que 0. Consequentemente, a função de perda não pode ser minimizada mais, se predizermos corretamente o rótulo real com *certeza*, ou seja, se a probabilidade prevista $P(\mathbf{y} \mid \mathbf{x}) = 1$ for o *label* real $\mathbf{y}$. Observe que isso geralmente é impossível. Por exemplo, pode haver ruído de *label* no *dataset* (alguns exemplos podem estar classificados incorretamente). Também pode não ser possível quando os recursos de entrada não são suficientemente informativos para classificar todos os exemplos perfeitamente.

Softmax e Derivadas

Uma vez que o *softmax* e a perda correspondente são tão comuns, vale a pena entender um pouco melhor como ele é calculado. Conectando (3.4.3) na definição da perda em (3.4.8) e usando a definição do *softmax* obtemos:

$$
\begin{aligned}
l(\mathbf{y}, \hat{\mathbf{y}}) &= - \sum_{j=1}^q y_j \log \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} \\
&= \sum_{j=1}^q y_j \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j \\
&= \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j.
\end{aligned} \tag{3.4.9}
$$

Para entender um pouco melhor o que está acontecendo, considere a derivada com respeito a qualquer *logit* o_j. Nós temos

$$
\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} - y_j = \mathrm{softmax}(\mathbf{o})_j - y_j. \tag{3.4.10}
$$

Em outras palavras, a derivada é a diferença entre a probabilidade atribuída pelo nosso modelo, conforme expresso pela operação *softmax*, e o que realmente aconteceu, conforme expresso por elementos no vetor *one-hot* de *labels*. Nesse sentido, é muito semelhante ao que vimos na regressão, onde o gradiente era a diferença entre a observação y e a estimativa $\hat{y}$. Isso não é coincidência. Em qualquer família exponencial (veja o modelo no apêndice online sobre distribuições[36]), os gradientes da probabilidade logarítmica são dados precisamente por esse termo. Esse fato torna a computação dos gradientes fáceis na prática.

Perda de Entropia Cruzada

Agora considere o caso em que observamos não apenas um único resultado mas toda uma distribuição de resultados. Podemos usar a mesma representação de antes para o rótulo $\mathbf{y}$. A única diferença é que, em vez de um vetor contendo apenas entradas binárias, digamos $(0, 0, 1)$, agora temos um vetor de probabilidade genérico, digamos $(0.1, 0.2, 0.7)$. A matemática que usamos anteriormente para definir a perda l em (3.4.8) ainda funciona bem, apenas que a interpretação é um pouco mais geral. É o valor esperado da perda de uma distribuição nos *labels*. Esta perda é chamada de *perda de entropia cruzada* e é uma das perdas mais comumente usadas para problemas de classificação. Podemos desmistificar o nome apresentando apenas os fundamentos da teoria da informação. Se você deseja entender mais detalhes da teoria da informação, você também pode consultar o apêndice online sobre teoria da informação[37].

[36] https://d2l.ai/chapter_apencha-mathematics-for-deep-learning/distributions.html
[37] https://d2l.ai/chapter_apencha-mathematics-for-deep-learning/information-theory.html

3.4.7 Fundamentos da Teoria da Informação

Teoria da informação lida com o problema de codificação, decodificação, transmissão, e manipulação informações (também conhecidas como dados) da forma mais concisa possível.

Entropia

A ideia central na teoria da informação é quantificar o conteúdo da informação nos dados. Essa quantidade impõe um limite rígido à nossa capacidade de compactar os dados. Na teoria da informação, essa quantidade é chamada de *entropia* de uma distribuição P, e é definida pela seguinte equação:

$$H[P] = \sum_j -P(j) \log P(j). \tag{3.4.11}$$

Um dos teoremas fundamentais da teoria da informação afirma que, a fim de codificar dados retirados aleatoriamente da distribuição P, precisamos de pelo menos $H[P]$ "*nats*" para codificá-lo. Se você quer saber o que é um "*nat*", é o equivalente a bit mas ao usar um código com base e em vez de um com base 2. Assim, um *nat* é $\frac{1}{\log(2)} \approx 1.44$ bit.

Surpresa

Você pode estar se perguntando o que a compressão tem a ver com a predição. Imagine que temos um fluxo de dados que queremos compactar. Se sempre for fácil para nós prevermos o próximo *token*, então esses dados são fáceis de compactar! Veja o exemplo extremo em que cada token no fluxo sempre leva o mesmo valor. Esse é um fluxo de dados muito chato! E não só é chato, mas também é fácil de prever. Por serem sempre iguais, não precisamos transmitir nenhuma informação para comunicar o conteúdo do fluxo. Fácil de prever, fácil de compactar.

No entanto, se não podemos prever perfeitamente todos os eventos, então às vezes podemos ficar surpresos. Nossa surpresa é maior quando atribuímos uma probabilidade menor a um evento. Claude Shannon estabeleceu $\log \frac{1}{P(j)} = -\log P(j)$ para quantificar a *surpresa* de alguém ao observar um evento j tendo-lhe atribuído uma probabilidade (subjetiva) $P(j)$. A entropia definida em (3.4.11) é então a *surpresa esperada* quando alguém atribuiu as probabilidades corretas que realmente correspondem ao processo de geração de dados.

Entropia Cruzada Revisitada

Então, se a entropia é o nível de surpresa experimentado por alguém que conhece a verdadeira probabilidade, então você deve estar se perguntando, o que é entropia cruzada? A entropia cruzada *de* P *a* Q, denotada $H(P, Q)$, é a surpresa esperada de um observador com probabilidades subjetivas Q ao ver os dados que realmente foram gerados de acordo com as probabilidades P. A menor entropia cruzada possível é alcançada quando $P = Q$. Nesse caso, a entropia cruzada de P a Q é $H(P, P) = H(P)$.

Em suma, podemos pensar no objetivo da classificação de entropia cruzada de duas maneiras: (i) maximizando a probabilidade dos dados observados; e (ii) minimizando nossa surpresa (e, portanto, o número de bits) necessário para comunicar os rótulos.

3.4.8 Predição do Modelo e Avaliação

Depois de treinar o modelo de regressão *softmax*, dados quaisquer recursos de exemplo, podemos prever a probabilidade de cada classe de saída. Normalmente, usamos a classe com a maior probabilidade prevista como a classe de saída. A previsão está correta se for consistente com a classe real (*label*). Na próxima parte do experimento, usaremos *exatidão* para avaliar o desempenho do modelo. Isso é igual à razão entre o número de previsões corretas e o número total de previsões.

3.4.9 Resumo

- A operação *softmax* pega um vetor e o mapeia em probabilidades.
- A regressão *Softmax* se aplica a problemas de classificação. Ela usa a distribuição de probabilidade da classe de saída na operação *softmax*.
- A entropia cruzada é uma boa medida da diferença entre duas distribuições de probabilidade. Ela mede o número de bits necessários para codificar os dados de nosso modelo.

3.4.10 Exercícios

1. Podemos explorar a conexão entre as famílias exponenciais e o *softmax* com um pouco mais de profundidade.
    1. Calcule a segunda derivada da perda de entropia cruzada $l(\mathbf{y}, \hat{\mathbf{y}})$ para o *softmax*.
    2. Calcule a variância da distribuição dada por $\mathrm{softmax}(\mathbf{o})$ e mostre que ela corresponde à segunda derivada calculada acima.
2. Suponha que temos três classes que ocorrem com probabilidade igual, ou seja, o vetor de probabilidade é $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    1. Qual é o problema se tentarmos projetar um código binário para ele?
    2. Você pode criar um código melhor? Dica: o que acontece se tentarmos codificar duas observações independentes? E se codificarmos n observações em conjunto?
3. *Softmax* é um nome impróprio para o mapeamento apresentado acima (mas todos no aprendizado profundo o usam). O *softmax* real é definido como $\mathrm{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
    1. Prove que $\mathrm{RealSoftMax}(a, b) > \mathrm{max}(a, b)$.
    2. Prove que isso vale para$\lambda^{-1} \mathrm{RealSoftMax}(\lambda a, \lambda b)$, desde que $\lambda > 0$.
    3. Mostre que para $\lambda \to \infty$ temos $\lambda^{-1} \mathrm{RealSoftMax}(\lambda a, \lambda b) \to \mathrm{max}(a, b)$.
    4. Qual é a aparência do soft-min?
    5. Estenda isso para mais de dois números.

Discussions[38]

[38] https://discuss.d2l.ai/t/46

3.5 O *Dataset* de Classificação de Imagens

Um dos *datasets* amplamente usados para classificação de imagens é o conjunto de dados MNIST (LeCun et al., 1998). Embora tenha tido uma boa execução como um conjunto de dados de referência, mesmo os modelos simples pelos padrões atuais alcançam uma precisão de classificação acima de 95%, tornando-o inadequado para distinguir entre modelos mais fortes e mais fracos. Hoje, o MNIST serve mais como verificação de sanidade do que como referência. Para aumentar um pouco a aposta, concentraremos nossa discussão nas próximas seções no *dataset* Fashion-MNIST, qualitativamente semelhante, mas comparativamente complexo (Xiao et al., 2017), que foi lançado em 2017.

```
%matplotlib inline
import torch
import torchvision
from torch.utils import data
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

3.5.1 Lendo o Dataset

Nós podemos baixar e ler o *dataset* Fashion-MNIST na memória por meio das funções integradas na estrutura.

```
# `ToTensor` converts the image data from PIL type to 32-bit floating point
# tensors. It divides all numbers by 255 so that all pixel values are between
# 0 and 1
trans = transforms.ToTensor()
mnist_train = torchvision.datasets.FashionMNIST(
    root="../data", train=True, transform=trans, download=True)
mnist_test = torchvision.datasets.FashionMNIST(
    root="../data", train=False, transform=trans, download=True)
```

O Fashion-MNIST consiste em imagens de 10 categorias, cada uma representada por 6.000 imagens no conjunto de dados de treinamento e por 1.000 no conjunto de dados de teste. Um *dataset de teste* (ou *conjunto de teste*) é usado para avaliar o desempenho do modelo e não para treinamento. Consequentemente, o conjunto de treinamento e o conjunto de teste contém 60.000 e 10.000 imagens, respectivamente.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

A altura e a largura de cada imagem de entrada são 28 pixels. Observe que o *dataset* consiste em imagens em tons de cinza, cujo número de canais é 1. Para resumir, ao longo deste livro armazenamos a forma de qualquer imagem com altura h largura w pixels como $h \times w$ or (h, w).

```
mnist_train[0][0].shape
```

```
torch.Size([1, 28, 28])
```

As imagens no Fashion-MNIST estão associadas às seguintes categorias: t-shirt, calças, pulôver, vestido, casaco, sandália, camisa, tênis, bolsa e bota. A função a seguir converte entre índices de rótulos numéricos e seus nomes em texto.

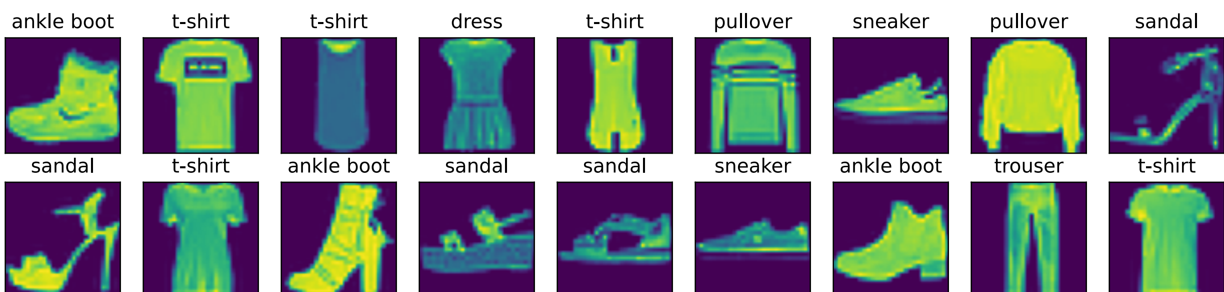
```
def get_fashion_mnist_labels(labels):  #@save
    """Return text labels for the Fashion-MNIST dataset."""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

Agora podemos criar uma função para visualizar esses exemplos.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        if torch.is_tensor(img):
            # Tensor Image
            ax.imshow(img.numpy())
        else:
            # PIL Image
            ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Aqui estão as imagens e seus *labels correspondentes (no texto) para os primeiros exemplos no*dataset* de treinamento.

```
X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels(y));
```



3.5.2 Lendo um *Minibatch*

Para tornar nossa vida mais fácil ao ler os conjuntos de treinamento e teste, usamos o iterador de dados integrado em vez de criar um do zero. Lembre-se de que a cada iteração, um carregador de dados lê um minibatch de dados com tamanho `batch_size` cada vez. Também misturamos aleatoriamente os exemplos para o iterador de dados de treinamento.

```
batch_size = 256

def get_dataloader_workers():  #@save
    """Use 4 processes to read the data."""
    return 4

train_iter = data.DataLoader(mnist_train, batch_size, shuffle=True,
                             num_workers=get_dataloader_workers())
```

Vejamos o tempo que leva para ler os dados de treinamento.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
f'{timer.stop():.2f} sec'
```

```
'1.96 sec'
```

3.5.3 Juntando Tudo

Agora definimos a função `load_data_fashion_mnist` que obtém e lê o *dataset* Fashion-MNIST. Ele retorna os iteradores de dados para o conjunto de treinamento e o conjunto de validação. Além disso, ele aceita um argumento opcional para redimensionar imagens para outra forma.

```
def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """Download the Fashion-MNIST dataset and then load it into memory."""
    trans = [transforms.ToTensor()]
    if resize:
        trans.insert(0, transforms.Resize(resize))
    trans = transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (data.DataLoader(mnist_train, batch_size, shuffle=True,
                            num_workers=get_dataloader_workers()),
            data.DataLoader(mnist_test, batch_size, shuffle=False,
                            num_workers=get_dataloader_workers()))
```

Abaixo testamos o recurso de redimensionamento de imagem da função `load_data_fashion_mnist` especificando o argumento `resize`.

```
train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
for X, y in train_iter:
```

(continues on next page)

(continued from previous page)

```
    print(X.shape, X.dtype, y.shape, y.dtype)
    break
```

```
torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64
```

Agora estamos prontos para trabalhar com o *dataset* Fashion-MNIST nas seções a seguir.

3.5.4 Resumo

- Fashion-MNIST é um *dataset* de classificação de vestuário que consiste em imagens que representam 10 categorias. Usaremos esse conjunto de dados nas seções e capítulos subsequentes para avaliar vários algoritmos de classificação.
- Armazenamos a forma de qualquer imagem com altura h largura w pixels como $h \times w$ or (h, w).
- Os iteradores de dados são um componente chave para um desempenho eficiente. Conte com iteradores de dados bem implementados que exploram a computação de alto desempenho para evitar desacelerar o ciclo de treinamento.

3.5.5 Exercícios

1. A redução de `batch_size` (por exemplo, para 1) afeta o desempenho de leitura?
2. O desempenho do iterador de dados é importante. Você acha que a implementação atual é rápida o suficiente? Explore várias opções para melhorá-lo.
3. Verifique a documentação online da API do *framework*. Quais outros conjuntos de dados estão disponíveis?

Discussions[39]

3.6 Implementação da Regressão *Softmax* do Zero

Assim como implementamos a regressão linear do zero, acreditamos que regressão *softmax* é igualmente fundamental e você deve saber os detalhes sangrentos de

como implementá-lo sozinho. Vamos trabalhar com o *dataset* Fashion-MNIST, recém-introduzido em Section 3.5, configurando um iterador de dados com *batch size* 256.

```
import torch
from IPython import display
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

[39] https://discuss.d2l.ai/t/49

3.6.1 Inicializando os Parâmetros do Modelo

Como em nosso exemplo de regressão linear, cada exemplo aqui será representado por um vetor de comprimento fixo. Cada exemplo no conjunto de dados bruto é uma imagem 28×28. Nesta seção, vamos nivelar cada imagem, tratando-os como vetores de comprimento 784. No futuro, falaremos sobre estratégias mais sofisticadas para explorar a estrutura espacial em imagens, mas, por enquanto, tratamos cada localização de pixel como apenas outro recurso.

Lembre-se de que na regressão *softmax*, temos tantas saídas quanto classes. Como nosso conjunto de dados tem 10 classes, nossa rede terá uma dimensão de saída de 10. Consequentemente, nossos pesos constituirão uma matriz 784×10 e os *bias* constituirão um vetor-linha 1×10. Tal como acontece com a regressão linear, vamos inicializar nossos pesos `W` com ruído Gaussiano e nossos *bias* com o valor inicial 0.

```
num_inputs = 784
num_outputs = 10

W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True)
b = torch.zeros(num_outputs, requires_grad=True)
```

3.6.2 Definindo a Operação do *Softmax*

Antes de implementar o modelo de regressão do *softmax*, vamos revisar brevemente como o operador de soma funciona ao longo de dimensões específicas em um tensor, conforme discutido em: numref Section 2.3.6 e Section 2.3.6. Dada uma matriz `X`, podemos somar todos os elementos (por padrão) ou apenas sobre elementos no mesmo eixo, ou seja, a mesma coluna (eixo 0) ou a mesma linha (eixo 1). Observe que se `X` é um tensor com forma (2, 3) e somamos as colunas, o resultado será um vetor com forma (3,). Ao invocar o operador de soma, podemos especificar para manter o número de eixos no tensor original, em vez de reduzir a dimensão que resumimos. Isso resultará em um tensor bidimensional com forma (1, 3).

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdim=True), X.sum(1, keepdim=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

Agora estamos prontos para implementar a operação do *softmax* . Lembre-se de que o *softmax* consiste em três etapas: i) exponenciamos cada termo (usando `exp`); ii) somamos cada linha (temos uma linha por exemplo no lote) para obter a constante de normalização para cada exemplo; iii) dividimos cada linha por sua constante de normalização, garantindo que o resultado seja 1. Antes de olhar para o código, vamos lembrar como isso parece, expresso como uma equação:

$$\mathrm{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{3.6.1}$$

O denominador, ou constante de normalização, às vezes também é chamada de *função de partição* (e seu logaritmo é chamado de função de partição de log). As origens desse nome estão em física estatística[40] onde uma equação relacionada modela a distribuição sobre um conjunto de partículas.

[40] https://en.wikipedia.org/wiki/Partition_function_%20(estatística_mecânica)

```python
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdim=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

Como você pode ver, para qualquer entrada aleatória, transformamos cada elemento em um número não negativo. Além disso, cada linha soma 1, como é necessário para uma probabilidade.

```python
X = torch.normal(0, 1, (2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2947, 0.2476, 0.0139, 0.3211, 0.1228],
         [0.0861, 0.2708, 0.0988, 0.0651, 0.4792]]),
 tensor([1.0000, 1.0000]))
```

Observe que embora pareça correto matematicamente, fomos um pouco desleixados em nossa implementação porque falhamos em tomar precauções contra estouro numérico ou estouro negativo devido a elementos grandes ou muito pequenos da matriz.

3.6.3 Definindo o Modelo

Agora que definimos a operação do *softmax*, podemos implementar o modelo de regressão softmax. O código a seguir define como a entrada é mapeada para a saída por meio da rede. Observe que achatamos cada imagem original no lote em um vetor usando a função `reshape` antes de passar os dados pelo nosso modelo.

```python
def net(X):
    return softmax(torch.matmul(X.reshape((-1, W.shape[0])), W) + b)
```

3.6.4 Definindo a Função de Perda

Em seguida, precisamos implementar a função de perda de entropia cruzada, conforme apresentado em Section 3.4. Esta pode ser a função de perda mais comum em todo o *deep learning* porque, no momento, os problemas de classificação superam em muito os problemas de regressão.

Lembre-se de que a entropia cruzada leva a *log-likelihood* negativa da probabilidade prevista atribuída ao rótulo verdadeiro. Em vez de iterar as previsões com um *loop for* Python (que tende a ser ineficiente), podemos escolher todos os elementos por um único operador. Abaixo, nós criamos dados de amostra `y_hat` com 2 exemplos de probabilidades previstas em 3 classes e seus rótulos correspondentes `y`. Com `y` sabemos que no primeiro exemplo a primeira classe é a previsão correta e no segundo exemplo, a terceira classe é a verdade fundamental. Usando `y` como os índices das probabilidades em`y_hat`, escolhemos a probabilidade da primeira classe no primeiro exemplo e a probabilidade da terceira classe no segundo exemplo.

```python
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

Agora podemos implementar a função de perda de entropia cruzada de forma eficiente com apenas uma linha de código.

```
def cross_entropy(y_hat, y):
    return - torch.log(y_hat[range(len(y_hat)), y])

cross_entropy(y_hat, y)
```

```
tensor([2.3026, 0.6931])
```

3.6.5 Exatidão da Classificação

Dada a distribuição de probabilidade prevista `y_hat`, normalmente escolhemos a classe com a maior probabilidade prevista sempre que a previsão que devemos produzir é difícil. Na verdade, muitos aplicativos exigem que façamos uma escolha. O Gmail deve categorizar um e-mail em "Principal", "Social", "Atualizações" ou "Fóruns". Pode estimar probabilidades internamente, mas no final do dia ele tem que escolher uma das classes.

Quando as previsões são consistentes com a classe de *label* `y`, elas estão corretas. A precisão da classificação é a fração de todas as previsões corretas. Embora possa ser difícil otimizar a precisão diretamente (não é diferenciável), muitas vezes é a medida de desempenho que mais nos preocupa, e quase sempre o relataremos ao treinar classificadores.

Para calcular a precisão, fazemos o seguinte. Primeiro, se `y_hat` é uma matriz, presumimos que a segunda dimensão armazena pontuações de predição para cada classe. Usamos `argmax` para obter a classe prevista pelo índice para a maior entrada em cada linha. Em seguida, comparamos a classe prevista com a verdade fundamental `y` elemento a elemento. Uma vez que o operador de igualdade == é sensível aos tipos de dados, convertemos o tipo de dados de `y_hat` para corresponder ao dey. O resultado é um tensor contendo entradas de 0 (falso) e 1 (verdadeiro). Tirar a soma resulta no número de previsões corretas.

```
def accuracy(y_hat, y):  #@save
    """Compute the number of correct predictions."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())
```

Continuaremos a usar as variáveis `y_hat` e`y` definidas antes como as distribuições de probabilidade e *labels* previstos, respectivamente. Podemos ver que a classe prevista no primeiro exemplo é 2 (o maior elemento da linha é 0,6 com o índice 2), que é inconsistente com o rótulo real, 0. A classe prevista do segundo exemplo é 2 (o maior elemento da linha é 0,5 com o índice de 2), que é consistente com o rótulo real, 2. Portanto, a taxa de precisão da classificação para esses dois exemplos é 0,5.

```
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Da mesma forma, podemos avaliar a precisão da `rede` de qualquer modelo em um conjunto de dados que é acessado por meio do iterador de dados `data_iter`.

```
def evaluate_accuracy(net, data_iter):  #@save
    """Compute the accuracy for a model on a dataset."""
    if isinstance(net, torch.nn.Module):
        net.eval()  # Set the model to evaluation mode
    metric = Accumulator(2)  # No. of correct predictions, no. of predictions
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

Aqui, `Accumulator` é uma classe utilitária para acumular somas sobre múltiplas variáveis. Na função `evaluate_accuracy` acima, criamos 2 variáveis na instância `Accumulator` para armazenar ambos o número de previsões corretas e o número de previsões, respectivamente. Ambos serão acumulados ao longo do tempo à medida que iteramos no conjunto de dados.

```
class Accumulator:  #@save
    """For accumulating sums over `n` variables."""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

PComo inicializamos o modelo `net` com pesos aleatórios, a precisão deste modelo deve ser próxima à aleatoriedade, ou seja, 0,1 para 10 classes.

```
evaluate_accuracy(net, test_iter)
```

```
0.0918
```

3.6.6 Treinamento

O *loop* de treinamento para regressão *softmax* deve ser extremamente familiar se você ler nossa implementação de regressão linear em Section 3.2. Aqui, nós refatoramos a implementação para torná-la reutilizável. Primeiro, definimos uma função para treinar por uma época. Observe que `updater` é uma função geral para atualizar os parâmetros do modelo, que aceita o tamanho do lote como argumento. Pode ser um *wrapper* da função `d2l.sgd` ou a função de otimização integrada de uma estrutura.

```
def train_epoch_ch3(net, train_iter, loss, updater):  #@save
    """The training loop defined in Chapter 3."""
    # Set the model to training mode
    if isinstance(net, torch.nn.Module):
        net.train()
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        y_hat = net(X)
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            # Using PyTorch in-built optimizer & loss criterion
            updater.zero_grad()
            l.backward()
            updater.step()
            metric.add(float(l) * len(y), accuracy(y_hat, y),
                       y.size().numel())
        else:
            # Using custom built optimizer & loss criterion
            l.sum().backward()
            updater(X.shape[0])
            metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())
    # Return training loss and training accuracy
    return metric[0] / metric[2], metric[1] / metric[2]
```

Antes de mostrar a implementação da função de treinamento, definimos uma classe de utilitário que plota dados em animação. Novamente, o objetivo é simplificar o código no restante do livro.

```
class Animator:  #@save
    """For plotting data in animation."""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # Incrementally plot multiple lines
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda function to capture arguments
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # Add multiple data points into the figure
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
```

(continues on next page)

(continued from previous page)

```
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

A seguinte função de treinamento então treina um modelo `net` em um conjunto de dados de treinamento acessado via`train_iter` para várias épocas, que é especificado por `num_epochs`. No final de cada época, o modelo é avaliado em um conjunto de dados de teste acessado via `test_iter`. Vamos aproveitar a classe `Animator` para visualizar o progresso do treinamento.

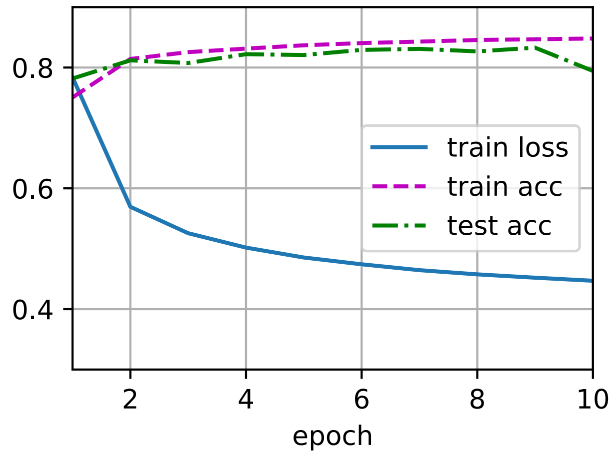
```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """Train a model (defined in Chapter 3)."""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

Como uma implementação do zero, nós usamos a descida gradiente estocástica do *minibatch* definido em Section 3.2 para otimizar a função de perda do modelo com uma taxa de aprendizado de 0,1.

```
lr = 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

Agora treinamos o modelo com 10 épocas. Observe que tanto o número de épocas (`num_epochs`), e a taxa de aprendizagem (`lr`) são hiperparâmetros ajustáveis. Mudando seus valores, podemos ser capazes de aumentar a precisão da classificação do modelo.

```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



3.6.7 Predição

Agora que o treinamento está completo, nosso modelo está pronto para classificar algumas imagens. Dada uma série de imagens, vamos comparar seus *labels* reais (primeira linha de saída de texto) e as previsões do modelo (segunda linha de saída de texto).

```
def predict_ch3(net, test_iter, n=6):  #@save
    """Predict labels (defined in Chapter 3)."""
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true +'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(
        X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```

3.6.8 Resumo

- Com a regressão *softmax*, podemos treinar modelos para classificação multiclasse.
- O loop de treinamento da regressão *softmax* é muito semelhante ao da regressão linear: recuperar e ler dados, definir modelos e funções de perda e treinar modelos usando algoritmos de otimização. Como você descobrirá em breve, os modelos de *deep learning* mais comuns têm procedimentos de treinamento semelhantes.

3.6.9 Exercícios

1. Nesta seção, implementamos diretamente a função *softmax* com base na definição matemática da operação do *softmax*. Que problemas isso pode causar? Dica: tente calcular o tamanho de $\exp(50)$.
2. A função `cross_entropy` nesta seção foi implementada de acordo com a definição da função de perda de entropia cruzada. Qual poderia ser o problema com esta implementação? Dica: considere o domínio do logaritmo.
3. Que soluções você pode pensar para resolver os dois problemas acima?
4. É sempre uma boa ideia retornar o *label* mais provável? Por exemplo, você faria isso para diagnóstico médico?
5. Suponha que quiséssemos usar a regressão *softmax* para prever a próxima palavra com base em alguns recursos. Quais são alguns problemas que podem surgir de um vocabulário extenso?

Discussions[41]

3.7 Implementação Concisa da Regressão *Softmax*

APIs de alto nível tal como os *frameworks* de *deep learning* tornaram muito mais fácil de implementar a regressão linear em Section 3.3, encontraremos de forma semelhante (ou possivelmente mais) conveniente, implementar modelos de classificação. Vamos ficar com o conjunto de dados *Fashion-MNIST* e manter o tamanho do lote em 256 como em Section 3.6.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

[41] https://discuss.d2l.ai/t/51

3.7.1 Inicializando os Parâmetros do Modelo

Conforme mencionado em: numref: `sec_softmax`, a camada de saída da regressão *softmax* é uma camada totalmente conectada. Portanto, para implementar nosso modelo, só precisamos adicionar uma camada totalmente conectada com 10 saídas para nosso `Sequential`. Novamente, aqui, o `Sequential` não é realmente necessário, mas podemos também criar o hábito, pois será onipresente ao implementar modelos profundos. Novamente, inicializamos os pesos aleatoriamente com média zero e desvio padrão 0,01.

```
# PyTorch does not implicitly reshape the inputs. Thus we define the flatten
# layer to reshape the inputs before the linear layer in our network
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

3.7.2 Implementação do *Softmax* Revisitada

No exemplo anterior de Section 3.6, calculamos a saída do nosso modelo e então executamos esta saída através da perda de entropia cruzada. Matematicamente, isso é uma coisa perfeitamente razoável de se fazer. No entanto, de uma perspectiva computacional, a exponenciação pode ser uma fonte de problemas de estabilidade numérica.

Lembre-se de que a função *softmax* calcula $\hat y_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$, onde $\hat y_j$ é o elemento j^th da distribuição de probabilidade prevista $\hat{\mathbf{y}}$ e o_j é o elemento j^th dos *logits* $\mathbf{o}$. Se alguns dos o_k forem muito grandes (ou seja, muito positivos), então $\exp(o_k)$ pode ser maior que o maior número, podemos ter para certos tipos de dados (ou seja, *estouro*). Isso tornaria o denominador (e/ou numerador) `inf` (infinito) e acabamos encontrando 0, `inf` ou`nan` (não um número) para $\hat y_j$. Nessas situações, não obtemos uma definição bem definida valor de retorno para entropia cruzada.

Um truque para contornar isso é primeiro subtrair $\max(o_k)$ de todos o_k antes de prosseguir com o cálculo do *softmax*. Você pode ver que este deslocamento de cada o_k por um fator constante não altera o valor de retorno de *softmax*:

$$
\begin{aligned}
\hat y_j & = \frac{\exp(o_j - \max(o_k))\exp(\max(o_k))}{\sum_k \exp(o_k - \max(o_k))\exp(\max(o_k))} \\
& = \frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}.
\end{aligned} \tag{3.7.1}
$$

Após a etapa de subtração e normalização, pode ser possível que alguns $o_j - \max(o_k)$ tenham grandes valores negativos e assim que o $\exp(o_j - \max(o_k))$ correspondente assumirá valores próximos a zero. Eles podem ser arredondados para zero devido à precisão finita (ou seja, *underflow*), tornando $\hat y_j$ zero e dando-nos `-inf` para $\log(\hat y_j)$. Alguns passos abaixo na *backpropagation*, podemos nos encontrar diante de uma tela cheia dos temidos resultados `nan`.

Felizmente, somos salvos pelo fato de que embora estejamos computando funções exponenciais, em última análise, pretendemos levar seu log (ao calcular a perda de entropia cruzada). Combinando esses dois operadores *softmax* e entropia cruzada juntos, podemos escapar dos problemas de estabilidade numérica que poderia nos atormentar durante a *backpropagation*. Conforme

mostrado na equação abaixo, evitamos calcular $\exp(o_j - \max(o_k))$ e podemos usar $o_j - \max(o_k)$ diretamente devido ao cancelamento em $\log(\exp(\cdot))$:

$$\begin{aligned}
\log{(\hat y_j)} & = \log\left(\frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}\right) \\
& = \log{(\exp(o_j - \max(o_k)))}-\log{\left(\sum_k \exp(o_k - \max(o_k)) \right)} \\
& = o_j - \max(o_k) -\log{\left(\sum_k \exp(o_k - \max(o_k)) \right)}.
\end{aligned} \tag{3.7.2}$$

Queremos manter a função *softmax* convencional acessível no caso de querermos avaliar as probabilidades de saída por nosso modelo. Mas em vez de passar probabilidades de *softmax* para nossa nova função de perda, nós vamos apenas passar os *logits* e calcular o *softmax* e seu log tudo de uma vez dentro da função de perda de entropia cruzada, que faz coisas inteligentes como o "Truque LogSumExp"[42].

```
loss = nn.CrossEntropyLoss()
```

3.7.3 Otimização do Algoritmo

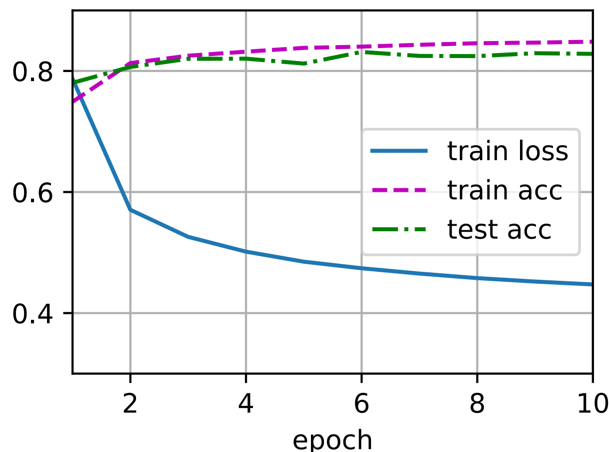
Aqui, nós usamos gradiente descendente estocástico de *minibatch* com uma taxa de aprendizado de 0,1 como o algoritmo de otimização. Observe que este é o mesmo que aplicamos no exemplo de regressão linear e ilustra a aplicabilidade geral dos otimizadores.

```
trainer = torch.optim.SGD(net.parameters(), lr=0.1)
```

3.7.4 Trainamento

Em seguida, chamamos a função de treinamento definida em Section 3.6 para treinar o modelo.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



[42] https://en.wikipedia.org/wiki/LogSumExp

Como antes, este algoritmo converge para uma solução que atinge uma precisão decente, embora desta vez com menos linhas de código do que antes.

3.7.5 Resumo

- Usando APIs de alto nível, podemos implementar a regressão *softmax* de forma muito mais concisa.
- De uma perspectiva computacional, a implementação da regressão *softmax* tem complexidades. Observe que, em muitos casos, uma estrutura de *deep learning* toma precauções adicionais além desses truques mais conhecidos para garantir a estabilidade numérica, salvando-nos de ainda mais armadilhas que encontraríamos se tentássemos codificar todos os nossos modelos do zero na prática.

3.7.6 Exercícios

1. Tente ajustar os hiperparâmetros, como *batch size*, número de épocas e taxa de aprendizado, para ver quais são os resultados.
2. Aumente o número de épocas de treinamento. Por que a precisão do teste pode diminuir depois de um tempo? Como poderíamos consertar isso?
3.

Discussions[43]

[43] https://discuss.d2l.ai/t/53

4 | *Perceptrons* Multicamada

Neste capítulo, apresentaremos sua primeira rede verdadeiramente *profunda*. As redes profundas mais simples são chamadas *perceptrons* multicamada, e eles consistem em várias camadas de neurônios cada um totalmente conectado àqueles na camada abaixo (do qual eles recebem contribuições) e aqueles acima (que eles, por sua vez, influenciam). Quando treinamos modelos de alta capacidade, corremos o risco de fazer *overfitting*. Portanto, precisaremos fornecer sua primeira introdução rigorosa às noções de *overfitting*, *underfitting* e seleção de modelo. Para ajudá-lo a combater esses problemas, apresentaremos técnicas de regularização, como redução do peso e abandono escolar. Também discutiremos questões relacionadas à estabilidade numérica e inicialização de parâmetros que são essenciais para o treinamento bem-sucedido de redes profundas. Durante todo o tempo, nosso objetivo é dar a você uma compreensão firme não apenas dos conceitos mas também da prática de usar redes profundas. No final deste capítulo, aplicamos o que apresentamos até agora a um caso real: a previsão do preço da casa. Nós examinamos questões relacionadas ao desempenho computacional, escalabilidade e eficiência de nossos modelos para os capítulos subsequentes.

4.1 *Perceptrons* Multicamada

Em Chapter 3,, apresentamos regressão *softmax* (Section 3.4), implementando o algoritmo do zero (Section 3.6) e usando APIs de alto nível (Section 3.7), e classificadores de treinamento para reconhecer 10 categorias de roupas a partir de imagens de baixa resolução. Ao longo do caminho, aprendemos como organizar dados, coagir nossos resultados em uma distribuição de probabilidade válida, aplicar uma função de perda apropriada, e minimizá-la em relação aos parâmetros do nosso modelo. Agora que dominamos essa mecânica no contexto de modelos lineares simples, podemos lançar nossa exploração de redes neurais profundas, a classe comparativamente rica de modelos com o qual este livro se preocupa principalmente.

4.1.1 Camadas Ocultas

Descrevemos a transformação afim em Section 3.1.1, que é uma transformação linear adicionada por um *bias. Para começar, relembre a arquitetura do modelo correspondendo ao nosso exemplo de regressão*softmax*, ilustrado em :numref:`fig_softmaxreg`. Este modelo mapeou nossas entradas diretamente para nossas saídas por meio de uma única transformação afim, seguido por uma operação*softmax. *Se nossos*labels* realmente estivessem relacionados aos nossos dados de entrada por uma transformação afim, então esta abordagem seria suficiente. Mas a linearidade nas transformações afins é uma suposição *forte*.

Modelos Lineares Podem Dar Errado

Por exemplo, linearidade implica a *mais fraca* suposição de *monotonicidade*: que qualquer aumento em nosso recurso deve sempre causar um aumento na saída do nosso modelo (se o peso correspondente for positivo), ou sempre causa uma diminuição na saída do nosso modelo (se o peso correspondente for negativo). Às vezes, isso faz sentido. Por exemplo, se estivéssemos tentando prever se um indivíduo vai pagar um empréstimo, podemos razoavelmente imaginar que mantendo tudo o mais igual, um candidato com uma renda maior sempre estaria mais propenso a retribuir do que um com uma renda mais baixa. Embora monotônico, esse relacionamento provavelmente não está linearmente associado à probabilidade de reembolso. Um aumento na receita de 0 a 50 mil provavelmente corresponde a um aumento maior em probabilidade de reembolso do que um aumento de 1 milhão para 1,05 milhão. Uma maneira de lidar com isso pode ser pré-processar nossos dados de forma que a linearidade se torne mais plausível, digamos, usando o logaritmo da receita como nosso recurso.

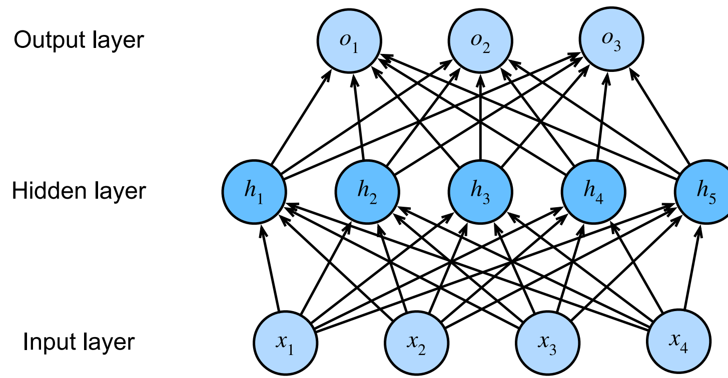
Observe que podemos facilmente encontrar exemplos que violam a monotonicidade. Digamos, por exemplo, que queremos prever a probabilidade de morte com base na temperatura corporal. Para indivíduos com temperatura corporal acima de 37 ° C (98,6 ° F), temperaturas mais altas indicam maior risco. No entanto, para indivíduos com temperatura corporal abaixo de 37 ° C, temperaturas mais altas indicam risco menor! Também neste caso, podemos resolver o problema com algum pré-processamento inteligente. Ou seja, podemos usar a distância de 37 ° C como nossa *feature*.

Mas que tal classificar imagens de cães e gatos? Aumentar a intensidade do pixel no local (13, 17) deveria sempre aumentar (ou sempre diminuir) a probabilidade de que a imagem retrate um cachorro? A confiança em um modelo linear corresponde à implícita suposição de que o único requisito para diferenciar gatos vs. cães é avaliar o brilho de pixels individuais. Esta abordagem está fadada ao fracasso em um mundo onde inverter uma imagem preserva a categoria.

E ainda, apesar do aparente absurdo da linearidade aqui, em comparação com nossos exemplos anteriores, é menos óbvio que poderíamos resolver o problema com uma correção de pré-processamento simples. Isso ocorre porque o significado de qualquer pixel depende de maneiras complexas de seu contexto (os valores dos pixels circundantes). Embora possa existir uma representação de nossos dados isso levaria em consideração as interações relevantes entre nossas características, no topo das quais um modelo linear seria adequado, nós simplesmente não sabemos como calculá-lo à mão. Com redes neurais profundas, usamos dados observacionais para aprender conjuntamente uma representação por meio de camadas ocultas e um preditor linear que atua sobre essa representação.

Incorporando Camadas Ocultas

Podemos superar essas limitações dos modelos lineares e lidar com uma classe mais geral de funções incorporando uma ou mais camadas ocultas. A maneira mais fácil de fazer isso é empilhar muitas camadas totalmente conectadas umas sobre as outras. Cada camada alimenta a camada acima dela, até gerarmos resultados. Podemos pensar nas primeiras $L - 1$ camadas como nossa representação e a camada final como nosso preditor linear. Esta arquitetura é comumente chamada um *perceptron multicamadas*, frequentemente abreviado como *MLP*. Abaixo, representamos um MLP em diagrama (`fig_mlp`).



.. _fig_mlp:

Este MLP tem 4 entradas, 3 saídas, e sua camada oculta contém 5 unidades ocultas. Uma vez que a camada de entrada não envolve nenhum cálculo, produzindo saídas com esta rede requer a implementação dos cálculos para as camadas ocultas e de saída; assim, o número de camadas neste MLP é 2. Observe que essas camadas estão totalmente conectadas. Cada entrada influencia cada neurônio na camada oculta, e cada um deles, por sua vez, influencia cada neurônio na camada de saída. No entanto, conforme sugerido por Section 3.4.3, o custo de parametrização de MLPs com camadas totalmente conectadas pode ser proibitivamente alto, o que pode motivar compensação entre o salvamento do parâmetro e a eficácia do modelo, mesmo sem alterar o tamanho de entrada ou saída (Zhang et al., 2021).

De Linear Para não Linear

Como antes, pela matriz $\mathbf{X} \in \mathbb{R}^{n \times d}$, denotamos um *minibatch* de n exemplos em que cada exemplo tem d entradas (*features*). Para um MLP de uma camada oculta, cuja camada oculta tem h unidades ocultas, denotamos por $\mathbf{H} \in \mathbb{R}^{n \times h}$ as saídas da camada oculta, que são *representações ocultas*. Em matemática ou código, $\mathbf{H}$ também é conhecido como uma *variável de camada oculta* ou uma *variável oculta*. Uma vez que as camadas ocultas e de saída estão totalmente conectadas, temos pesos de camada oculta $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ e *bias* $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ e pesos da camada de saída $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ e *bias* $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. Formalmente, calculamos as saídas $\mathbf{O} \in \mathbb{R}^{n \times q}$ do MLP de uma camada oculta da seguinte maneira:

$$\begin{aligned} \mathbf{H} & = \mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} & = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.1}$$

Observe que depois de adicionar a camada oculta, nosso modelo agora exige que rastreemos e atualizemos conjuntos adicionais de parâmetros. Então, o que ganhamos em troca? Você pode se surpreender ao descobrir que — no modelo definido acima — *nós não ganhamos nada pelos nossos problemas*! O motivo é claro. As unidades ocultas acima são fornecidas por uma função afim das entradas, e as saídas (pré-*softmax*) são apenas uma função afim das unidades ocultas. Uma função afim de uma função afim é em si uma função afim. Além disso, nosso modelo linear já era capaz de representar qualquer função afim.

Podemos ver a equivalência formalmente provando que para quaisquer valores dos pesos, podemos apenas recolher a camada oculta, produzindo um modelo de camada única equivalente com parâmetros $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X} \mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X} \mathbf{W} + \mathbf{b}. \tag{4.1.2}$$

Para perceber o potencial das arquiteturas multicamadas, precisamos de mais um ingrediente chave: a *função de ativação* não linear σ a ser aplicada a cada unidade oculta seguindo a transformação afim. As saídas das funções de ativação (por exemplo, $\sigma(\cdot)$) são chamadas de *ativações*. Em geral, com as funções de ativação implementadas, não é mais possível reduzir nosso MLP em um modelo linear:

$$\begin{aligned} \mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.3}$$

Uma vez que cada linha em $\mathbf{X}$ corresponde a um exemplo no *minibatch*, com algum abuso de notação, definimos a não linearidade σ para aplicar às suas entradas de uma forma em linha, ou seja, um exemplo de cada vez. Observe que usamos a notação para *softmax* da mesma forma para denotar uma operação nas linhas em Section 3.4.5. Frequentemente, como nesta seção, as funções de ativação que aplicamos a camadas ocultas não são apenas nas linhas, mas elemento a elemento. Isso significa que depois de calcular a parte linear da camada, podemos calcular cada ativação sem olhar para os valores assumidos pelas outras unidades ocultas. Isso é verdadeiro para a maioria das funções de ativação.

Para construir MLPs mais gerais, podemos continuar empilhando tais camadas escondidas, por exemplo, $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ e $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, uma sobre a outra, rendendo modelos cada vez mais expressivos.

Aproximadores Universais

MLPs podem capturar interações complexas entre nossas entradas por meio de seus neurônios ocultos, que dependem dos valores de cada uma das entradas. Podemos projetar nós ocultos facilmente para realizar cálculos arbitrários, por exemplo, operações lógicas básicas em um par de entradas. Além disso, para certas escolhas da função de ativação, é amplamente conhecido que os MLPs são aproximadores universais. Mesmo com uma rede de camada única oculta, dados nós suficientes (possivelmente absurdamente muitos deles), e o conjunto certo de pesos, podemos modelar qualquer função, embora realmente aprender essa função seja a parte difícil. Você pode pensar em sua rede neural como sendo um pouco como a linguagem de programação C. A linguagem, como qualquer outra linguagem moderna, é capaz de expressar qualquer programa computável. Mas, na verdade, criar um programa que atenda às suas especificações é a parte difícil.

Além disso, só porque uma rede de camada única oculta *pode* aprender qualquer função não significa que você deve tentar resolver todos os seus problemas com redes de camada única oculta. Na verdade, podemos aproximar muitas funções de forma muito mais compacta, usando redes mais profundas (em comparação com redes mais amplas). Trataremos de argumentos mais rigorosos nos capítulos subsequentes.

4.1.2 Funções de Ativação

As funções de ativação decidem se um neurônio deve ser ativado ou não por calcular a soma ponderada e adicionar ainda o *bias* com ela. Eles são operadores diferenciáveis para transformar sinais de entrada em saídas, enquanto a maioria deles adiciona não linearidade. Como as funções de ativação são fundamentais para o *deep learning*, vamos examinar brevemente algumas funções de ativação comuns.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

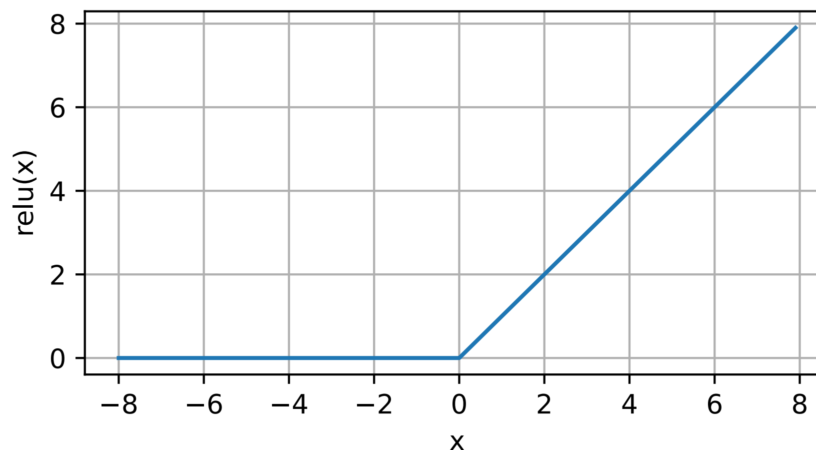
Função ReLU

A escolha mais popular, devido à simplicidade de implementação e seu bom desempenho em uma variedade de tarefas preditivas, é a *unidade linear retificada* (*ReLU*). ReLU fornece uma transformação não linear muito simples. Dado um elemento x, a função é definida como o máximo desse elemento e 0:

$$\mathrm{ReLU}(x) = \max(x, 0). \tag{4.1.4}$$

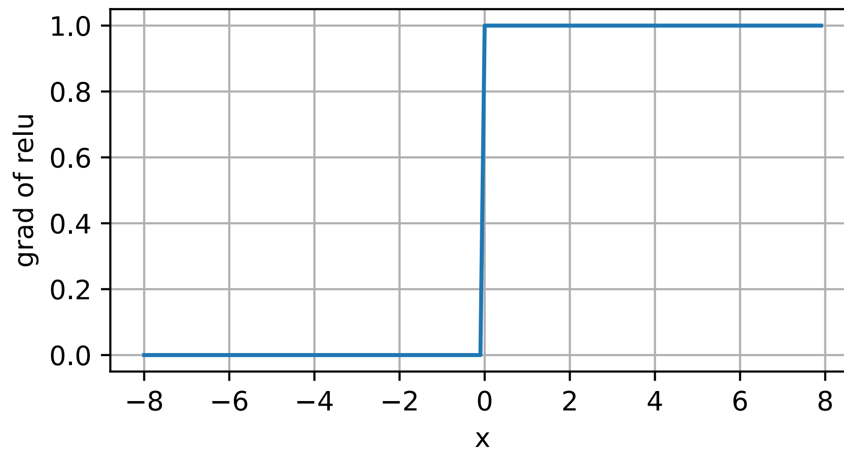
Informalmente, a função ReLU retém apenas elementos positivos e descarta todos os elementos negativos definindo as ativações correspondentes para 0. Para obter alguma intuição, podemos representar graficamente a função. Como você pode ver, a função de ativação é linear por partes.

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Quando a entrada é negativa, a derivada da função ReLU é 0, e quando a entrada é positiva, a derivada da função ReLU é 1. Observe que a função ReLU não é diferenciável quando a entrada assume um valor precisamente igual a 0. Nesses casos, o padrão é o lado esquerdo da derivada, e dizemos que a derivada é 0 quando a entrada é 0. Podemos escapar impunes porque a entrada pode nunca ser realmente zero. Há um velho ditado que diz que se as condições de contorno sutis são importantes, provavelmente estamos fazendo matemática (*real*), não engenharia. Essa sabedoria convencional pode se aplicar aqui. Plotamos a derivada da função ReLU abaixo.

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



A razão para usar ReLU é que suas derivadas são particularmente bem comportadas: ou elas desaparecem ou simplesmente deixam * argumento passar. Isso torna a otimização melhor comportada e mitiga o problema bem documentado de gradientes de desaparecimento que atormentaram versões anteriores de redes neurais (mais sobre isso mais tarde).

Observe que existem muitas variantes da função ReLU, incluindo a função *ReLU parametrizada* (*pReLU*) (He et al., 2015). Esta variação adiciona um termo linear ao ReLU, então algumas informações ainda são transmitidas, mesmo quando o argumento é negativo:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{4.1.5}$$

Função Sigmoid

A *função sigmoid* transforma suas entradas, para as quais os valores estão no domínio $\mathbb{R}$, para saídas que estão no intervalo (0, 1). Por esse motivo, o sigmoid é frequentemente chamada de *função de esmagamento*: ele esmaga qualquer entrada no intervalo (-inf, inf) para algum valor no intervalo (0, 1):

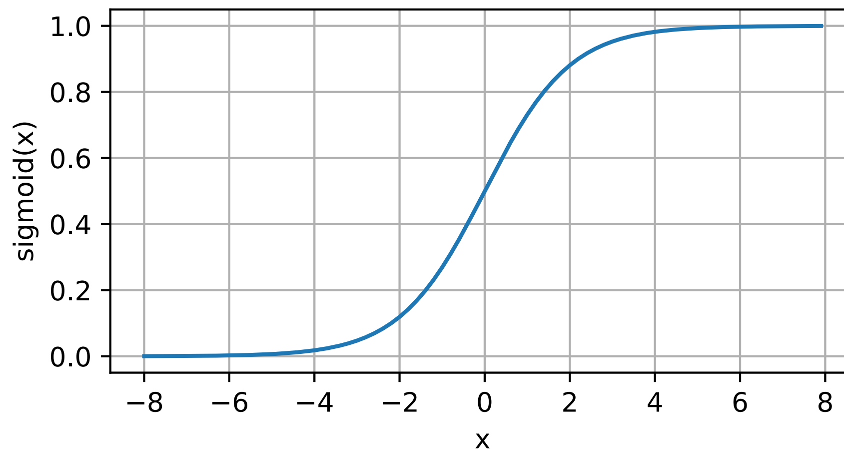
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.1.6}$$

Nas primeiras redes neurais, os cientistas estavam interessados em modelar neurônios biológicos que *disparam* ou *não disparam*. Assim, os pioneiros neste campo, voltando para McCulloch e Pitts, os inventores do neurônio artificial, focaram em unidades de limiar. Uma ativação de limite assume valor 0 quando sua entrada está abaixo de algum limite e valor 1 quando a entrada excede o limite.

Quando a atenção mudou para o aprendizado baseado em gradiente, a função sigmoid foi uma escolha natural porque é uma boa, diferenciável aproximação a uma unidade de limiar. Sigmoids ainda são amplamente usados como funções de ativação nas unidades de saída, quando queremos interpretar as saídas como probabilidades para problemas de classificação binária (você pode pensar na sigmoid como um caso especial do *softmax*). No entanto, a sigmoid foi quase toda substituído pelo ReLU, mais simples e facilmente treinável para mais uso em camadas ocultas. Em capítulos posteriores sobre redes neurais recorrentes, iremos descrever arquiteturas que alavancam unidades sigmoid para controlar o fluxo de informações ao longo do tempo.

Abaixo, traçamos a função sigmoid. Observe que quando a entrada está próxima de 0, a função sigmoid se aproxima uma transformação linear.

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

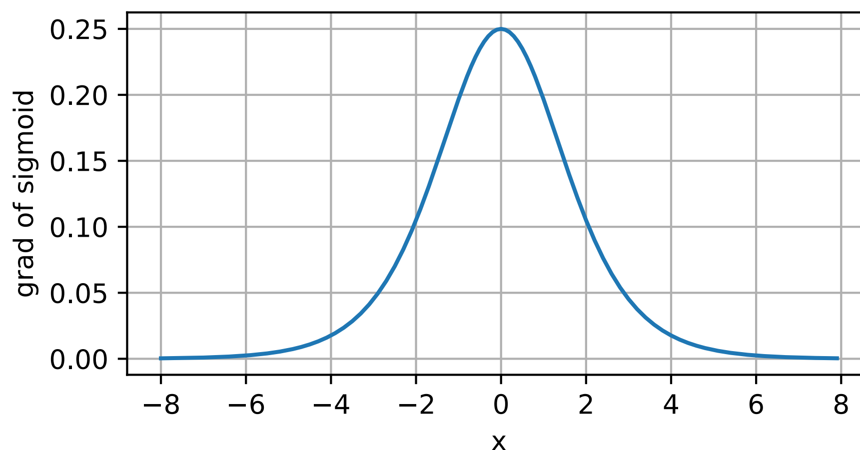


A derivada da função sigmoid é dada pela seguinte equação:

$$\frac{d}{dx} \operatorname{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \operatorname{sigmoid}(x) \left(1 - \operatorname{sigmoid}(x)\right). \tag{4.1.7}$$

A derivada da função sigmoid é plotada abaixo. Observe que quando a entrada é 0, a derivada da função sigmoid atinge um máximo de 0,25. À medida que a entrada diverge de 0 em qualquer direção, a derivada se aproxima de 0.

```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



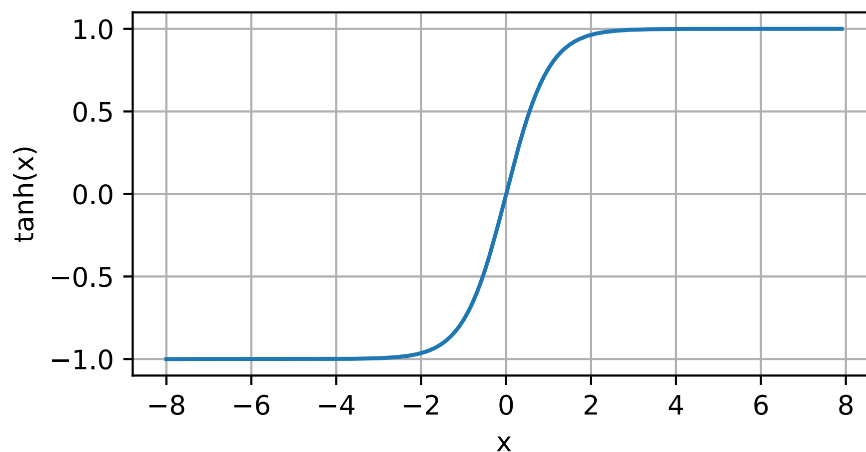
Função Tanh

Como a função sigmoid, a tanh (tangente hiperbólica) função também comprime suas entradas, transformando-as em elementos no intervalo entre -1 e 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{4.1.8}$$

Traçamos a função tanh abaixo. Observe que, à medida que a entrada se aproxima de 0, a função tanh se aproxima de uma transformação linear. Embora a forma da função seja semelhante à da função sigmoid, a função tanh exibe uma simetria de ponto sobre a origem do sistema de coordenadas.

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

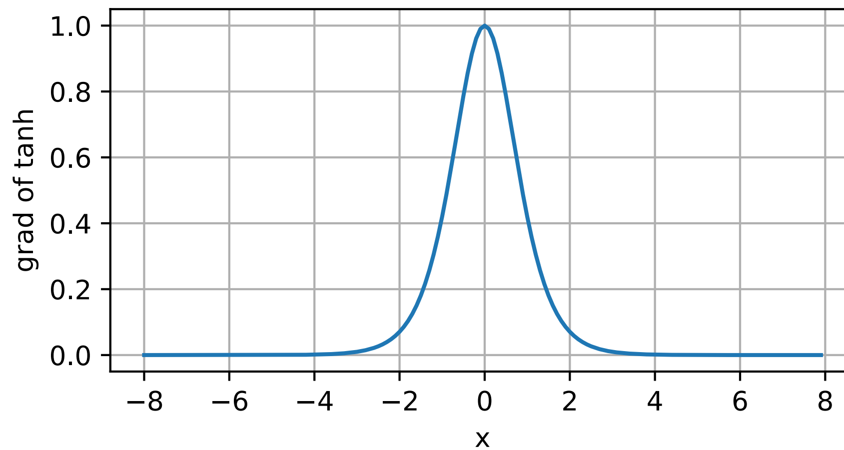


A derivada da função tanh é:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x). \tag{4.1.9}$$

A derivada da função tanh é plotada abaixo. Conforme a entrada se aproxima de 0, a derivada da função tanh se aproxima de um máximo de 1. E como vimos com a função sigmoid, conforme a entrada se afasta de 0 em qualquer direção, a derivada da função tanh se aproxima de 0.

```
# Clear out previous gradients.
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



Em resumo, agora sabemos como incorporar não linearidades para construir arquiteturas de rede neural multicamadas expressivas. Como uma nota lateral, o seu conhecimento já coloca você no comando de um kit de ferramentas semelhante para um praticante por volta de 1990. De certa forma, você tem uma vantagem sobre qualquer pessoa que trabalhou na década de 1990, porque você pode alavancar *frameworks* de *deep learning* de código aberto para construir modelos rapidamente, usando apenas algumas linhas de código. Anteriormente, o treinamento dessas redes pesquisadores obrigados a codificar milhares de linhas de C e Fortran.

4.1.3 Resumo

- O MLP adiciona uma ou várias camadas ocultas totalmente conectadas entre as camadas de saída e de entrada e transforma a saída da camada oculta por meio de uma função de ativação.
- As funções de ativação comumente usadas incluem a função ReLU, a função sigmoid e a função tanh.

4.1.4 Exercícios

1. Calcule a derivada da função de ativação pReLU.
2. Mostre que um MLP usando apenas ReLU (ou pReLU) constrói uma função linear contínua por partes.
3. Mostre que $\operatorname{tanh}(x) + 1 = 2 \operatorname{sigmoid}(2x)$.
4. Suponha que temos uma não linearidade que se aplica a um *minibatch* por vez. Que tipo de problemas você espera que isso cause?

Discussions[44]

[44] https://discuss.d2l.ai/t/91

4.2 Implementação de Perceptrons Multicamadas do Zero

Agora que caracterizamos perceptrons multicamadas (MLPs) matematicamente, vamos tentar implementar um nós mesmos. Para comparar com nossos resultados anteriores alcançado com regressão *softmax* (Section 3.6), vamos continuar a trabalhar com o conjunto de dados de classificação de imagens Fashion-MNIST (Section 3.5).

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.2.1 Inicializando os Parâmetros do Modelo

Lembre-se de que o Fashion-MNIST contém 10 classes, e que cada imagem consiste em uma graded $28 \times 28 = 784$ de valores de pixel em tons de cinza. Novamente, vamos desconsiderar a estrutura espacial entre os pixels por enquanto, então podemos pensar nisso simplesmente como um conjunto de dados de classificação com 784 características de entrada e 10 classes. Para começar, iremos implementar um MLP com uma camada oculta e 256 unidades ocultas. Observe que podemos considerar essas duas quantidades como hiperparâmetros. Normalmente, escolhemos larguras de camada em potências de 2, que tendem a ser computacionalmente eficientes porque de como a memória é alocada e endereçada no hardware.

Novamente, iremos representar nossos parâmetros com vários tensores. Observe que *para cada camada*, devemos acompanhar uma matriz de ponderação e um vetor de polarização. Como sempre, alocamos memória para os gradientes da perda com relação a esses parâmetros.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = nn.Parameter(torch.randn(
    num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(torch.randn(
    num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

4.2.2 Função de Ativação

Para ter certeza de que sabemos como tudo funciona, iremos implementar a ativação ReLU nós mesmos usar a função máxima em vez de invocar a função embutida `relu` diretamente.

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

4.2.3 Modelo

Porque estamos desconsiderando a estrutura espacial, nós `remodelamos` cada imagem bidimensional em um vetor plano de comprimento `num_inputs`. Finalmente, nós implementamos nosso modelo com apenas algumas linhas de código.

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # Here '@' stands for matrix multiplication
    return (H@W2 + b2)
```

4.2.4 Função de Perda

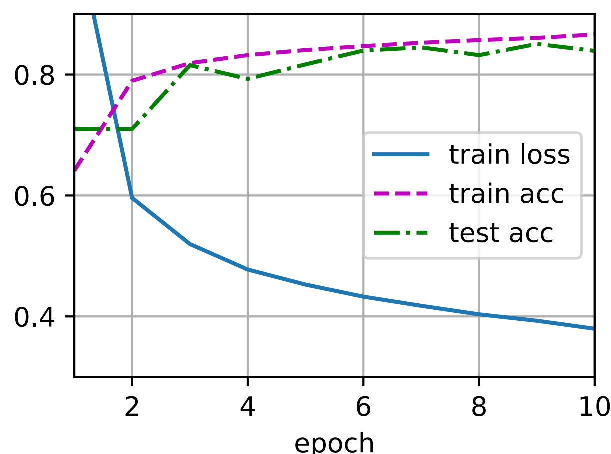
Para garantir estabilidade numérica, e porque já implementamos a função* softmax* do zero (Section 3.6), alavancamos a função integrada de APIs de alto nível para calcular o *softmax* e a perda de entropia cruzada. Lembre-se de nossa discussão anterior sobre essas complexidades em Section 3.7.2. Nós encorajamos o leitor interessado a examinar o código-fonte para a função de perda para aprofundar seu conhecimento dos detalhes de implementação.

```
loss = nn.CrossEntropyLoss()
```

4.2.5 Trainamento

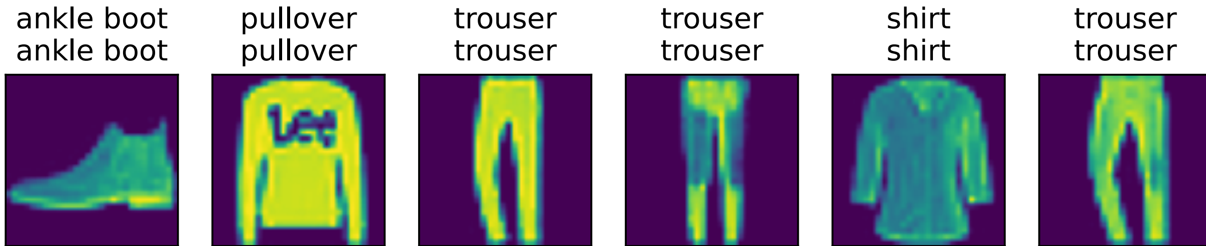
Felizmente, o loop de treinamento para MLPs é exatamente igual à regressão *softmax*. Aproveitando o pacote `d2l` novamente, chamamos a função `train_ch3` (ver Section 3.6), definindo o número de épocas para 10 e a taxa de aprendizagem para 0,1.

```
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```



Para avaliar o modelo aprendido, nós aplicamos em alguns dados de teste.

```
d2l.predict_ch3(net, test_iter)
```



4.2.6 Resumo

- Vimos que implementar um MLP simples é fácil, mesmo quando feito manualmente.
- No entanto, com um grande número de camadas, implementar MLPs do zero ainda pode ser complicado (por exemplo, nomear e controlar os parâmetros do nosso modelo).

4.2.7 Exercícios

1. Altere o valor do hiperparâmetro `num_hiddens` e veja como esse hiperparâmetro influencia seus resultados. Determine o melhor valor deste hiperparâmetro, mantendo todos os outros constantes.
2. Experimente adicionar uma camada oculta adicional para ver como isso afeta os resultados.
3. Como mudar a taxa de aprendizado altera seus resultados? Corrigindo a arquitetura do modelo e outros hiperparâmetros (incluindo o número de épocas), qual taxa de aprendizado oferece os melhores resultados?
4. Qual é o melhor resultado que você pode obter otimizando todos os hiperparâmetros (taxa de aprendizagem, número de épocas, número de camadas ocultas, número de unidades ocultas por camada) em conjunto?
5. Descreva por que é muito mais difícil lidar com vários hiperparâmetros.
6. Qual é a estratégia mais inteligente que você pode imaginar para estruturar uma pesquisa em vários hiperparâmetros?

Discussions[45]

[45] https://discuss.d2l.ai/t/93

4.3 Implementação Concisa de *Perceptrons* Multicamadas

As you might expect, by relying on the high-level APIs, we can implement MLPs even more concisely.

```
import torch
from torch import nn
from d2l import torch as d2l
```

4.3.1 Modelo

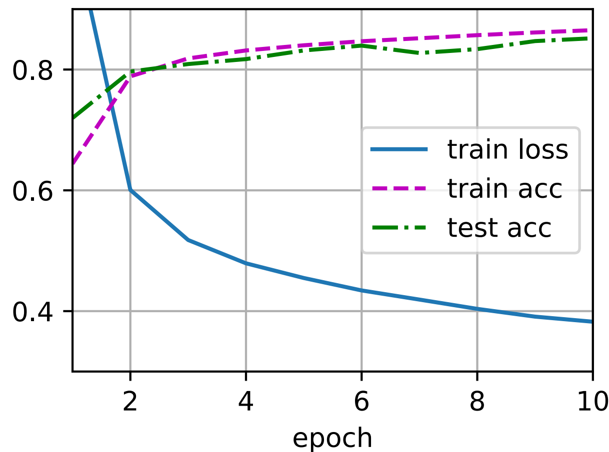
Em comparação com nossa implementação concisa de implementação de regressão *softmax* (Section 3.7), a única diferença é que adicionamos *duas* camadas totalmente conectadas (anteriormente, adicionamos *uma*). A primeira é nossa camada oculta, que contém 256 unidades ocultas e aplica a função de ativação ReLU. A segunda é nossa camada de saída.

```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

O loop de treinamento é exatamente o mesmo como quando implementamos a regressão *softmax*. Essa modularidade nos permite separar questões relativas à arquitetura do modelo a partir de considerações ortogonais.

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



4.3.2 Resumo

- Usando APIs de alto nível, podemos implementar MLPs de forma muito mais concisa.
- Para o mesmo problema de classificação, a implementação de um MLP é a mesma da regressão *softmax*, exceto para camadas ocultas adicionais com funções de ativação.

4.3.3 Exercícios

1. Tente adicionar diferentes números de camadas ocultas (você também pode modificar a taxa de aprendizagem). Qual configuração funciona melhor?
2. Experimente diferentes funções de ativação. Qual funciona melhor?
3. Experimente diferentes esquemas para inicializar os pesos. Qual método funciona melhor?

Discussions[46]

4.4 Seleção do Modelo, *Underfitting*, e *Overfitting*

Como cientistas de *machine learning*, nosso objetivo é descobrir *padrões*. Mas como podemos ter certeza de que realmente descobrimos um padrão *geral* e não simplesmente memorizamos nossos dados? Por exemplo, imagine que queremos caçar padrões entre marcadores genéticos ligando os pacientes ao seu estado de demência, onde os rótulos são retirados do conjunto $\{\text{dementia}, \text{mild cognitive impairment}, \text{healthy}\}$ Como os genes de cada pessoa os identificam de forma única (ignorando irmãos idênticos), é possível memorizar todo o conjunto de dados.

Não queremos que nosso modelo diga *"É o Bob! Lembro-me dele! Ele tem demência!"* O motivo é simples. Quando implantamos o modelo no futuro, nós encontraremos pacientes que o modelo nunca viu antes. Nossas previsões só serão úteis se nosso modelo realmente descobriu um padrão *geral*.

Para recapitular mais formalmente, nosso objetivo é descobrir padrões que capturam regularidades na população subjacente da qual nosso conjunto de treinamento foi extraído. Se tivermos

[46] https://discuss.d2l.ai/t/95

sucesso neste empreendimento, então poderíamos avaliar com sucesso o risco mesmo para indivíduos que nunca encontramos antes. Este problema — como descobrir padrões que *generalizam* — é o problema fundamental do *machine learning*.

O perigo é que, quando treinamos modelos, acessamos apenas uma pequena amostra de dados. Os maiores conjuntos de dados de imagens públicas contêm cerca de um milhão de imagens. Mais frequentemente, devemos aprender com apenas milhares ou dezenas de milhares de exemplos de dados. Em um grande sistema hospitalar, podemos acessar centenas de milhares de registros médicos. Ao trabalhar com amostras finitas, corremos o risco de poder descobrir associações aparentes que acabam não se sustentando quando coletamos mais dados.

O fenômeno de ajustar nossos dados de treinamento mais precisamente do que ajustamos, a distribuição subjacente é chamada de *overfitting*, e as técnicas usadas para combater o *overfitting* são chamadas de *regularização*. Nas seções anteriores, você deve ter observado esse efeito durante a experiência com o conjunto de dados *Fashion-MNIST*. Se você alterou a estrutura do modelo ou os hiperparâmetros durante o experimento, deve ter notado que, com neurônios, camadas e períodos de treinamento suficientes, o modelo pode eventualmente atingir uma precisão perfeita no conjunto de treinamento, mesmo quando a precisão dos dados de teste se deteriora.

4.4.1 Erro de Treinamento e Erro de Generalização

Para discutir este fenômeno de forma mais formal, precisamos diferenciar entre erro de treinamento e erro de generalização. O *erro de treinamento* é o erro do nosso modelo conforme calculado no conjunto de dados de treinamento, enquanto *erro de generalização* é a expectativa do erro do nosso modelo deveríamos aplicá-lo a um fluxo infinito de exemplos de dados adicionais extraído da mesma distribuição de dados subjacente que nossa amostra original.

De forma problemática, nunca podemos calcular o erro de generalização com exatidão. Isso ocorre porque o fluxo de dados infinitos é um objeto imaginário. Na prática, devemos *estimar* o erro de generalização aplicando nosso modelo a um conjunto de teste independente constituído de uma seleção aleatória de exemplos de dados que foram retirados de nosso conjunto de treinamento.

Os três experimentos mentais a seguir ajudarão a ilustrar melhor esta situação. Considere um estudante universitário tentando se preparar para o exame final. Um aluno diligente se esforçará para praticar bem e testar suas habilidades usando exames de anos anteriores. No entanto, um bom desempenho em exames anteriores não é garantia que ele se sobressairá quando for importante. Por exemplo, o aluno pode tentar se preparar aprendendo de cor as respostas às questões do exame. Isso requer que o aluno memorize muitas coisas. Ela pode até se lembrar das respostas de exames anteriores perfeitamente. Outro aluno pode se preparar tentando entender as razões para dar certas respostas. Na maioria dos casos, o último aluno se sairá muito melhor.

Da mesma forma, considere um modelo que simplesmente usa uma tabela de pesquisa para responder às perguntas. Se o conjunto de entradas permitidas for discreto e razoavelmente pequeno, talvez depois de ver *muitos* exemplos de treinamento, essa abordagem teria um bom desempenho. Ainda assim, esse modelo não tem capacidade de fazer melhor do que adivinhação aleatória quando confrontado com exemplos que nunca viu antes. Na realidade, os espaços de entrada são muito grandes para memorizar as respostas correspondentes a cada entrada concebível. Por exemplo, considere as imagens 28×28 em preto e branco. Se cada pixel pode ter um entre 256 valores de tons de cinza, então há 256^{784} imagens possíveis. Isso significa que há muito mais imagens em miniatura em escala de cinza de baixa resolução do que átomos no universo. Mesmo se pudéssemos encontrar esses dados, nunca poderíamos nos dar ao luxo de armazenar a tabela de pesquisa.

Por último, considere o problema de tentar classificar os resultados dos lançamentos de moeda (classe 0: cara, classe 1: coroa) com base em alguns recursos contextuais que podem estar disponíveis. Suponha que a moeda seja justa. Não importa o algoritmo que criamos, o erro de generalização sempre será $\frac{1}{2}$. No entanto, para a maioria dos algoritmos, devemos esperar que nosso erro de treinamento seja consideravelmente menor, dependendo da sorte do sorteio, mesmo se não tivéssemos nenhuma *feature*! Considere o conjunto de dados {0, 1, 1, 1, 0, 1}. Nosso algoritmo sem recursos teria que recorrer sempre à previsão da *classe majoritária*, que parece ser *1* em nossa amostra limitada. Neste caso, o modelo que sempre prevê a classe 1 incorrerá em um erro de $\frac{1}{3}$, consideravelmente melhor do que nosso erro de generalização. Conforme aumentamos a quantidade de dados, a probabilidade de que a fração de caras irá se desviar significativamente de $\frac{1}{2}$ diminui, e nosso erro de treinamento viria a corresponder ao erro de generalização.

Teoria de Aprendizagem Estatística

Como a generalização é o problema fundamental no *machine learning*, você pode não se surpreender ao aprender que muitos matemáticos e teóricos dedicaram suas vidas para desenvolver teorias formais para descrever este fenômeno. Em seu teorema de mesmo nome[47], Glivenko e Cantelli derivaram a taxa na qual o erro de treinamento converge para o erro de generalização. Em uma série de artigos seminais, Vapnik e Chervonenkis[48] estenderam esta teoria a classes de funções mais gerais. Este trabalho lançou as bases da teoria da aprendizagem estatística.

No ambiente de aprendizagem supervisionada padrão, que abordamos até agora e manteremos ao longo da maior parte deste livro, presumimos que tanto os dados de treinamento quanto os dados de teste são desenhados *independentemente* de distribuições *idênticas*. Isso é comumente chamado de *suposição i.i.d.*, o que significa que o processo que faz a amostragem de nossos dados não tem memória. Em outras palavras, o segundo exemplo desenhado e o terceiro desenhado não são mais correlacionados do que a segunda e a segunda milionésima amostra extraída.

Ser um bom cientista de *machine learning* exige pensar criticamente, e você já deve estar cutucando buracos nessa suposição, surgindo com casos comuns em que a suposição falha. E se treinarmos um preditor de risco de mortalidade em dados coletados de pacientes no UCSF Medical Center, e aplicá-lo em pacientes no *Massachusetts General Hospital*? Essas distribuições simplesmente não são idênticas. Além disso, os empates podem ser correlacionados no tempo. E se estivermos classificando os tópicos dos Tweets? O ciclo de notícias criaria dependências temporais nos tópicos em discussão, violando quaisquer pressupostos de independência.

Às vezes, podemos escapar impunes de violações menores da suposição i.i.d. e nossos modelos continuarão a funcionar muito bem. Afinal, quase todos os aplicativos do mundo real envolvem pelo menos alguma violação menor da suposição i.i.d., e ainda temos muitas ferramentas úteis para várias aplicações, como reconhecimento de rosto, reconhecimento de voz e tradução de idiomas.

Outras violações certamente causarão problemas. Imagine, por exemplo, se tentarmos treinar um sistema de reconhecimento de rosto treinando-o exclusivamente em estudantes universitários e então tentar implantá-lo como uma ferramenta para monitorar a geriatria em uma população de lares de idosos. É improvável que funcione bem, uma vez que estudantes universitários tendem a parecer consideravelmente diferentes dos idosos.

Nos capítulos subsequentes, discutiremos problemas decorrentes de violações da suposição i.i.d.. Por enquanto, mesmo tomando a suposição i.i.d. como certa, compreender a generalização é um

[47] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem
[48] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory

problema formidável. Além disso, elucidando os fundamentos teóricos precisos isso que podem explicar por que redes neurais profundas generalizam tão bem como o fazem, continua a irritar as maiores mentes da teoria do aprendizado.

Quando treinamos nossos modelos, tentamos pesquisar uma função que se ajusta aos dados de treinamento da melhor maneira possível. Se a função é tão flexível que pode pegar padrões falsos tão facilmente quanto às associações verdadeiras, então ele pode funcionar *muito bem* sem produzir um modelo que generaliza bem para dados invisíveis. Isso é exatamente o que queremos evitar ou pelo menos controlar. Muitas das técnicas de aprendizado profundo são heurísticas e truques visando a proteção contra *overfitting*.

Complexidade do Modelo

Quando temos modelos simples e dados abundantes, esperamos que o erro de generalização seja semelhante ao erro de treinamento. Quando trabalhamos com modelos mais complexos e menos exemplos, esperamos que o erro de treinamento diminua, mas a lacuna de generalização cresça. O que constitui precisamente a complexidade do modelo é um assunto complexo. Muitos fatores determinam se um modelo irá generalizar bem. Por exemplo, um modelo com mais parâmetros pode ser considerado mais complexo. Um modelo cujos parâmetros podem ter uma gama mais ampla de valores pode ser mais complexo. Muitas vezes, com redes neurais, pensamos em um modelo que exige mais iterações de treinamento quanto mais complexas, e um sujeito a *parada antecipada* (menos iterações de treinamento) como menos complexo.

Pode ser difícil comparar a complexidade entre os membros de classes de modelo substancialmente diferentes (digamos, árvores de decisão vs. redes neurais). Por enquanto, uma regra prática simples é bastante útil: um modelo que pode facilmente explicar fatos arbitrários é o que os estatísticos consideram complexo, ao passo que aquele que tem apenas um poder expressivo limitado mas ainda consegue explicar bem os dados provavelmente está mais perto da verdade. Em filosofia, isso está intimamente relacionado ao critério de falseabilidade de Popper de uma teoria científica: uma teoria é boa se ela se encaixa nos dados e se existem testes específicos que podem ser usados para contestá-lo. Isso é importante, pois toda estimativa estatística é *post hoc*, ou seja, estimamos depois de observar os fatos, portanto, vulnerável à falácia associada. Por enquanto, deixaremos a filosofia de lado e nos limitaremos a questões mais tangíveis.

Nesta seção, para lhe dar alguma intuição, vamos nos concentrar em alguns fatores que tendem para influenciar a generalização de uma classe de modelo:

1. O número de parâmetros ajustáveis. Quando o número de parâmetros ajustáveis, às vezes chamados de *graus de liberdade*, é grande, os modelos tendem a ser mais suscetíveis a *overfitting*.
2. Os valores assumidos pelos parâmetros. Quando os pesos podem assumir uma faixa mais ampla de valores, os modelos podem ser mais suscetíveis a *overfitting*.
3. O número de exemplos de treinamento. É trivialmente fácil fazer *overfitting* em um conjunto de dados contendo apenas um ou dois exemplos, mesmo se seu modelo for simples. Mas ajustar um conjunto de dados com milhões de exemplos requer um modelo extremamente flexível.

4.4.2 Seleção do Modelo

No *machine learning*, geralmente selecionamos nosso modelo final depois de avaliar vários modelos candidatos. Este processo é denominado *seleção de modelo*. Às vezes, os modelos que estão sujeitos a comparação são fundamentalmente diferentes em natureza (digamos, árvores de decisão vs. modelos lineares). Em outras ocasiões, estamos comparando membros da mesma classe de modelos que foram treinados com diferentes configurações de hiperparâmetros.

Com MLPs, por exemplo, podemos desejar comparar modelos com diferentes números de camadas ocultas, diferentes números de unidades ocultas, e várias opções das funções de ativação aplicado a cada camada oculta. A fim de determinar o melhor entre nossos modelos candidatos, normalmente empregaremos um conjunto de dados de validação.

Dataset de Validação

Em princípio, não devemos tocar em nosso conjunto de teste até depois de termos escolhido todos os nossos hiperparâmetros. Se usarmos os dados de teste no processo de seleção do modelo, existe o risco de ajustarmos demais os dados de teste (*overfitting*). Então, estaríamos em sérios problemas. Se ajustarmos demais nossos dados de treinamento, há sempre a avaliação dos dados de teste para nos manter honestos. Mas se ajustarmos demais os dados de teste, como saberemos?

Portanto, nunca devemos confiar nos dados de teste para a seleção do modelo. E ainda não podemos confiar apenas nos dados de treinamento para seleção de modelo porque não podemos estimar o erro de generalização nos próprios dados que usamos para treinar o modelo.

Em aplicações práticas, a imagem fica mais turva. Embora o ideal seja tocar nos dados de teste apenas uma vez, para avaliar o melhor modelo ou para comparar um pequeno número de modelos entre si, dados de teste do mundo real raramente são descartados após apenas um uso. Raramente podemos pagar um novo conjunto de teste para cada rodada de experimentos.

A prática comum para resolver este problema é dividir nossos dados de três maneiras, incorporando um *dataset de validação* (ou *conjunto de validação*) além dos conjuntos de dados de treinamento e teste. O resultado é uma prática obscura onde os limites entre a validação e os dados de teste são preocupantemente ambíguos. A menos que seja explicitamente declarado de outra forma, nos experimentos deste livro estamos realmente trabalhando com o que deveria ser corretamente chamado dados de treinamento e dados de validação, sem conjuntos de teste verdadeiros. Portanto, a precisão relatada em cada experimento do livro é realmente a precisão da validação e não uma precisão do conjunto de teste verdadeiro.

Validação Cruzada *:math:`K`-Fold*

Quando os dados de treinamento são escassos, podemos nem mesmo ser capazes de resistir dados suficientes para constituir um conjunto de validação adequado. Uma solução popular para este problema é empregar $validaocruzadaK$*-fold*. Aqui, os dados de treinamento originais são divididos em subconjuntos não sobrepostos de K. Então, o treinamento e a validação do modelo são executados K vezes, cada vez treinando em subconjuntos $K - 1$ e validando em um subconjunto diferente (aquele não usado para treinamento nessa rodada). Finalmente, os erros de treinamento e validação são estimados calculando a média dos resultados dos experimentos de K.

4.4.3 *Underfitting* ou *Overfitting*?

Quando comparamos os erros de treinamento e validação, queremos estar atentos a duas situações comuns. Primeiro, queremos estar atentos aos casos quando nosso erro de treinamento e erro de validação são substanciais mas há uma pequena lacuna entre eles. Se o modelo não for capaz de reduzir o erro de treinamento, isso pode significar que nosso modelo é muito simples (ou seja, insuficientemente expressivo) para capturar o padrão que estamos tentando modelar. Além disso, uma vez que a *lacuna de generalização* entre nossos erros de treinamento e validação é pequena, temos motivos para acreditar que poderíamos sair impunes de um modelo mais complexo. Este fenômeno é conhecido como *underfitting*.

Por outro lado, como discutimos acima, queremos estar atentos aos casos quando nosso erro de treinamento é significativamente menor do que o nosso erro de validação, indicando *overfitting* severo. Observe que o *overfitting* nem sempre é uma coisa ruim. Especialmente no aprendizado profundo, é bem conhecido que os melhores modelos preditivos frequentemente executam muito melhor em dados de treinamento do que em dados de validação. Em última análise, geralmente nos preocupamos mais com o erro de validação do que sobre a lacuna entre os erros de treinamento e validação.

Se fazemos *underfitting* ou *overfitting* pode depender tanto na complexidade do nosso modelo e o tamanho dos conjuntos de dados de treinamento disponíveis, dois tópicos que discutiremos a seguir.

Complexidade do Modelo

Para ilustrar alguma intuição clássica sobre *overfitting* e complexidade do modelo, damos um exemplo usando polinômios. Dados de treinamento dados consistindo em uma única *feature* x e um rótulo de valor real correspondente y, tentamos encontrar o polinômio de grau d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{4.4.1}$$

para estimar os rótulos y. Este é apenas um problema de regressão linear onde nossos recursos são dados pelos poderes de x, os pesos do modelo são dados por w_i, e o *bias* é dado por w_0 visto que $x^0 = 1$ para todo x. Uma vez que este é apenas um problema de regressão linear, podemos usar o erro quadrático como nossa função de perda.

Uma função polinomial de ordem superior é mais complexa do que uma função polinomial de ordem inferior, uma vez que o polinômio de ordem superior tem mais parâmetros e a faixa de seleção da função do modelo é mais ampla. Corrigindo o conjunto de dados de treinamento, funções polinomiais de ordem superior devem sempre alcançar menor (na pior das hipóteses, igual) erro de treinamento em relação a polinômios de grau inferior. Na verdade, sempre que os exemplos de dados cada um tem um valor distinto de x, uma função polinomial com grau igual ao número de exemplos de dados pode se encaixar perfeitamente no conjunto de treinamento. Nós visualizamos a relação entre o grau polinomial e *underfitting* vs. *overfitting* em :numref: `fig_capacity_vs_error`.

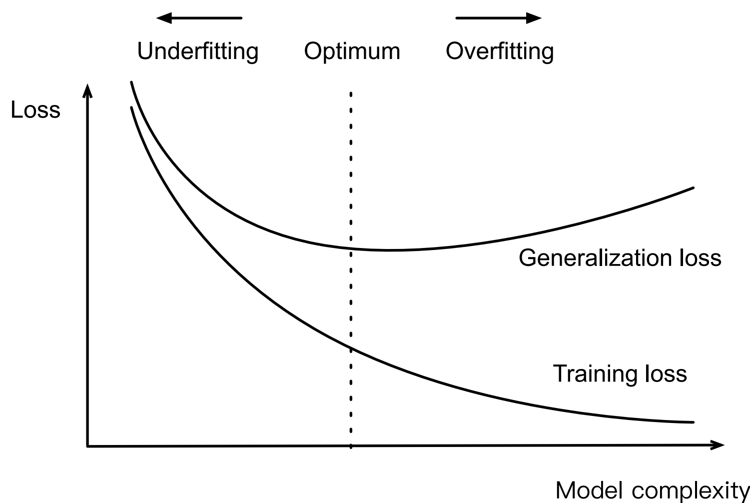


Fig. 4.4.1: Influence of model complexity on underfitting and overfitting

Tamanho do Dataset

A outra grande consideração a ter em mente é o tamanho do *dataset*. Corrigindo nosso modelo, menos amostras temos no *dataset* de treinamento, o mais provável (e mais severamente) é encontrar *overfitting*. À medida que aumentamos a quantidade de dados de treinamento, o erro de generalização geralmente diminui. Além disso, em geral, mais dados nunca fazem mal. Para uma tarefa fixa e distribuição de dados, normalmente existe uma relação entre a complexidade do modelo e o tamanho do *dataset*. Com mais dados, podemos tentar, de maneira lucrativa, ajustar um modelo mais complexo. Na ausência de dados suficientes, os modelos mais simples podem ser mais difíceis de superar. Para muitas tarefas, o aprendizado profundo supera apenas os modelos lineares quando muitos milhares de exemplos de treinamento estão disponíveis. Em parte, o sucesso atual do aprendizado profundo deve-se à atual abundância de enormes conjuntos de dados devido a empresas de Internet, armazenamento barato, dispositivos conectados, e a ampla digitalização da economia.

4.4.4 Regressão Polinomial

Agora podemos explorar esses conceitos interativamente ajustando polinômios aos dados.

```
import math
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

Gerando o *Dataset*

Primeiro, precisamos de dados. Dado x, iremos usar o seguinte polinômio cúbico para gerar os rótulos nos dados de treinamento e teste:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1^2). \tag{4.4.2}$$

O termo de ruído ϵ obedece a uma distribuição normal com uma média de 0 e um desvio padrão de 0,1. Para otimização, normalmente queremos evitar valores muito grandes de gradientes ou perdas. É por isso que as *features* são redimensionadas de x^i to $\frac{x^i}{i!}$. Isso nos permite evitar valores muito grandes para grandes expoentes i. Vamos sintetizar 100 amostras cada para o conjunto de treinamento e o conjunto de teste.

```
max_degree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(max_degree)  # Allocate lots of empty space
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
np.random.shuffle(features)
poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
for i in range(max_degree):
    poly_features[:, i] /= math.gamma(i + 1)  # `gamma(n)` = (n-1)!
# Shape of `labels`: (`n_train` + `n_test`,)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

Novamente, monômios armazenados em `poly_features` são redimensionados pela função gama, onde $\Gamma(n) = (n-1)!$. Dê uma olhada nas 2 primeiras amostras do conjunto de dados gerado. O valor 1 é tecnicamente uma *feature*, ou seja, a *feature* constante correspondente ao *bias*.

```
# Convert from NumPy ndarrays to tensors
true_w, features, poly_features, labels = [torch.tensor(x, dtype=
    torch.float32) for x in [true_w, features, poly_features, labels]]
```

```
features[:2], poly_features[:2, :], labels[:2]
```

```
(tensor([[-1.0036],
         [-0.1484]]),
 tensor([[ 1.0000e+00, -1.0036e+00,  5.0366e-01, -1.6850e-01,  4.2278e-02,
          -8.4865e-03,  1.4196e-03, -2.0354e-04,  2.5535e-05, -2.8476e-06,
           2.8580e-07, -2.6076e-08,  2.1810e-09, -1.6838e-10,  1.2071e-11,
          -8.0766e-13,  5.0663e-14, -2.9911e-15,  1.6678e-16, -8.8098e-18],
         [ 1.0000e+00, -1.4841e-01,  1.1013e-02, -5.4484e-04,  2.0215e-05,
          -6.0004e-07,  1.4842e-08, -3.1468e-10,  5.8379e-12, -9.6269e-14,
           1.4288e-15, -1.9277e-17,  2.3841e-19, -2.7218e-21,  2.8854e-23,
          -2.8549e-25,  2.6481e-27, -2.3119e-29,  1.9062e-31, -1.4889e-33]]),
 tensor([1.3268, 4.6359]))
```

Trainando e Testando o Modelo

Vamos primeiro implementar uma função para avaliar a perda em um determinado conjunto de dados.

```
def evaluate_loss(net, data_iter, loss):  #@save
    """Evaluate the loss of a model on the given dataset."""
    metric = d2l.Accumulator(2)  # Sum of losses, no. of examples
    for X, y in data_iter:
        out = net(X)
        y = y.reshape(out.shape)
        l = loss(out, y)
        metric.add(l.sum(), l.numel())
    return metric[0] / metric[1]
```

Agora definimos a função de treinamento.

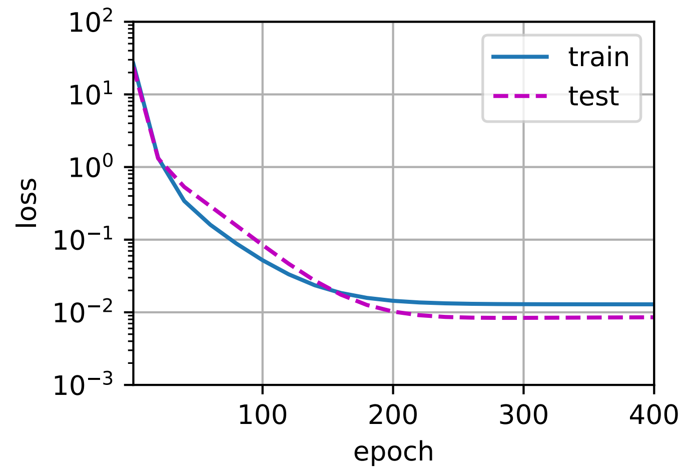
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=400):
    loss = nn.MSELoss()
    input_shape = train_features.shape[-1]
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net = nn.Sequential(nn.Linear(input_shape, 1, bias=False))
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels.reshape(-1,1)),
                                batch_size)
    test_iter = d2l.load_array((test_features, test_labels.reshape(-1,1)),
                               batch_size, is_train=False)
    trainer = torch.optim.SGD(net.parameters(), lr=0.01)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (evaluate_loss(net, train_iter, loss),
                                     evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data.numpy())
```

Ajuste de Função Polinomial de Terceira Ordem (Normal)

Começaremos usando primeiro uma função polinomial de terceira ordem, que é a mesma ordem da função de geração de dados. Os resultados mostram que as perdas de treinamento e teste deste modelo podem ser reduzidas de forma eficaz. Os parâmetros do modelo aprendido também estão próximos dos valores verdadeiros $w = [5, 1.2, -3.4, 5.6]$.

```
# Pick the first four dimensions, i.e., 1, x, x^2/2!, x^3/3! from the
# polynomial features
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 4.9941854  1.1942908 -3.3794312  5.6211023]]
```

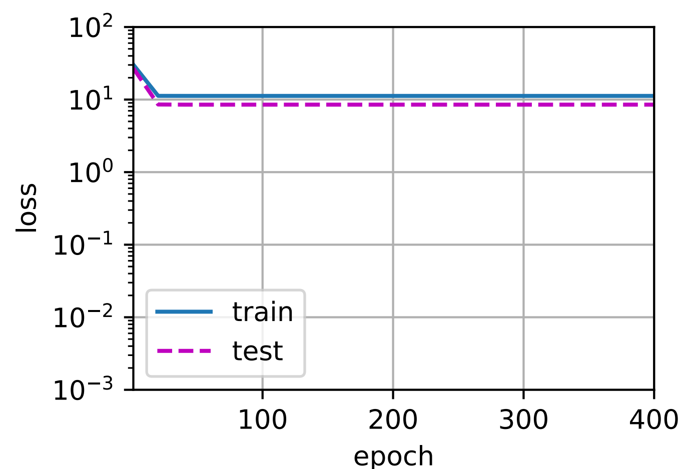


Ajuste de Função Linear (*Underfitting*)

Vamos dar uma outra olhada no ajuste de função linear. Após o declínio nas primeiras épocas, torna-se difícil diminuir ainda mais perda de treinamento deste modelo. Depois que a última iteração de época foi concluída, a perda de treinamento ainda é alta. Quando usado para ajustar padrões não lineares (como a função polinomial de terceira ordem aqui) os modelos lineares podem ser insuficientes e cometer *undefitting*.

```
# Pick the first two dimensions, i.e., 1, x, from the polynomial features
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

```
weight: [[3.3221662 3.7898972]]
```

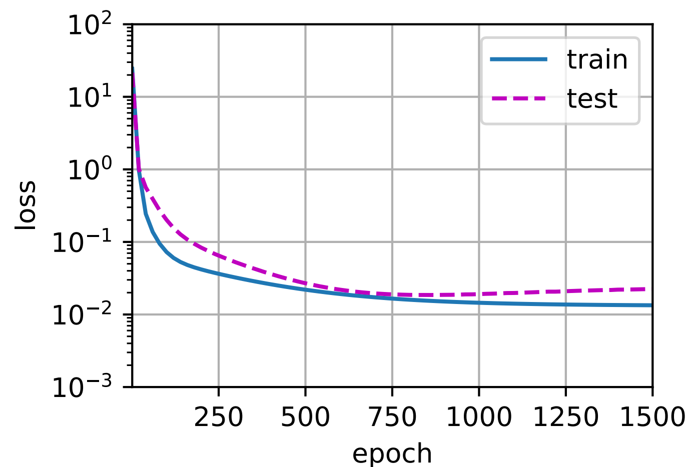


Ajuste de Função Polinomial de Ordem Superior (Overfitting)

Agora vamos tentar treinar o modelo usando um polinômio de grau muito alto. Aqui, não há dados suficientes para aprender que os coeficientes de grau superior devem ter valores próximos a zero. Como resultado, nosso modelo excessivamente complexo é tão suscetível que está sendo influenciado por ruído nos dados de treinamento. Embora a perda de treinamento possa ser efetivamente reduzida, a perda de teste ainda é muito maior. Mostra que o modelo complexo comete *overfitting*.

```
# Pick all the dimensions from the polynomial features
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
weight: [[ 4.98955870e+00  1.27648234e+00 -3.34345722e+00  5.14908457e+00
  -3.49774025e-02  1.36843681e+00 -1.54382661e-02  2.57640749e-01
   1.57875896e-01 -1.11054614e-01 -4.52921316e-02 -1.55643880e-01
  -1.13468990e-01  1.32403582e-01  1.19915836e-01  2.06710547e-02
   1.18126033e-03 -1.23166457e-01 -2.95518357e-02 -7.61602893e-02]]
```



Nas seções subsequentes, continuaremos a discutir problemas de *overfitting* e métodos para lidar com eles, como *weight decay* e *dropout*.

4.4.5 Resumo

- Uma vez que o erro de generalização não pode ser estimado com base no erro de treinamento, simplesmente minimizar o erro de treinamento não significa necessariamente uma redução no erro de generalização. Os modelos de *machine learning* precisam ter cuidado para evitar *overfitting*, de modo a minimizar o erro de generalização.
- Um conjunto de validação pode ser usado para seleção de modelo, desde que não seja usado com muita liberalidade.
- *Underfitting* significa que um modelo não é capaz de reduzir o erro de treinamento. Quando o erro de treinamento é muito menor do que o erro de validação, há *overfitting*.
- Devemos escolher um modelo apropriadamente complexo e evitar o uso de amostras de treinamento insuficientes.

4.4.6 Exercícios

1. Você pode resolver o problema de regressão polinomial exatamente? Dica: use álgebra linear.
2. Considere a seleção de modelo para polinômios:
   1. Plote a perda de treinamento vs. complexidade do modelo (grau do polinômio). O que você observa? Que grau de polinômio você precisa para reduzir a perda de treinamento para 0?
   2. Trace a perda de teste neste caso.
   3. Gere o mesmo gráfico em função da quantidade de dados.
3. O que acontece se você descartar a normalização ($1/i!$) Das feições polinomiais x^i? Você pode consertar isso de outra maneira?
4. Você pode esperar ver erro de generalização zero?

Discussions[49]

4.5 *Weight Decay*

Agora que caracterizamos o problema de *overfitting*, podemos apresentar algumas técnicas padrão para regularizar modelos. Lembre-se de que sempre podemos mitigar o *overfitting* saindo e coletando mais dados de treinamento. Isso pode ser caro, demorado, ou totalmente fora de nosso controle, tornando-o impossível a curto prazo. Por enquanto, podemos assumir que já temos tantos dados de alta qualidade quanto nossos recursos permitem e focar em técnicas de regularização.

Lembre-se disso em nosso exemplo de regressão polinomial (Section 4.4) poderíamos limitar a capacidade do nosso modelo simplesmente ajustando o grau do polinômio ajustado. Na verdade, limitar o número de características é uma técnica popular para mitigar o *overfitting*. No entanto, simplesmente deixando de lado as características pode ser um instrumento muito rude para o trabalho. Ficando com o exemplo da regressão polinomial, considere o que pode acontecer com entradas de alta dimensão. As extensões naturais de polinômios a dados multivariados são chamados de *monômios*, que são simplesmente produtos de potências de variáveis. O grau de um monômio é a soma das potências. Por exemplo, $x_1^2 x_2$, e $x_3 x_5^2$ são ambos monômios de grau 3.

Observe que o número de termos com grau d explode rapidamente à medida que d fica maior. Dadas as variáveis k, o número de monômios de grau d (ou seja, k escolheu d) é $\binom{k-1+d}{k-1}$. Mesmo pequenas mudanças no grau, digamos de 2 a 3, aumentam drasticamente a complexidade do nosso modelo. Portanto, muitas vezes precisamos de uma ferramenta mais refinada para ajustar a complexidade da função.

[49] https://discuss.d2l.ai/t/97

4.5.1 Normas e *Weight Decay*

Nós descrevemos a norma L_2 e a norma L_1, que são casos especiais da norma L_p mais geral em :numref: `subsec_lin-algebra-norms`. *Weight Decay* (comumente chamada de regularização L_2), pode ser a técnica mais amplamente usada para regularizar modelos paramétricos de aprendizado de máquina. A técnica é motivada pela intuição básica que entre todas as funções f, a função $f = 0$ (atribuindo o valor 0 a todas as entradas) é em certo sentido a *mais simples*, e que podemos medir a complexidade de uma função por sua distância de zero. Mas com que precisão devemos medir a distância entre uma função e zero? Não existe uma única resposta certa. Na verdade, ramos inteiros da matemática, incluindo partes da análise funcional e a teoria dos espaços de Banach, se dedicam a responder a esta questão.

Uma interpretação simples pode ser para medir a complexidade de uma função linear $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ por alguma norma de seu vetor de peso, por exemplo, $\|\mathbf{w}\|^2$. O método mais comum para garantir um vetor de peso pequeno é adicionar sua norma como um termo de penalidade para o problema de minimizar a perda. Assim, substituímos nosso objetivo original, *minimizando a perda de previsão nos rótulos de treinamento*, com novo objetivo, *minimizando a soma da perda de previsão e o prazo de penalização*. Agora, se nosso vetor de peso ficar muito grande, nosso algoritmo de aprendizagem pode se concentrar sobre como minimizar a norma de peso $\|\mathbf{w}\|^2$ vs. minimizar o erro de treinamento. Isso é exatamente o que queremos. Para ilustrar coisas no código, vamos reviver nosso exemplo anterior de Section 3.1 para regressão linear. Lá, nossa perda foi dada por

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{4.5.1}$$

Lembre-se de que $\mathbf{x}^{(i)}$ são as características, $y^{(i)}$ são rótulos para todos os exemplos de dados i e $(\mathbf{w}, b)$ são os parâmetros de peso e polarização, respectivamente. Para penalizar o tamanho do vetor de peso, devemos de alguma forma adicionar $\|\mathbf{w}\|^2$ para a função de perda, mas como o modelo deve negociar a perda padrão para esta nova penalidade aditiva? Na prática, caracterizamos essa compensação por meio da *constante de regularização* λ, um hiperparâmetro não negativo que ajustamos usando dados de validação:

$$L(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2, \tag{4.5.2}$$

Para $\lambda = 0$, recuperamos nossa função de perda original. Para $\lambda > 0$, restringimos o tamanho de $\|\mathbf{w}\|$. Dividimos por 2 por convenção: quando tomamos a derivada de uma função quadrática, o 2 e $1/2$ cancelam, garantindo que a expressão para a atualização pareça agradável e simples. O leitor astuto pode se perguntar por que trabalhamos com o quadrado norma e não a norma padrão (ou seja, a distância euclidiana). Fazemos isso por conveniência computacional. Ao elevar a norma L_2 ao quadrado, removemos a raiz quadrada, deixando a soma dos quadrados de cada componente do vetor de peso. Isso torna a derivada da penalidade fácil de calcular: a soma das derivadas é igual à derivada da soma.

Além disso, você pode perguntar por que trabalhamos com a norma L_2 em primeiro lugar e não, digamos, a norma L_1. Na verdade, outras escolhas são válidas e popular em todas as estatísticas. Enquanto modelos lineares L_2-regularizados constituem o algoritmo clássico de *regressão ridge*, regressão linear L_1-regularizada é um modelo igualmente fundamental em estatística, popularmente conhecido como *regressão lasso*.

Uma razão para trabalhar com a norma L_2 é que ela coloca uma penalidade descomunal em grandes componentes do vetor de peso. Isso influencia nosso algoritmo de aprendizagem para modelos que distribuem o peso uniformemente em um número maior de *features*. Na prática,

isso pode torná-los mais robustos ao erro de medição em uma única variável. Por outro lado, penalidades de L_1 levam a modelos que concentram pesos em um pequeno conjunto de recursos, zerando os outros pesos. Isso é chamado de *seleção de recursos*, o que pode ser desejável por outras razões.

Usando a mesma notação em (3.1.10), as atualizações de gradiente descendente estocástico de *minibatch* para regressão L_2-regularizada seguem:

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{4.5.3}$$

Como antes, atualizamos $\mathbf{w}$ com base no valor pelo qual nossa estimativa difere da observação. No entanto, também reduzimos o tamanho de $\mathbf{w}$ para zero. É por isso que o método às vezes é chamado de "queda de pesos" (*weigth decay*): dado o termo de pena sozinho, nosso algoritmo de otimização *decai* o peso em cada etapa do treinamento. Em contraste com a seleção de recursos, o *weight decay* nos oferece um mecanismo contínuo para ajustar a complexidade de uma função. Valores menores de λ correspondem para $\mathbf{w}$, menos restritos enquanto valores maiores de λ restringem $\mathbf{w}$ mais consideravelmente.

Se incluirmos uma penalidade de polarização correspondente b^2 pode variar entre as implementações, e pode variar entre as camadas de uma rede neural. Muitas vezes, não regularizamos o termo de polarização da camada de saída de uma rede.

4.5.2 Regressão Linear de Alta Dimensão

Podemos ilustrar os benefícios do *weight decay* por meio de um exemplo sintético simples.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

Primeiro, nós geramos alguns dados como antes

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{4.5.4}$$

Escolhemos nosso rótulo para ser uma função linear de nossas entradas, corrompidas por ruído gaussiano com média zero e desvio padrão 0,01. Para tornar os efeitos do *overfitting* pronunciados, podemos aumentar a dimensionalidade do nosso problema para $d = 200$ e trabalhar com um pequeno conjunto de treinamento contendo apenas 20 exemplos.

```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = torch.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

4.5.3 Implementação do Zero

A seguir, implementaremos o *weight decay* do zero, simplesmente adicionando a penalidade quadrada de L_2 para a função de destino original.

Inicializando os Parâmetros do Modelo

Primeiro, vamos definir uma função para inicializar aleatoriamente os parâmetros do nosso modelo.

```
def init_params():
    w = torch.normal(0, 1, size=(num_inputs, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return [w, b]
```

Definindo Penalidade de Norma L_2

Talvez a maneira mais conveniente de implementar essa penalidade é colocar todos os termos no lugar e resumi-los.

```
def l2_penalty(w):
    return torch.sum(w.pow(2)) / 2
```

Definindo o *Loop* de Treinamento

O código a seguir se ajusta a um modelo no conjunto de treinamento e avalia no conjunto de teste. A rede linear e a perda quadrada não mudaram desde Chapter 3, então iremos apenas importá-los via `d2l.linreg` e`d2l.squared_loss`. A única mudança aqui é que nossa perda agora inclui o prazo de penalidade.

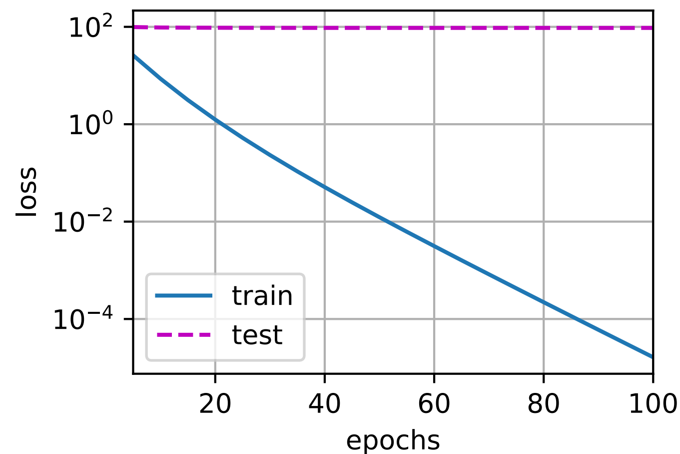
```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with torch.enable_grad():
                # The L2 norm penalty term has been added, and broadcasting
                # makes `l2_penalty(w)` a vector whose length is `batch_size`
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.sum().backward()
            d2l.sgd([w, b], lr, batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', torch.norm(w).item())
```

Treinamento sem Regularização

Agora executamos este código com `lambd = 0`, desabilitando o *weight decay*. Observe que fizemos muito *overfitting*, diminuindo o erro de treinamento, mas não o erro de teste — um caso clássico de *overfitting*.

```
train(lambd=0)
```

```
L2 norm of w: 12.984902381896973
```

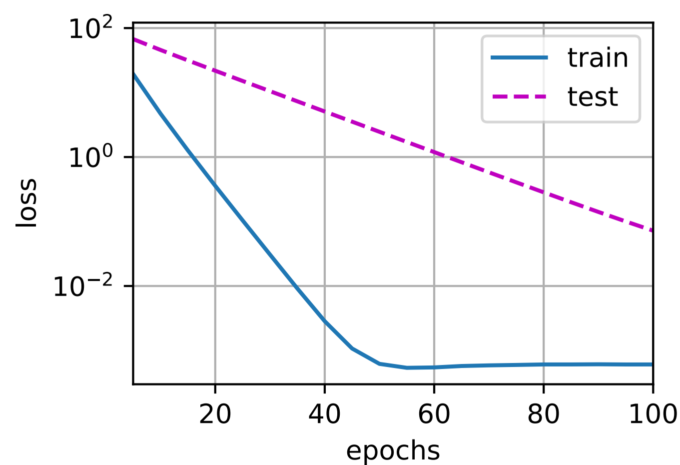


Usando *Weight Decay*

Abaixo, executamos com *weight decay* substancial. Observe que o erro de treinamento aumenta mas o erro de teste diminui. Este é precisamente o efeito esperamos da regularização.

```
train(lambd=3)
```

```
L2 norm of w: 0.37194105982780457
```



4.5.4 Implementação Concisa

Como o *weight decay* é onipresente na otimização da rede neural, a estrutura de *deep learning* torna-o especialmente conveniente, integrando o *weight decay* no próprio algoritmo de otimização para fácil uso em combinação com qualquer função de perda. Além disso, essa integração tem um benefício computacional, permitindo truques de implementação para adicionar *weight decay* ao algoritmo, sem qualquer sobrecarga computacional adicional. Uma vez que a parte de *weight decay* da atualização depende apenas do valor atual de cada parâmetro, o otimizador deve tocar em cada parâmetro uma vez de qualquer maneira.

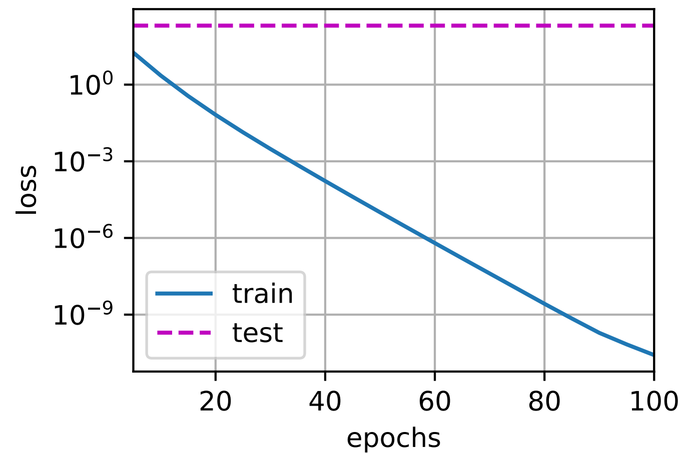
No código a seguir, especificamos o hiperparâmetro de *weight decay* diretamente por meio de `weight_decay` ao instanciar nosso otimizador. Por padrão, PyTorch decai ambos pesos e *bias* simultaneamente. Aqui nós apenas definimos `weight_decay` para o peso, para que o parâmetro de polarização b não diminua.

```
def train_concise(wd):
    net = nn.Sequential(nn.Linear(num_inputs, 1))
    for param in net.parameters():
        param.data.normal_()
    loss = nn.MSELoss()
    num_epochs, lr = 100, 0.003
    # The bias parameter has not decayed
    trainer = torch.optim.SGD([
        {"params":net[0].weight,'weight_decay': wd},
        {"params":net[0].bias}], lr=lr)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with torch.enable_grad():
                trainer.zero_grad()
                l = loss(net(X), y)
            l.backward()
            trainer.step()
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', net[0].weight.norm().item())
```

Os gráficos parecem idênticos aos de quando implementamos *weight decay* do zero. No entanto, eles funcionam consideravelmente mais rápido e são mais fáceis de implementar, um benefício que se tornará mais pronunciado para problemas maiores.

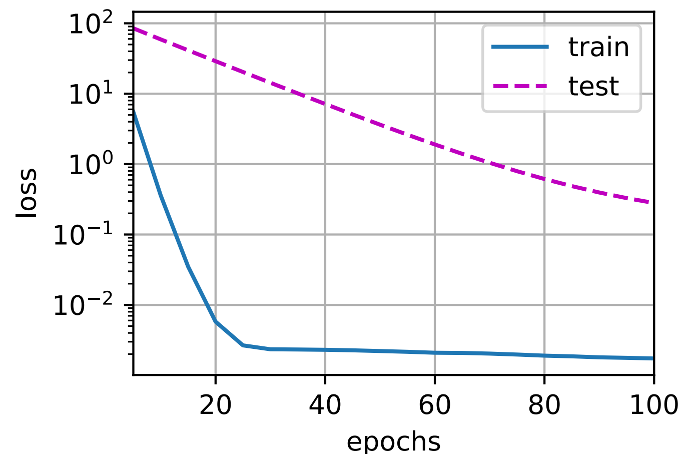
```
train_concise(0)
```

```
L2 norm of w: 13.588654518127441
```




```
train_concise(3)
```

```
L2 norm of w: 0.3481554687023163
```



Até agora, nós apenas tocamos em uma noção de o que constitui uma função linear simples. Além disso, o que constitui uma função não linear simples pode ser uma questão ainda mais complexa. Por exemplo, reproduzindo o espaço de Hilbert do kernel (RKHS)[50] permite aplicar ferramentas introduzidas para funções lineares em um contexto não linear. Infelizmente, algoritmos baseados em RKHS tendem a ser mal dimensionados para dados grandes e dimensionais. Neste livro, usaremos a heurística simples de aplicar o *weight decay* em todas as camadas de uma rede profunda.

[50] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

4.5.5 Resumo

- A regularização é um método comum para lidar com *overfitting*. Ela adiciona um termo de penalidade à função de perda no conjunto de treinamento para reduzir a complexidade do modelo aprendido.
- Uma escolha particular para manter o modelo simples é o *weight decay* usando uma penalidade de L_2. Isso leva à diminuição do peso nas etapas de atualização do algoritmo de aprendizagem.
- A funcionalidade de *weight decay* é fornecida em otimizadores de estruturas de *deep learning*.
- Diferentes conjuntos de parâmetros podem ter diferentes comportamentos de atualização no mesmo loop de treinamento.

4.5.6 Exercícios

1. Experimente o valor de λ no problema de estimação desta seção. Plote o treinamento e a precisão do teste como uma função de λ. O que você observa?
2. Use um conjunto de validação para encontrar o valor ideal de λ. É realmente o valor ideal? Isso importa?
3. Como seriam as equações de atualização se em vez de $\|\mathbf{w}\|^2$ usássemos $\sum_i |w_i|$ como nossa penalidade de escolha (regularização L_1)?
4. Sabemos que $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Você pode encontrar uma equação semelhante para matrizes (veja a norma Frobenius em :numref: `subsec_lin-algebra-norms`)?
5. Revise a relação entre o erro de treinamento e o erro de generalização. Além do *weight decay*, aumento do treinamento e o uso de um modelo de complexidade adequada, de que outras maneiras você pode pensar para lidar com o *overfitting*?
6. Na estatística bayesiana, usamos o produto anterior e a probabilidade de chegar a um posterior via $P(w \mid x) \propto P(x \mid w) P(w)$. Como você pode identificar $P(w)$ com regularização?

Discussions[51]

4.6 *Dropout*

Em Section 4.5, introduzimos a abordagem clássica para regularizar modelos estatísticos penalizando a norma L_2 dos pesos. Em termos probabilísticos, poderíamos justificar esta técnica argumentando que assumimos uma crença anterior que os pesos tomam valores de uma distribuição gaussiana com média zero. Mais intuitivamente, podemos argumentar que encorajamos o modelo a espalhar seus pesos entre muitas características, em vez de depender demais em um pequeno número de associações potencialmente espúrias.

[51] https://discuss.d2l.ai/t/99

4.6.1 *Overfitting* Revisitado

Diante de mais características do que exemplos, modelos lineares tendem a fazer *overfitting*. Mas dados mais exemplos do que características, geralmente podemos contar com modelos lineares para não ajustar demais. Infelizmente, a confiabilidade com a qual os modelos lineares generalizam têm um custo. Aplicados ingenuamente, os modelos lineares não levam em conta as interações entre as características. Para cada recurso, um modelo linear deve atribuir um peso positivo ou negativo, ignorando o contexto.

Em textos tradicionais, esta tensão fundamental entre generalização e flexibilidade é descrito como a *compensação de variação de polarização*. Modelos lineares têm alta polarização: eles podem representar apenas uma pequena classe de funções. No entanto, esses modelos têm baixa variação: eles fornecem resultados semelhantes em diferentes amostras aleatórias dos dados.

Redes neurais profundas habitam o oposto fim do espectro de polarização-variância. Ao contrário dos modelos lineares, as redes neurais não se limitam a examinar cada recurso individualmente. Eles podem aprender interações entre grupos de recursos. Por exemplo, eles podem inferir que "Nigéria" e "Western Union" aparecendo juntos em um e-mail indicam spam mas separadamente eles não o fazem.

Mesmo quando temos muito mais exemplos do que características, redes neurais profundas são capazes de fazer *overfitting*. Em 2017, um grupo de pesquisadores demonstrou a extrema flexibilidade das redes neurais treinando redes profundas em imagens rotuladas aleatoriamente. Apesar da ausência de qualquer padrão verdadeiro ligando as entradas às saídas, eles descobriram que a rede neural otimizada pelo gradiente descendente estocástico poderia rotular todas as imagens no conjunto de treinamento perfeitamente. Considere o que isso significa. Se os rótulos forem atribuídos uniformemente aleatoriamente e há 10 classes, então nenhum classificador pode fazer melhor precisão de 10% nos dados de validação. A lacuna de generalização aqui é de 90%. Se nossos modelos são tão expressivos que podem fazer tanto *overftitting*, então, quando deveríamos esperar que eles não se ajustem demais?

Os fundamentos matemáticos para as propriedades de generalização intrigantes de redes profundas permanecem questões de pesquisa em aberto, e encorajamos os leitores orientados teoricamente para se aprofundar no assunto. Por enquanto, nos voltamos para a investigação de ferramentas práticas que tendem a melhorar empiricamente a generalização de redes profundas.

4.6.2 Robustez por Meio de Perturbações

Vamos pensar brevemente sobre o que nós esperamos de um bom modelo preditivo. Queremos que ele funcione bem com dados não vistos. A teoria da generalização clássica sugere que para fechar a lacuna entre treinar e testar o desempenho, devemos ter como objetivo um modelo simples. A simplicidade pode vir na forma de um pequeno número de dimensões. Exploramos isso ao discutir as funções de base monomial de modelos lineares em Section 4.4. Além disso, como vimos ao discutir o *weight decay* (regularização L_2) em Section 4.5, a norma (inversa) dos parâmetros também representa uma medida útil de simplicidade. Outra noção útil de simplicidade é suavidade, ou seja, que a função não deve ser sensível a pequenas mudanças em suas entradas. Por exemplo, quando classificamos imagens, esperaríamos que adicionar algum ruído aleatório aos pixels seja inofensivo.

Em 1995, Christopher Bishop formalizou essa ideia quando ele provou que o treinamento com ruído de entrada equivale à regularização de Tikhonov (Bishop, 1995). Este trabalho traçou uma conexão matemática clara entre o requisito de que uma função seja suave (e, portanto, simples), e a exigência de que seja resiliente a perturbações na entrada.

Então, em 2014, Srivastava et al. (Srivastava et al., 2014) desenvolveram uma ideia inteligente de como aplicar a ideia de Bishop às camadas internas de uma rede também. Ou seja, eles propuseram injetar ruído em cada camada da rede antes de calcular a camada subsequente durante o treinamento. Eles perceberam que durante o treinamento uma rede profunda com muitas camadas, injetando ruído reforça suavidade apenas no mapeamento de entrada-saída.

A ideia deles, chamada *dropout*, envolve injetar ruído durante a computação de cada camada interna durante a propagação direta, e se tornou uma técnica padrão para treinar redes neurais. O método é chamado *dropout* porque nós literalmente *abandonamos [1]_* alguns neurônios durante o treinamento. Ao longo do treinamento, em cada iteração, *dropout* padrão consiste em zerar alguma fração dos nós em cada camada antes de calcular a camada subsequente.

Para ser claro, estamos impondo nossa própria narrativa com o link para Bishop. O artigo original em *dropout* oferece intuição através de uma surpreendente analogia com a reprodução sexual. Os autores argumentam que o *overfitting* da rede neural é caracterizado por um estado em que cada camada depende de um específico padrão de ativações na camada anterior, chamando essa condição de *co-adaptação*. A desistência, eles afirmam, acaba com a co-adaptação assim como a reprodução sexual é argumentada para quebrar genes co-adaptados.

O principal desafio é como injetar esse ruído. Uma ideia é injetar o ruído de uma maneira *imparcial* de modo que o valor esperado de cada camada — enquanto fixa os outros — seja igual ao valor que teria o ruído ausente.

No trabalho de Bishop, ele adicionou ruído gaussiano às entradas de um modelo linear. A cada iteração de treinamento, ele adicionava ruído amostrado a partir de uma distribuição com média zero $\epsilon \sim \mathcal{N}(0,\sigma^2)$ à entrada $\mathbf{x}$, produzindo um ponto perturbado $\mathbf{x}' = \mathbf{x} + \epsilon$. Na expectativa, $E[\mathbf{x}'] = \mathbf{x}$.

Na regularização de *dropout* padrão, um tira o *bias* de cada camada normalizando pela fração de nós que foram retidos (não descartados). Em outras palavras, com *probabilidade de dropout* p, cada ativação intermediária h é substituída por uma variável aleatória h' como segue:

$$h' = \begin{cases} 0 & \text{com probabilidade } p \\ \frac{h}{1-p} & \text{caso contrário} \end{cases} \tag{4.6.1}$$

Por design, a esperança permanece inalterada, ou seja, $E[h'] = h$.

4.6.3 *Dropout* na Prática

Lembre-se do MLP com uma camada oculta e 5 unidades ocultas em `fig_mlp`. Quando aplicamos o *dropout* a uma camada oculta, zerando cada unidade oculta com probabilidade p, o resultado pode ser visto como uma rede contendo apenas um subconjunto dos neurônios originais. Em Fig. 4.6.1, h_2 e h_5 são removidos. Consequentemente, o cálculo das saídas não depende mais de h_2 ou h_5 e seus respectivos gradientes também desaparecem ao executar retropropagação. Desta forma, o cálculo da camada de saída não pode ser excessivamente dependente de qualquer um elemento de $h_1, \ldots, h_5$.

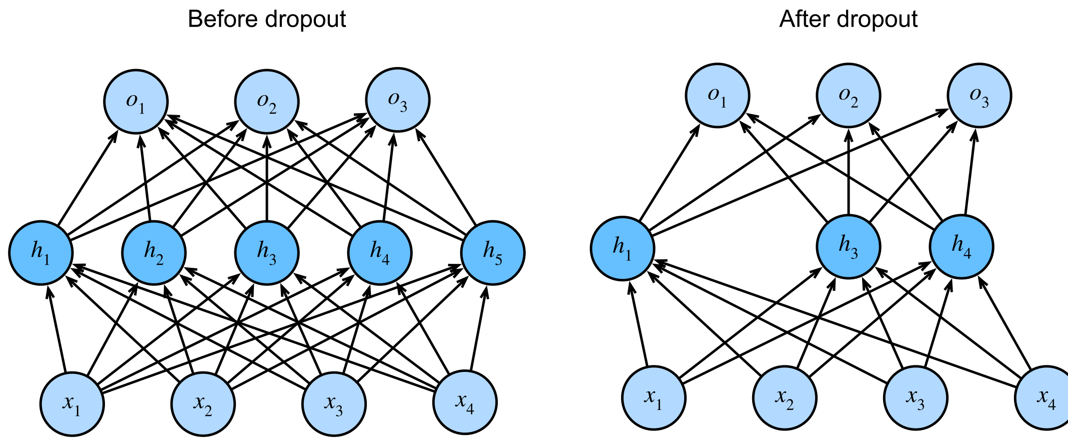


Fig. 4.6.1: MLP antes e depois do *dropout*.

Normalmente, desabilitamos o *dropout* no momento do teste. Dado um modelo treinado e um novo exemplo, nós não eliminamos nenhum nó e, portanto, não precisamos normalizar. No entanto, existem algumas exceções: alguns pesquisadores usam o *dropout* na hora do teste como uma heurística para estimar a *incerteza* das previsões da rede neural: se as previsões concordam em muitas máscaras de *dropout*, então podemos dizer que a rede está mais confiável.

4.6.4 Implementação do Zero

Para implementar a função *dropout* para uma única camada, devemos tirar tantas amostras de uma variável aleatória de Bernoulli (binária) quanto o número de dimensões de nossa camada, onde a variável aleatória assume o valor 1 (*keep*) com probabilidade $1 - p$ e 0 (*drop*) com probabilidade p. Uma maneira fácil de implementar isso é primeiro desenhar amostras da distribuição uniforme $U[0, 1]$. Então, podemos manter os nós para os quais a amostra correspondente é maior do que p, descartando o resto.

No código a seguir, implementamos uma função `dropout_layer` que elimina os elementos na entrada do tensor `X` com probabilidade de `dropout`, redimensionando o restante conforme descrito acima: dividindo os sobreviventes por `1.0-dropout`.

```
import torch
from torch import nn
from d2l import torch as d2l


def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return torch.zeros_like(X)
    # In this case, all elements are kept
    if dropout == 0:
        return X
    mask = (torch.Tensor(X.shape).uniform_(0, 1) > dropout).float()
    return mask * X / (1.0 - dropout)
```

Podemos testar a função `dropout_layer` em alguns exemplos. Nas seguintes linhas de código,

passamos nossa entrada X através da operação de *dropout*, com probabilidades 0, 0,5 e 1, respectivamente.

```
X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
print(X)
print(dropout_layer(X, 0.))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1.))
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  0.,  4.,  6.,  8.,  0.,  0.,  0.],
        [ 0.,  0., 20.,  0.,  0., 26., 28., 30.]])
tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Definindo os Parâmetros do Modelo

Mais uma vez, trabalhamos com o conjunto de dados Fashion-MNIST introduzido em Section 3.5. Nós definimos um MLP com duas camadas ocultas contendo 256 unidades cada.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
```

Definindo o Modelo

O modelo abaixo aplica *dropout* à saída de cada camada oculta (seguindo a função de ativação). Podemos definir probabilidades de *dropout* para cada camada separadamente. Uma tendência comum é definir uma probabilidade de *dropout* mais baixa perto da camada de entrada. Abaixo, nós o definimos como 0,2 e 0,5 para o primeiro e segundas camadas ocultas, respectivamente. Garantimos que o *dropout* seja ativo apenas durante o treinamento.

```
dropout1, dropout2 = 0.2, 0.5

class Net(nn.Module):
    def __init__(self, num_inputs, num_outputs, num_hiddens1, num_hiddens2,
                 is_training = True):
        super(Net, self).__init__()
        self.num_inputs = num_inputs
        self.training = is_training
        self.lin1 = nn.Linear(num_inputs, num_hiddens1)
        self.lin2 = nn.Linear(num_hiddens1, num_hiddens2)
        self.lin3 = nn.Linear(num_hiddens2, num_outputs)
        self.relu = nn.ReLU()

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape((-1, self.num_inputs))))
        # Use dropout only when training the model
        if self.training == True:
```

(continues on next page)

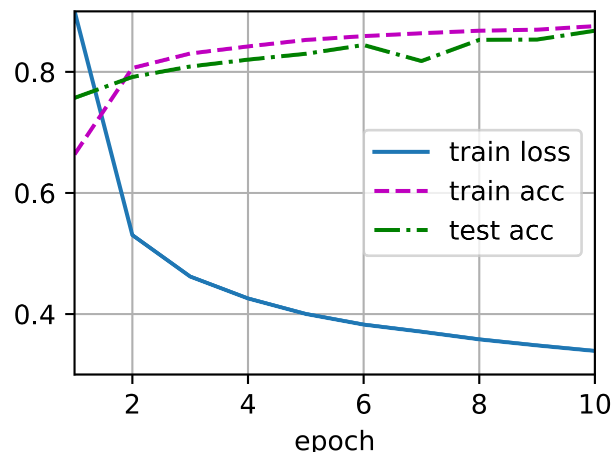
(continued from previous page)

```
            # Add a dropout layer after the first fully connected layer
            H1 = dropout_layer(H1, dropout1)
        H2 = self.relu(self.lin2(H1))
        if self.training == True:
            # Add a dropout layer after the second fully connected layer
            H2 = dropout_layer(H2, dropout2)
        out = self.lin3(H2)
        return out


net = Net(num_inputs, num_outputs, num_hiddens1, num_hiddens2)
```

Treinamento e Teste

Isso é semelhante ao treinamento e teste de MLPs descritos anteriormente.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = nn.CrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



4.6.5 Implementação Concisa

Com APIs de alto nível, tudo o que precisamos fazer é adicionar uma camada `Dropout` após cada camada totalmente conectada, passando na probabilidade de *dropout* como o único argumento para seu construtor. Durante o treinamento, a camada `Dropout` irá aleatoriamente eliminar as saídas da camada anterior (ou de forma equivalente, as entradas para a camada subsequente) de acordo com a probabilidade de abandono especificada. Quando não estiver no modo de treinamento, a camada `Dropout` simplesmente passa os dados durante o teste.

```
net = nn.Sequential(nn.Flatten(),
        nn.Linear(784, 256),
        nn.ReLU(),
```

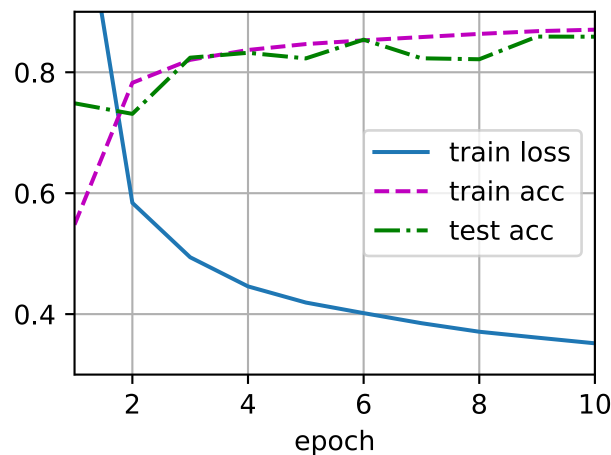
(continues on next page)

(continued from previous page)

```
        # Add a dropout layer after the first fully connected layer
        nn.Dropout(dropout1),
        nn.Linear(256, 256),
        nn.ReLU(),
        # Add a dropout layer after the second fully connected layer
        nn.Dropout(dropout2),
        nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

Em seguida, treinamos e testamos o modelo.

```
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



4.6.6 Resumo

- Além de controlar o número de dimensões e o tamanho do vetor de peso, o *dropout* é outra ferramenta para evitar *overfitting*. Frequentemente, eles são usados em conjunto.
- *Dropout* substitui uma ativação h por uma variável aleatória com valor esperado h.
- O *dropout* é usado apenas durante o treinamento.

4.6.7 Exercícios

1. O que acontece se você alterar as probabilidades de *dropout* para a primeira e segunda camadas? Em particular, o que acontece se você trocar os de ambas as camadas? Projete um experimento para responder a essas perguntas, descreva seus resultados quantitativamente e resuma as conclusões qualitativas.
2. Aumente o número de épocas e compare os resultados obtidos ao usar *dropout* com os que não o usam.
3. Qual é a variação das ativações em cada camada oculta quando o *dropout* é e não é aplicado? Desenhe um gráfico para mostrar como essa quantidade evolui ao longo do tempo para ambos os modelos.
4. Por que o *dropout* normalmente não é usado no momento do teste?
5. Usando o modelo nesta seção como exemplo, compare os efeitos do uso de *dropout* e *weight decay*. O que acontece quando o *dropout* e *weight decay* são usados ao mesmo tempo? Os resultados são cumulativos? Existem retornos diminuídos (ou pior)? Eles se cancelam?
6. O que acontece se aplicarmos o *dropout* aos pesos individuais da matriz de pesos em vez das ativações?
7. Invente outra técnica para injetar ruído aleatório em cada camada que seja diferente da técnica de *dropout* padrão. Você pode desenvolver um método que supere o *dropout* no conjunto de dados Fashion-MNIST (para uma arquitetura fixa)?

Discussions[52]

4.7 Propagação Direta, Propagação Reversa e Gráficos Computacionais

Até agora, treinamos nossos modelos com gradiente descendente estocástico de *minibatch*. No entanto, quando implementamos o algoritmo, nós apenas nos preocupamos com os cálculos envolvidos em *propagação direta* através do modelo. Quando chegou a hora de calcular os gradientes, acabamos de invocar a função *backpropagation* (propagação reversa) fornecida pela estrutura de *deep learning*.

O cálculo automático de gradientes (diferenciação automática) simplifica profundamente a implementação de algoritmos de *deep learning*. Antes da diferenciação automática, mesmo pequenas mudanças em modelos complicados exigiam recalcular derivadas complicadas manualmente. Surpreendentemente, muitas vezes, os trabalhos acadêmicos tiveram que alocar várias páginas para derivar regras de atualização. Embora devamos continuar a confiar na diferenciação automática para que possamos nos concentrar nas partes interessantes, você deve saber como esses gradientes são calculados sob o capô se você quiser ir além de uma rasa compreensão da aprendizagem profunda.

Nesta seção, fazemos um mergulho profundo nos detalhes de *propagação para trás* (mais comumente chamado de *backpropagation*). Para transmitir alguns *insights* para ambas as técnicas e suas implementações, contamos com alguma matemática básica e gráficos computacionais. Para começar, focamos nossa exposição em um MLP de uma camada oculta com *weight decay* (regularização:math:*L_2*).

[52] https://discuss.d2l.ai/t/101

4.7.1 Propagação Direta

Propagação direta (ou *passagem direta*) refere-se ao cálculo e armazenamento de variáveis intermediárias (incluindo saídas) para uma rede neural em ordem da camada de entrada para a camada de saída. Agora trabalhamos passo a passo com a mecânica de uma rede neural com uma camada oculta. Isso pode parecer tedioso, mas nas palavras eternas do virtuoso do funk James Brown, você deve "pagar o custo para ser o chefe".

Por uma questão de simplicidade, vamos assumir que o exemplo de entrada é $\mathbf{x} \in \mathbb{R}^d$ e que nossa camada oculta não inclui um termo de *bias*. Aqui, a variável intermediária é:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{4.7.1}$$

onde $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ é o parâmetro de peso da camada oculta. Depois de executar a variável intermediária $\mathbf{z} \in \mathbb{R}^h$ através da função de ativação ϕ obtemos nosso vetor de ativação oculto de comprimento h,

$$\mathbf{h} = \phi(\mathbf{z}). \tag{4.7.2}$$

A variável oculta $\mathbf{h}$ também é uma variável intermediária. Supondo que os parâmetros da camada de saída só possuem um peso de $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, podemos obter uma variável de camada de saída com um vetor de comprimento q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{4.7.3}$$

Supondo que a função de perda seja l e o *label* de exemplo é y, podemos então calcular o prazo de perda para um único exemplo de dados,

$$L = l(\mathbf{o}, y). \tag{4.7.4}$$

De acordo com a definição de regularização de L_2 dado o hiperparâmetro λ, o prazo de regularização é

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right), \tag{4.7.5}$$

onde a norma Frobenius da matriz é simplesmente a norma L_2 aplicada depois de achatar a matriz em um vetor. Por fim, a perda regularizada do modelo em um dado exemplo de dados é:

$$J = L + s. \tag{4.7.6}$$

Referimo-nos a J como a *função objetivo* na discussão a seguir.

4.7.2 Gráfico Computatcional de Propagação Direta

Traçar *gráficos computacionais* nos ajuda a visualizar as dependências dos operadores e variáveis dentro do cálculo. Fig. 4.7.1 contém o gráfico associado com a rede simples descrita acima, onde quadrados denotam variáveis e círculos denotam operadores. O canto inferior esquerdo significa a entrada e o canto superior direito é a saída. Observe que as direções das setas (que ilustram o fluxo de dados) são principalmente para a direita e para cima.

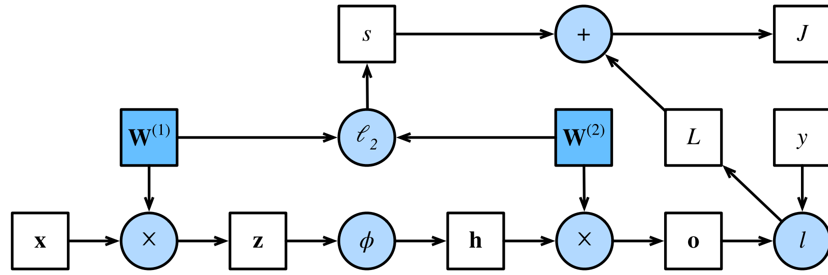


Fig. 4.7.1: Gráfico computacional de propagação direta.

4.7.3 Propagação Reversa

Propagação reversa (*retropropagação*) refere-se ao método de cálculo do gradiente dos parâmetros da rede neural. Em suma, o método atravessa a rede na ordem inversa, da saída para a camada de entrada, de acordo com a *regra da cadeia* do cálculo. O algoritmo armazena quaisquer variáveis intermediárias (derivadas parciais) necessárias ao calcular o gradiente com relação a alguns parâmetros. Suponha que temos funções $\mathsf{Y} = f(\mathsf{X})$ e $\mathsf{Z} = g(\mathsf{Y})$, em que a entrada e a saída $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ são tensores de formas arbitrárias. Usando a regra da cadeia, podemos calcular a derivada de Z with respect to X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{4.7.7}$$

Aqui usamos o operador prod para multiplicar seus argumentos depois que as operações necessárias, como transposição e troca de posições de entrada, foram realizadas. Para vetores, isso é simples: é simplesmente multiplicação matriz-matriz. Para tensores dimensionais superiores, usamos a contraparte apropriada. O operador prod esconde todo o *overhead* de notação.

Lembre-se disso os parâmetros da rede simples com uma camada oculta, cujo gráfico computacional está em Fig. 4.7.1, são $\mathbf{W}^{(1)}$ e $\mathbf{W}^{(2)}$. O objetivo da retropropagação é calcular os gradientes $\partial J/\partial \mathbf{W}^{(1)}$ e $\partial J/\partial \mathbf{W}^{(2)}$. Para conseguir isso, aplicamos a regra da cadeia e calcular, por sua vez, o gradiente de cada variável e parâmetro intermediário. A ordem dos cálculos é invertida em relação àquelas realizadas na propagação direta, uma vez que precisamos começar com o resultado do gráfico computacional e trabalhar nosso caminho em direção aos parâmetros. O primeiro passo é calcular os gradientes da função objetivo $J = L + s$ com relação ao prazo de perda L e o prazo de regularização s.

$$\frac{\partial J}{\partial L} = 1 \text{ e } \frac{\partial J}{\partial s} = 1. \tag{4.7.8}$$

Em seguida, calculamos o gradiente da função objetivo em relação à variável da camada de saída $\mathbf{o}$ de acordo com a regra da cadeia:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{4.7.9}$$

Em seguida, calculamos os gradientes do termo de regularização com respeito a ambos os parâmetros:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{4.7.10}$$

Agora podemos calcular o gradiente $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ dos parâmetros do modelo mais próximos da camada de saída. Usar a regra da cadeia produz:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{4.7.11}$$

Para obter o gradiente em relação a $\mathbf{W}^{(1)}$ precisamos continuar a retropropagação ao longo da camada de saída para a camada oculta. O gradiente em relação às saídas da camada oculta $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ é dado por

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{4.7.12}$$

Uma vez que a função de ativação ϕ se aplica aos elementos, calculando o gradiente $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ da variável intermediária $\mathbf{z}$ requer que usemos o operador de multiplicação elemento a elemento, que denotamos por $\odot$:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{4.7.13}$$

Finalmente, podemos obter o gradiente $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ dos parâmetros do modelo mais próximos da camada de entrada. De acordo com a regra da cadeia, obtemos

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{4.7.14}$$

4.7.4 Treinando Redes Neurais

Ao treinar redes neurais, a propagação direta e reversa dependem uma da outra. Em particular, para propagação direta, atravessamos o gráfico computacional na direção das dependências e calculamos todas as variáveis em seu caminho. Eles são então usadas para retropropagação onde a ordem de computação no gráfico é invertida.

Tome a rede simples mencionada acima como um exemplo para ilustrar. Por um lado, calcular o termo de regularização (4.7.5) durante a propagação para a frente depende dos valores atuais dos parâmetros do modelo $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. Eles são fornecidos pelo algoritmo de otimização de acordo com a retropropagação na iteração mais recente. Por outro lado, o cálculo do gradiente para o parâmetro (4.7.11) durante a retropropagação depende do valor atual da variável oculta $\mathbf{h}$, que é fornecido por propagação direta.

Portanto, ao treinar redes neurais, após os parâmetros do modelo serem inicializados, alternamos a propagação direta com a retropropagação, atualizando os parâmetros do modelo usando gradientes fornecidos por retropropagação. Observe que a retropropagação reutiliza os valores intermediários armazenados da propagação direta para evitar cálculos duplicados. Uma das consequências é que precisamos reter os valores intermediários até que a retropropagação seja concluída. Esta é também uma das razões pelas quais o treinamento requer muito mais memória do que a previsão simples. Além disso, o tamanho de tais valores intermediários é aproximadamente proporcional ao número de camadas de rede e ao tamanho do lote. Por isso, treinar redes mais profundas usando tamanhos de lote maiores mais facilmente leva a erros de *falta de memória*.

4.7.5 Resumo

- A propagação direta calcula e armazena sequencialmente variáveis intermediárias no gráfico computacional definido pela rede neural. Ela prossegue da camada de entrada para a camada de saída.
- A retropropagação calcula e armazena sequencialmente os gradientes de variáveis e parâmetros intermediários na rede neural na ordem inversa.
- Ao treinar modelos de *deep learning*, a propagação direta e reversa são interdependentes.
- O treinamento requer muito mais memória do que a previsão.

4.7.6 Exercícios

1. Suponha que as entradas $\mathbf{X}$ para alguma função escalar f sejam matrizes $n \times m$. Qual é a dimensionalidade do gradiente de f em relação a $\mathbf{X}$?
2. Adicione um *bias* à camada oculta do modelo descrito nesta seção (você não precisa incluir um *bias* no termo de regularização).
    1. Desenhe o gráfico computacional correspondente.
    2. Derive as equações de propagação direta e reversa.
3. Calcule a pegada de memória para treinamento e predição no modelo descrito nesta seção.
4. Suponha que você deseja calcular derivadas secundárias. O que acontece com o gráfico computacional? Quanto tempo você espera que o cálculo demore?
5. Suponha que o gráfico computacional seja muito grande para sua GPU.
    1. Você pode particioná-lo em mais de uma GPU?
    2. Quais são as vantagens e desvantagens em relação ao treinamento em um *minibatch* menor?

Discussions[53]

4.8 Estabilidade Numérica e Inicialização

Até agora, todos os modelos que implementamos exigiram que inicializássemos seus parâmetros de acordo com alguma distribuição pré-especificada. Até agora, considerávamos o esquema de inicialização garantido, encobrindo os detalhes de como essas escolhas são feitas. Você pode até ter ficado com a impressão de que essas escolhas não são especialmente importantes. Pelo contrário, a escolha do esquema de inicialização desempenha um papel significativo na aprendizagem da rede neural, e pode ser crucial para manter a estabilidade numérica. Além disso, essas escolhas podem ser amarradas de maneiras interessantes com a escolha da função de ativação não linear. Qual função escolhemos e como inicializamos os parâmetros pode determinar a rapidez com que nosso algoritmo de otimização converge. Escolhas ruins aqui podem nos fazer encontrar explosões ou desaparecimento de gradientes durante o treinamento. Nesta seção, nos aprofundamos nesses tópicos com mais detalhes e discutimos algumas heurísticas úteis que você achará útil ao longo de sua carreira em *deep learning*.

[53] https://discuss.d2l.ai/t/102

4.8.1 Explosão e Desaparecimento de Gradientes

Considere uma rede profunda com \$ L \$ camadas, entrada \$ mathbf {x} \$ e saída $\mathbf{o}$. Com cada camada l definida por uma transformação f_l parametrizada por pesos $\mathbf{W}^{(l)}$, cuja variável oculta é $\mathbf{h}^{(l)}$ (com $\mathbf{h}^{(0)} = \mathbf{x}$), nossa rede pode ser expressa como:

$$\mathbf{h}^{(l)} = f_l (\mathbf{h}^{(l-1)}) \text{ e portanto } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}). \tag{4.8.1}$$

Se todas as variáveis ocultas e a entrada forem vetores, podemos escrever o gradiente de $\mathbf{o}$ em relação a qualquer conjunto de parâmetros $\mathbf{W}^{(l)}$ da seguinte forma:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{ \mathbf{M}^{(L)} \stackrel{\mathrm{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{ \mathbf{M}^{(l+1)} \stackrel{\mathrm{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{ \mathbf{v}^{(l)} \stackrel{\mathrm{def}}{=}}. \tag{4.8.2}$$

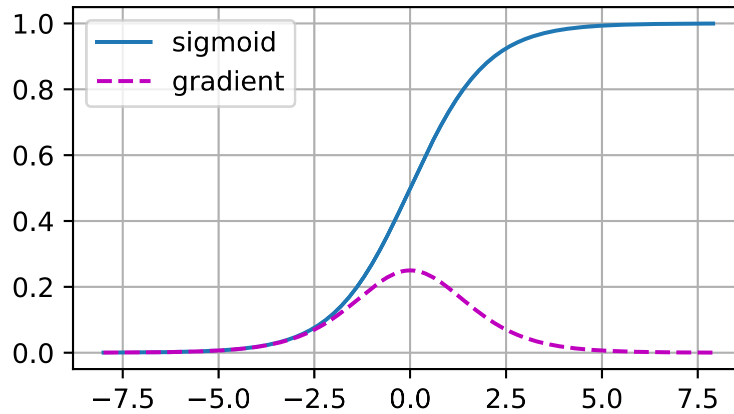
Em outras palavras, este gradiente é o produto das matrizes $L-l$ $\mathbf{M}^{(L)} \cdot \ldots \cdot \mathbf{M}^{(l+1)}$ e o vetor gradiente $\mathbf{v}^{(l)}$. Assim, somos suscetíveis aos mesmos problemas de *underflow* numérico que muitas vezes surgem ao multiplicar muitas probabilidades. Ao lidar com probabilidades, um truque comum é mudar para o espaço de registro, ou seja, mudar pressão da mantissa para o expoente da representação numérica. Infelizmente, nosso problema acima é mais sério: inicialmente as matrizes $\mathbf{M}^{(l)}$ podem ter uma grande variedade de autovalores. Eles podem ser pequenos ou grandes e seu produto pode ser *muito grande* ou *muito pequeno*.

Os riscos apresentados por gradientes instáveis vão além da representação numérica. Gradientes de magnitude imprevisível também ameaçam a estabilidade de nossos algoritmos de otimização. Podemos estar enfrentando atualizações de parâmetros que são (i) excessivamente grandes, destruindo nosso modelo (o problema da *explosão do gradiente*); ou (ii) excessivamente pequeno (o problema do *desaparecimento do gradiente*), tornando a aprendizagem impossível como parâmetros dificilmente se move a cada atualização.

Desaparecimento do Gradiente

Um culpado frequente que causa o problema do desaparecimento de gradiente é a escolha da função de ativação σ que é anexada após as operações lineares de cada camada. Historicamente, a função sigmóide $1/(1+\exp(-x))$ (introduzida em Section 4.1) era popular porque se assemelha a uma função de limiar. Como as primeiras redes neurais artificiais foram inspiradas por redes neurais biológicas, a ideia de neurônios que disparam *totalmente* ou *nem um pouco* (como neurônios biológicos) parecia atraente. Vamos dar uma olhada mais de perto no sigmóide para ver por que isso pode causar desaparecimento de gradientes.

```
%matplotlib inline
import torch
from d2l import torch as d2l

x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))

d2l.plot(x.detach().numpy(), [y.detach().numpy(), x.grad.numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



Como você pode ver, o gradiente do sigmóide desaparece tanto quando suas entradas são grandes quanto quando são pequenas. Além disso, ao retropropagar através de muitas camadas, a menos que estejamos na zona Cachinhos Dourados, onde as entradas para muitos dos sigmóides são próximas de zero, os gradientes do produto geral podem desaparecer. Quando nossa rede possui muitas camadas, a menos que tenhamos cuidado, o gradiente provavelmente será cortado em alguma camada. Na verdade, esse problema costumava atormentar o treinamento profundo da rede. Consequentemente, ReLUs, que são mais estáveis (mas menos neuralmente plausíveis), surgiram como a escolha padrão para os profissionais.

Explosão de Gradiente

O problema oposto, quando os gradientes explodem, pode ser igualmente irritante. Para ilustrar isso um pouco melhor, desenhamos 100 matrizes aleatórias Gaussianas e multiplicamos-nas com alguma matriz inicial. Para a escala que escolhemos (a escolha da variação $\sigma^2 = 1$), o produto da matriz explode. Quando isso acontece devido à inicialização de uma rede profunda, não temos chance de obter um otimizador de gradiente descendente capaz de convergir.

```python
M = torch.normal(0, 1, size=(4,4))
print('a single matrix \n',M)
for i in range(100):
    M = torch.mm(M,torch.normal(0, 1, size=(4, 4)))

print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[ 1.4391,  0.5853,  0.3140, -0.8884],
        [ 1.4385,  1.3793, -1.2473,  0.6873],
        [ 0.1406, -0.1450, -0.5697, -0.2563],
        [-0.8765, -0.1916,  0.1412,  0.3991]])
after multiplying 100 matrices
 tensor([[-1.5826e+23,  6.3882e+22, -2.6931e+23, -5.2050e+23],
        [-3.4426e+23,  1.3896e+23, -5.8582e+23, -1.1322e+24],
        [-2.7453e+22,  1.1081e+22, -4.6717e+22, -9.0290e+22],
        [ 1.1260e+23, -4.5449e+22,  1.9160e+23,  3.7031e+23]])
```

Quebrando a Simetria

Outro problema no projeto de rede neural é a simetria inerente à sua parametrização. Suponha que temos um MLP simples com uma camada oculta e duas unidades. Neste caso, poderíamos permutar os pesos $\mathbf{W}^{(1)}$ da primeira camada e da mesma forma permutar os pesos da camada de saída para obter a mesma função. Não há nada especial em diferenciar a primeira unidade oculta vs. a segunda unidade oculta. Em outras palavras, temos simetria de permutação entre as unidades ocultas de cada camada.

Isso é mais do que apenas um incômodo teórico. Considere o já mencionado MLP de uma camada oculta com duas unidades ocultas. Para ilustração, suponha que a camada de saída transforme as duas unidades ocultas em apenas uma unidade de saída. Imagine o que aconteceria se inicializássemos todos os parâmetros da camada oculta como $\mathbf{W}^{(1)} = c$ para alguma constante c. Neste caso, durante a propagação direta qualquer unidade oculta leva as mesmas entradas e parâmetros a produzir a mesma ativação, que é alimentada para a unidade de saída. Durante a retropropagação, diferenciar a unidade de saída com respeito aos parâmetros $\mathbf{W}^{(1)}$ dá um gradiente cujos elementos tomam o mesmo valor. Assim, após a iteração baseada em gradiente (por exemplo, gradiente descendente estocástico de *minibatch*), todos os elementos de $\mathbf{W}^{(1)}$ ainda têm o mesmo valor. Essas iterações nunca iriam *quebrar a simetria* por conta própria e podemos nunca ser capazes de perceber o poder expressivo da rede. A camada oculta se comportaria como se tivesse apenas uma unidade. Observe que, embora o gradiente descendente estocástico de *minibatch* não quebrasse essa simetria, a regularização do *dropout* iria!

4.8.2 Inicialização de Parâmetros

Uma forma de abordar — ou pelo menos mitigar — os problemas levantados acima é através de inicialização cuidadosa. Cuidado adicional durante a otimização e a regularização adequada pode aumentar ainda mais a estabilidade.

Inicialização Padrão

Nas seções anteriores, por exemplo, em Section 3.3, nós usamos uma distribuição normal para inicializar os valores de nossos pesos. Se não especificarmos o método de inicialização, o *framework* irá usar um método de inicialização aleatória padrão, que geralmente funciona bem na prática para tamanhos moderados de problemas.

Inicialização de Xavier

Vejamos a distribuição da escala de uma saída (por exemplo, uma variável oculta) o_i para alguma camada totalmente conectada *sem não linearidades*. Com n_in, entradas x_j e seus pesos associados w_{ij} para esta camada, uma saída é dada por

$$o_i = \sum_{j=1}^{n_\mathrm{in}} w_{ij} x_j. \tag{4.8.3}$$

Os pesos w_{ij} estão todos sorteados independentemente da mesma distribuição. Além disso, vamos supor que esta distribuição tem média zero e variância σ^2. Observe que isso não significa que a distribuição deve ser gaussiana, apenas que a média e a variância precisam existir. Por enquanto, vamos supor que as entradas para a camada x_j também têm média zero e variância γ^2 e que elas

são independentes de w_{ij} e independentes uma da outra. Nesse caso, podemos calcular a média e a variância de o_i da seguinte forma:

$$\begin{aligned}
E[o_i] & = \sum_{j=1}^{n_\mathrm{in}} E[w_{ij} x_j] \\
&= \sum_{j=1}^{n_\mathrm{in}} E[w_{ij}] E[x_j] \\
&= 0, \\
\mathrm{Var}[o_i] & = E[o_i^2] - (E[o_i])^2 \\
& = \sum_{j=1}^{n_\mathrm{in}} E[w^2_{ij} x^2_j] - 0 \\
& = \sum_{j=1}^{n_\mathrm{in}} E[w^2_{ij}] E[x^2_j] \\
& = n_\mathrm{in} \sigma^2 \gamma^2.
\end{aligned} \tag{4.8.4}$$

Uma maneira de manter a variância fixa é definir $n_\mathrm{in} \sigma^2 = 1$. Agora, considere a retropropagação. Lá nós enfrentamos um problema semelhante, embora com gradientes sendo propagados das camadas mais próximas da saída. Usando o mesmo raciocínio da propagação direta, vemos que a variância dos gradientes pode explodir a menos que $n_\mathrm{out} \sigma^2 = 1$, onde n_out é o número de saídas desta camada. Isso nos deixa em um dilema: não podemos satisfazer ambas as condições simultaneamente. Em vez disso, simplesmente tentamos satisfazer:

$$\frac{1}{2} (n_\mathrm{in} + n_\mathrm{out}) \sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_\mathrm{in} + n_\mathrm{out}}}. \tag{4.8.5}$$

Este é o raciocínio subjacente à agora padrão e praticamente benéfica *inicialização de Xavier*, em homenagem ao primeiro autor de seus criadores (Glorot & Bengio, 2010). Normalmente, a inicialização de Xavier amostra pesos de uma distribuição gaussiana com média e variância zero $\sigma^2 = \frac{2}{n_\mathrm{in} + n_\mathrm{out}}$. Também podemos adaptar a intuição de Xavier para escolher a variância ao amostrar os pesos de uma distribuição uniforme. Observe que a distribuição uniforme $U(-a, a)$ tem variância $\frac{a^2}{3}$. Conectar $\frac{a^2}{3}$ em nossa condição em σ^2 produz a sugestão de inicializar de acordo com

$$U\left(-\sqrt{\frac{6}{n_\mathrm{in} + n_\mathrm{out}}}, \sqrt{\frac{6}{n_\mathrm{in} + n_\mathrm{out}}}\right). \tag{4.8.6}$$

Embora a suposição de inexistência de não linearidades no raciocínio matemático acima pode ser facilmente violada em redes neurais, o método de inicialização de Xavier acaba funcionando bem na prática.

Além

O raciocínio acima mal arranha a superfície de abordagens modernas para inicialização de parâmetros. Uma estrutura de *deep learning* geralmente implementa mais de uma dúzia de heurísticas diferentes. Além disso, a inicialização do parâmetro continua a ser uma área quente de pesquisa fundamental em *deep learning*. Entre elas estão heurísticas especializadas para parâmetros vinculados (compartilhados), super-resolução, modelos de sequência e outras situações. Por exemplo, Xiao et al. demonstraram a possibilidade de treinar Redes neurais de 10.000 camadas

sem truques arquitetônicos usando um método de inicialização cuidadosamente projetado (Xiao et al., 2018).

Se o assunto interessar a você, sugerimos um mergulho profundo nas ofertas deste módulo, lendo os artigos que propuseram e analisaram cada heurística, e explorando as publicações mais recentes sobre o assunto. Talvez você tropece ou até invente uma ideia inteligente e contribuir com uma implementação para estruturas de *deep learning*.

4.8.3 Resumo

- Desaparecimento e explosão de gradientes são problemas comuns em redes profundas. É necessário muito cuidado na inicialização dos parâmetros para garantir que gradientes e parâmetros permaneçam bem controlados.
- As heurísticas de inicialização são necessárias para garantir que os gradientes iniciais não sejam nem muito grandes nem muito pequenos.
- As funções de ativação ReLU atenuam o problema do desaparecimento de gradiente. Isso pode acelerar a convergência.
- A inicialização aleatória é a chave para garantir que a simetria seja quebrada antes da otimização.
- A inicialização de Xavier sugere que, para cada camada, a variação de qualquer saída não é afetada pelo número de entradas e a variação de qualquer gradiente não é afetada pelo número de saídas.

4.8.4 Exercícios

1. Você pode projetar outros casos em que uma rede neural pode exibir simetria exigindo quebra, além da simetria de permutação nas camadas de um MLP?
2. Podemos inicializar todos os parâmetros de peso na regressão linear ou na regressão *softmax* para o mesmo valor?
3. Procure limites analíticos nos autovalores do produto de duas matrizes. O que isso diz a você sobre como garantir que os gradientes sejam bem condicionados?
4. Se sabemos que alguns termos divergem, podemos consertar isso após o fato? Veja o artigo sobre escalonamento de taxa adaptável em camadas para se inspirar (You et al., 2017).

Discussions[54]

[54] https://discuss.d2l.ai/t/104

4.9 Mudança de Ambiente e Distribuição

Nas seções anteriores, trabalhamos uma série de aplicações práticas de *machine learning*, ajustando modelos a uma variedade de conjuntos de dados. E, no entanto, nunca paramos para contemplar de onde vêm os dados em primeiro lugar ou o que planejamos fazer com as saídas de nossos modelos. Muitas vezes, desenvolvedores de *machine learning* na posse de pressa de dados para desenvolver modelos, não param para considerar essas questões fundamentais.

Muitas implantações de *machine learning* com falha podem ser rastreadas até este padrão. Às vezes, os modelos parecem ter um desempenho maravilhoso conforme medido pela precisão do conjunto de teste mas falham catastroficamente na implantação quando a distribuição de dados muda repentinamente. Mais insidiosamente, às vezes a própria implantação de um modelo pode ser o catalisador que perturba a distribuição de dados. Digamos, por exemplo, que treinamos um modelo para prever quem vai pagar em comparação com o inadimplemento de um empréstimo, descobrindo que a escolha de calçado de um candidato foi associado ao risco de inadimplência (Oxfords indicam reembolso, tênis indicam inadimplência). Podemos estar inclinados a, a partir daí, conceder empréstimos a todos os candidatos vestindo Oxfords e negar a todos os candidatos o uso de tênis.

Neste caso, nosso salto imprudente de reconhecimento de padrões para a tomada de decisão e nossa falha em considerar criticamente o ambiente pode ter consequências desastrosas. Para começar, assim que começamos tomar decisões com base em calçados, os clientes perceberiam e mudariam seu comportamento. Em pouco tempo, todos os candidatos estariam usando Oxfords, sem qualquer melhoria coincidente na capacidade de crédito. Dedique um minuto para digerir isso, porque há muitos problemas semelhantes em muitas aplicações de *machine learning*: introduzindo nossas decisões baseadas em modelos para o ambiente, podemos quebrar o modelo.

Embora não possamos dar a esses tópicos um tratamento completo em uma seção, pretendemos aqui expor algumas preocupações comuns, e estimular o pensamento crítico necessário para detectar essas situações precocemente, mitigar os danos e usar o *machine learning* com responsabilidade. Algumas das soluções são simples (peça os dados "corretos"), alguns são tecnicamente difíceis (implementar um sistema de aprendizagem por reforço), e outros exigem que saiamos do reino de previsão estatística em conjunto e lidemos com difíceis questões filosóficas relativas à aplicação ética de algoritmos.

4.9.1 Tipos de Turno de Distribuição

Para começar, ficamos com a configuração de predição passiva considerando as várias maneiras que as distribuições de dados podem mudar e o que pode ser feito para salvar o desempenho do modelo. Em uma configuração clássica, assumimos que nossos dados de treinamento foram amostrados de alguma distribuição $p_S(\mathbf{x}, y)$ mas que nossos dados de teste consistirão de exemplos não rotulados retirados de alguma distribuição diferente $p_T(\mathbf{x}, y)$. Já, devemos enfrentar uma realidade preocupante. Ausentes quaisquer suposições sobre como p_S e p_T se relacionam entre si, aprender um classificador robusto é impossível.

Considere um problema de classificação binária, onde desejamos distinguir entre cães e gatos. Se a distribuição pode mudar de forma arbitrária, então nossa configuração permite o caso patológico em que a distribuição sobre os insumos permanece constante: $p_S(\mathbf{x}) = p_T(\mathbf{x})$, mas os *labels* estão todos invertidos: $p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$. Em outras palavras, se Deus pode decidir de repente que no futuro todos os "gatos" agora são cachorros e o que anteriormente chamamos de "cães" agora são gatos — sem qualquer mudança na distribuição de entradas $p(\mathbf{x})$, então não podemos distinguir essa configuração de um em que a distribuição não mudou nada.

Felizmente, sob algumas suposições restritas sobre como nossos dados podem mudar no futuro, algoritmos de princípios podem detectar mudanças e às vezes até se adaptam na hora, melhorando a precisão do classificador original.

Mudança Covariável

Entre as categorias de mudança de distribuição, o deslocamento covariável pode ser o mais amplamente estudado. Aqui, assumimos que, embora a distribuição de entradas pode mudar com o tempo, a função de rotulagem, ou seja, a distribuição condicional $P(y \mid \mathbf{x})$ não muda. Os estatísticos chamam isso de *mudança covariável* porque o problema surge devido a um mudança na distribuição das covariáveis (*features*). Embora às vezes possamos raciocinar sobre a mudança de distribuição sem invocar causalidade, notamos que a mudança da covariável é a suposição natural para invocar nas configurações onde acreditamos que $\mathbf{x}$ causa y.

Considere o desafio de distinguir cães e gatos. Nossos dados de treinamento podem consistir em imagens do tipo em Fig. 4.9.1.

Fig. 4.9.1: Dados de treinamento para distinguir cães e gatos.

No momento do teste, somos solicitados a classificar as imagens em Fig. 4.9.2.

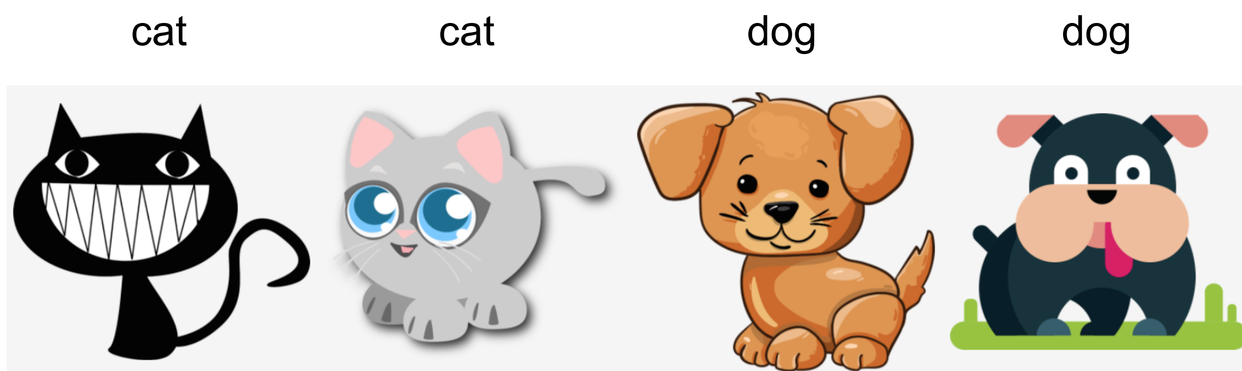


Fig. 4.9.2: Dados de teste para distinguir cães e gatos.

O conjunto de treinamento consiste em fotos, enquanto o conjunto de teste contém apenas desenhos animados. Treinamento em um conjunto de dados com características do conjunto de teste pode significar problemas na ausência de um plano coerente para saber como se adaptar ao novo domínio.

Mudança de *Label*

A *Mudança de Label* descreve o problema inverso. Aqui, assumimos que o rótulo marginal $P(y)$ pode mudar mas a distribuição condicional de classe $P(\mathbf{x} \mid y)$ permanece fixa nos domínios. A mudança de *label* é uma suposição razoável a fazer quando acreditamos que y causa $\mathbf{x}$. Por exemplo, podemos querer prever diagnósticos dados seus sintomas (ou outras manifestações), mesmo enquanto a prevalência relativa de diagnósticos esteja mudando com o tempo. A mudança de rótulo é a suposição apropriada aqui porque as doenças causam sintomas. Em alguns casos degenerados, a mudança de rótulo e as suposições de mudança de covariável podem ser mantidas simultaneamente. Por exemplo, quando o rótulo é determinístico, a suposição de mudança da covariável será satisfeita, mesmo quando y causa $\mathbf{x}$. Curiosamente, nesses casos, muitas vezes é vantajoso trabalhar com métodos que fluem da suposição de mudança de rótulo. Isso ocorre porque esses métodos tendem envolver a manipulação de objetos que se parecem com rótulos (muitas vezes de baixa dimensão), ao contrário de objetos que parecem entradas, que tendem a ser altamente dimensionais no *deep learning*.

Mudança de Conceito

Também podemos encontrar o problema relacionado de *mudança de conceito*, que surge quando as próprias definições de rótulos podem mudar. Isso soa estranho — um *gato* é um *gato*, não? No entanto, outras categorias estão sujeitas a mudanças no uso ao longo do tempo. Critérios de diagnóstico para doença mental, o que passa por moda e cargos, estão todos sujeitos a consideráveis quantidades de mudança de conceito. Acontece que se navegarmos pelos Estados Unidos, mudando a fonte de nossos dados por geografia, encontraremos uma mudança considerável de conceito em relação a distribuição de nomes para *refrigerantes* como mostrado em Fig. 4.9.3.

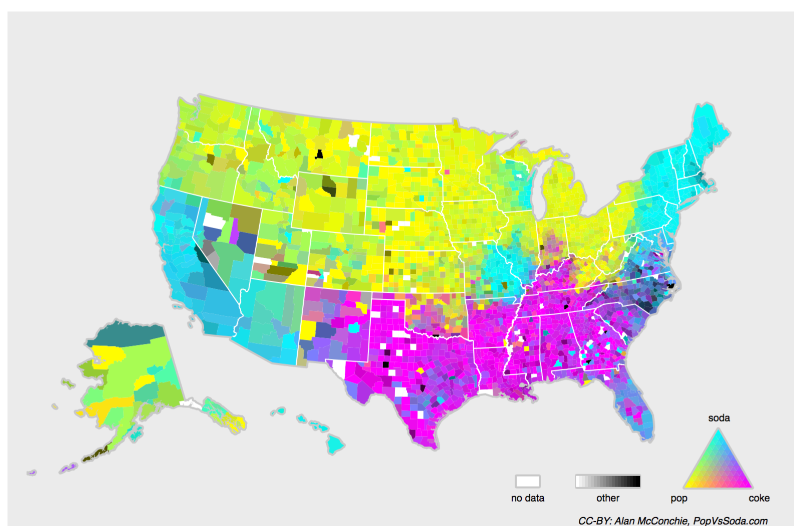


Fig. 4.9.3: Mudança de conceito em nomes de refrigerantes nos Estados Unidos.

Se fossemos construir um sistema de tradução automática, a distribuição $P(y \mid \mathbf{x})$ pode ser diferente dependendo de nossa localização. Esse problema pode ser difícil de detectar. Podemos ter esperança de explorar o conhecimento cuja mudança só ocorre gradualmente seja em um sentido temporal ou geográfico.

4.9.2 Exemplos de Mudança de Distribuição

Antes de mergulhar no formalismo e algoritmos, podemos discutir algumas situações concretas onde a covariável ou mudança de conceito pode não ser óbvia.

Diagnóstico Médico

Imagine que você deseja criar um algoritmo para detectar o câncer. Você coleta dados de pessoas saudáveis e doentes e você treina seu algoritmo. Funciona bem, oferecendo alta precisão e você conclui que está pronto para uma carreira de sucesso em diagnósticos médicos. *Não tão rápido.*

As distribuições que deram origem aos dados de treinamento e aqueles que você encontrará na natureza podem diferir consideravelmente. Isso aconteceu com uma inicialização infeliz que alguns de nós (autores) trabalhamos anos atrás. Eles estavam desenvolvendo um exame de sangue para uma doença que afeta predominantemente homens mais velhos e esperava estudá-lo usando amostras de sangue que eles haviam coletado de pacientes. No entanto, é consideravelmente mais difícil obter amostras de sangue de homens saudáveis do que pacientes doentes já no sistema. Para compensar, a *startup* solicitou doações de sangue de estudantes em um campus universitário para servir como controles saudáveis no desenvolvimento de seu teste. Então eles perguntaram se poderíamos ajudá-los a construir um classificador para detecção da doença.

Como explicamos a eles, seria realmente fácil distinguir entre as coortes saudáveis e doentes com precisão quase perfeita. No entanto, isso ocorre porque os assuntos de teste diferiam em idade, níveis hormonais, atividade física, dieta, consumo de álcool, e muitos outros fatores não relacionados à doença. Era improvável que fosse o caso com pacientes reais. Devido ao seu procedimento de amostragem, poderíamos esperar encontrar mudanças extremas covariadas. Além disso, este caso provavelmente não seria corrigível por meio de métodos convencionais. Resumindo, eles desperdiçaram uma quantia significativa de dinheiro.

Carros Autônomos

Digamos que uma empresa queira aproveitar o *machine learning* para o desenvolvimento de carros autônomos. Um componente chave aqui é um detector de beira de estrada. Uma vez que dados anotados reais são caros de se obter, eles tiveram a ideia (inteligente e questionável) de usar dados sintéticos de um motor de renderização de jogo como dados de treinamento adicionais. Isso funcionou muito bem em "dados de teste" extraídos do mecanismo de renderização. Infelizmente, dentro de um carro de verdade foi um desastre. Como se viu, a beira da estrada havia sido renderizada com uma textura muito simplista. Mais importante, *todo* o acostamento havia sido renderizado com a *mesma* textura e o detector de beira de estrada aprendeu sobre essa "característica" muito rapidamente.

Algo semelhante aconteceu com o Exército dos EUA quando eles tentaram detectar tanques na floresta pela primeira vez. Eles tiraram fotos aéreas da floresta sem tanques, em seguida, dirigiram os tanques para a floresta e tiraram outro conjunto de fotos. O classificador pareceu funcionar *perfeitamente*. Infelizmente, ele apenas aprendeu como distinguir árvores com sombras de árvores sem sombras — o primeiro conjunto de fotos foi tirado no início da manhã, o segundo conjunto ao meio-dia.

Distribuições Não-estacionárias

Surge uma situação muito mais sutil quando a distribuição muda lentamente (também conhecido como *distribuição não-estacionária*) e o modelo não é atualizado de forma adequada. Abaixo estão alguns casos típicos.

- Treinamos um modelo de publicidade computacional e deixamos de atualizá-lo com frequência (por exemplo, esquecemos de incorporar que um novo dispositivo obscuro chamado iPad acabou de ser lançado).
- Construímos um filtro de spam. Ele funciona bem na detecção de todos os spams que vimos até agora. Mas então os spammers se tornaram mais inteligentes e criaram novas mensagens que se parecem com tudo o que vimos antes.
- Construímos um sistema de recomendação de produtos. Ele funciona durante todo o inverno, mas continua a recomendar chapéus de Papai Noel muito depois do Natal.

Mais Anedotas

- Construímos um detector de rosto. Funciona bem em todos os *benchmarks*. Infelizmente, ele falha nos dados de teste — os exemplos ofensivos são closes em que o rosto preenche a imagem inteira (nenhum dado desse tipo estava no conjunto de treinamento).
- Construímos um mecanismo de busca na Web para o mercado dos EUA e queremos implantá-lo no Reino Unido.
- Treinamos um classificador de imagens compilando um grande conjunto de dados onde cada um entre um grande conjunto de classes é igualmente representado no conjunto de dados, digamos 1000 categorias, representadas por 1000 imagens cada. Em seguida, implantamos o sistema no mundo real, onde a distribuição real do rótulo das fotos é decididamente não uniforme.

4.9.3 Correção de Mudança de Distribuição

Como já discutimos, existem muitos casos onde distribuições de treinamento e teste $P(\mathbf{x}, y)$ são diferentes. Em alguns casos, temos sorte e os modelos funcionam apesar da covariável, rótulo ou mudança de conceito. Em outros casos, podemos fazer melhor empregando estratégias baseadas em princípios para lidar com a mudança. O restante desta seção torna-se consideravelmente mais técnico. O leitor impaciente pode continuar na próxima seção já que este material não é pré-requisito para conceitos subsequentes.

Risco Empírico e Risco

Vamos primeiro refletir sobre o que exatamente está acontecendo durante o treinamento do modelo: nós iteramos sobre recursos e rótulos associados de dados de treinamento $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ e atualizamos os parâmetros de um modelo f após cada *minibatch*. Para simplificar, não consideramos regularização, portanto, minimizamos amplamente a perda no treinamento:

$$\mathop{\mathrm{minimizar}}_f \frac{1}{n} \sum_{i=1}^n l(f(\mathbf{x}_i), y_i), \tag{4.9.1}$$

onde l é a função de perda medir "quão ruim" a previsão $f(\mathbf{x}_i)$ recebe o rótulo associado y_i. Os estatísticos chamam o termo em (4.9.1) *risco empírico*. O *risco empírico* é uma perda média sobre os dados de treinamento para aproximar o *risco*, que é a expectativa de perda sobre toda a população de dados extraídos de sua verdadeira distribuição $p(\mathbf{x}, y)$:

$$E_{p(\mathbf{x}, y)} [l(f(\mathbf{x}), y)] = \int\int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \;d\mathbf{x}dy. \tag{4.9.2}$$

No entanto, na prática, normalmente não podemos obter toda a população de dados. Assim, a *minimização de risco empírico*, que está minimizando o risco empírico em (4.9.1), é uma estratégia prática para *machine learning*, com a esperança de aproximar minimizando o risco.

Covariate Shift Correction

Suponha que queremos estimar algumas dependências $P(y \mid \mathbf{x})$ para as quais rotulamos os dados $(\mathbf{x}_i, y_i)$. Infelizmente, as observações $\mathbf{x}_i$ são desenhadas de alguma *distribuição de origem* q(**:raw-latex:`\mathbf{x}`**)$ em vez da *distribuição de destino* $p(\mathbf{x})$. Felizmente, a suposição de dependência significa que a distribuição condicional não muda: $p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$. Se a distribuição de origem $q(\mathbf{x})$ está "errada", podemos corrigir isso usando a seguinte identidade simples no risco:

$$\int\int l(f(\mathbf{x}), y) p(y \mid \mathbf{x})p(\mathbf{x}) \;d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y) q(y \mid \mathbf{x})q(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})} \;d\mathbf{x}dy. \tag{4.9.3}$$

Em outras palavras, precisamos pesar novamente cada exemplo de dados pela proporção do probabilidade que teria sido extraída da distribuição correta para a errada:

$$\beta_i \stackrel{\mathrm{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{4.9.4}$$

Conectando o peso β_i para cada exemplo de dados $(\mathbf{x}_i, y_i)$ podemos treinar nosso modelo usando *minimização de risco empírico ponderado*:

$$\mathop{\mathrm{minimizar}}_f \frac{1}{n} \sum_{i=1}^n \beta_i l(f(\mathbf{x}_i), y_i). \tag{4.9.5}$$

Infelizmente, não sabemos essa proporção, portanto, antes de fazermos qualquer coisa útil, precisamos estimá-la. Muitos métodos estão disponíveis, incluindo algumas abordagens teóricas de operador extravagantes que tentam recalibrar o operador de expectativa diretamente usando uma norma mínima ou um princípio de entropia máxima. Observe que, para qualquer abordagem desse tipo, precisamos de amostras extraídas de ambas as distribuições — o "verdadeiro" p, por exemplo, por acesso aos dados de teste, e aquele usado para gerar o conjunto de treinamento q (o último está trivialmente disponível). Observe, entretanto, que só precisamos dos recursos $\mathbf{x} \sim p(\mathbf{x})$; não precisamos acessar os rótulos $y \sim p(y)$.

Neste caso, existe uma abordagem muito eficaz que dará resultados quase tão bons quanto a original: regressão logística, que é um caso especial de regressão *softmax* (ver Section 3.4) para classificação binária. Isso é tudo o que é necessário para calcular as razões de probabilidade estimadas. Aprendemos um classificador para distinguir entre os dados extraídos de $p(\mathbf{x})$ e dados extraídos de $q(\mathbf{x})$. Se é impossível distinguir entre as duas distribuições então isso significa que as instâncias associadas são igualmente prováveis de virem de qualquer uma das duas distribuições. Por outro lado, quaisquer instâncias que podem ser bem discriminadas devem ser significativamente sobreponderadas ou subponderadas em conformidade.

Para simplificar, suponha que temos um número igual de instâncias de ambas as distribuições $p(\mathbf{x})$ e $q(\mathbf{x})$,, respectivamente. Agora denote por z rótulos que são 1 para dados extraídos de p e -1 para dados extraídos de q. Então, a probabilidade em um conjunto de dados misto é dada por

$$P(z=1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x})+q(\mathbf{x})} \text{ e portanto } \frac{P(z=1 \mid \mathbf{x})}{P(z=-1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.9.6}$$

Assim, se usarmos uma abordagem de regressão logística, onde $P(z=1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$ (h é uma função parametrizada), segue que

$$\beta_i = \frac{1/(1+\exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1+\exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{4.9.7}$$

Como resultado, precisamos resolver dois problemas: primeiro a distinguir entre dados extraídos de ambas as distribuições, e, em seguida, um problema de minimização de risco empírico ponderado em (4.9.5) onde pesamos os termos em β_i..

Agora estamos prontos para descrever um algoritmo de correção. Suponha que temos um conjunto de treinamento $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ e um conjunto de teste não rotulado $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$. Para mudança de covariável, assumimos que $\mathbf{x}_i$ para todos os $1 \leq i \leq n$ são retirados de alguma distribuição de origem e $\mathbf{u}_i$ for all $1 \leq i \leq m$ são retirados da distribuição de destino. Aqui está um algoritmo prototípico para corrigir a mudança da covariável:

1. Gere um conjunto de treinamento de classificação binária: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Treine um classificador binário usando regressão logística para obter a função h.
3. Pese os dados de treinamento usando $\beta_i = \exp(h(\mathbf{x}_i))$ ou melhor \$$\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ para alguma constante c.
4. Use pesos β_i para treinar em $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ em (4.9.5).

Observe que o algoritmo acima se baseia em uma suposição crucial. Para que este esquema funcione, precisamos que cada exemplo de dados na distribuição de destino (por exemplo, tempo de teste) tenha probabilidade diferente de zero de ocorrer no momento do treinamento. Se encontrarmos um ponto onde $p(\mathbf{x}) > 0$ mas $q(\mathbf{x}) = 0$, então, o peso de importância correspondente deve ser infinito.

Correção de Mudança de *Label*

Suponha que estamos lidando com um tarefa de classificação com k categorias. Usando a mesma notação em Section 4.9.3, q e p são as distribuições de origem (por exemplo, tempo de treinamento) e a distribuição de destino (por exemplo, tempo de teste), respectivamente. Suponha que a distribuição dos rótulos mude ao longo do tempo: $q(y) \neq p(y)$, mas a distribuição condicional de classe permanece a mesma: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$. Se a distribuição de origem $q(y)$ estiver "errada", nós podemos corrigir isso de acordo com a seguinte identidade no risco conforme definido em :eqref: `eq_true-risk`:

$$\int\int l(f(\mathbf{x}), y)p(\mathbf{x} \mid y)p(y) \; d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y)q(\mathbf{x} \mid y)q(y)\frac{p(y)}{q(y)} \; d\mathbf{x}dy. \tag{4.9.8}$$

Aqui, nossos pesos de importância corresponderão às taxas de probabilidade de rótulo

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{4.9.9}$$

Uma coisa boa sobre a mudança de rótulo é que se tivermos um modelo razoavelmente bom na distribuição de origem, então podemos obter estimativas consistentes desses pesos sem nunca ter que lidar com a dimensão ambiental. No aprendizado profundo, as entradas tendem a ser objetos de alta dimensão, como imagens, enquanto os rótulos são frequentemente objetos mais simples, como categorias.

Para estimar a distribuição de rótulos de destino, primeiro pegamos nosso classificador de prateleira razoavelmente bom (normalmente treinado nos dados de treinamento) e calculamos sua matriz de confusão usando o conjunto de validação (também da distribuição de treinamento). A *matriz de confusão*, **C**, é simplesmente uma matriz $k \times k$, onde cada coluna corresponde à categoria do rótulo (informações básicas) e cada linha corresponde à categoria prevista do nosso modelo. O valor de cada célula c_{ij} é a fração do total de previsões no conjunto de validação onde o verdadeiro rótulo era j e nosso modelo previu i.

Agora, não podemos calcular a matriz de confusão nos dados de destino diretamente, porque não conseguimos ver os rótulos dos exemplos que vemos na natureza, a menos que invistamos em um pipeline de anotação em tempo real complexo. O que podemos fazer, no entanto, é calcular a média de todas as nossas previsões de modelos no momento do teste juntas, produzindo os resultados médios do modelo $\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$, cujo i^th elemento $\mu(\hat{y}_i)$ é a fração das previsões totais no conjunto de teste onde nosso modelo previu i.

Acontece que sob algumas condições amenas — se nosso classificador era razoavelmente preciso em primeiro lugar, e se os dados alvo contiverem apenas categorias que vimos antes, e se a suposição de mudança de rótulo se mantém em primeiro lugar (a suposição mais forte aqui), então podemos estimar a distribuição do rótulo do conjunto de teste resolvendo um sistema linear simples

$$\mathbf{C} p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{4.9.10}$$

porque como uma estimativa $\sum_{j=1}^k c_{ij} p(y_j) = \mu(\hat{y}_i)$ vale para todos $1 \leq i \leq k$, onde $p(y_j)$ é o elemento j^th do vetor de distribuição de rótulo k-dimensional $p(\mathbf{y})$. Se nosso classificador é suficientemente preciso para começar, então a matriz de confusão **C** será invertível, e obtemos uma solução $p(\mathbf{y}) = \mathbf{C}^{-1} \mu(\hat{\mathbf{y}})$.

Porque observamos os rótulos nos dados de origem, é fácil estimar a distribuição $q(y)$. Então, para qualquer exemplo de treinamento i com rótulo y_i, podemos pegar a razão de nossa estimativa de $p(y_i)/q(y_i)$ para calcular o peso β_i, e conecter isso à minimização de risco empírico ponderado em (4.9.5).

Correção da Mudança de Conceito

A mudança de conceito é muito mais difícil de corrigir com base em princípios. Por exemplo, em uma situação em que de repente o problema muda de distinguir gatos de cães para um de distinguir animais brancos de negros, não será razoável supor que podemos fazer muito melhor do que apenas coletar novos rótulos e treinar do zero. Felizmente, na prática, essas mudanças extremas são raras. Em vez disso, o que geralmente acontece é que a tarefa continua mudando lentamente. Para tornar as coisas mais concretas, aqui estão alguns exemplos:

- Na publicidade computacional, novos produtos são lançados, produtos antigos tornam-se menos populares. Isso significa que a distribuição dos anúncios e sua popularidade mudam gradualmente e qualquer preditor de taxa de cliques precisa mudar gradualmente com isso.
- As lentes das câmeras de trânsito degradam-se gradualmente devido ao desgaste ambiental, afetando progressivamente a qualidade da imagem.

- O conteúdo das notícias muda gradualmente (ou seja, a maioria das notícias permanece inalterada, mas novas histórias aparecem).

Nesses casos, podemos usar a mesma abordagem que usamos para treinar redes para fazê-las se adaptarem à mudança nos dados. Em outras palavras, usamos os pesos de rede existentes e simplesmente executamos algumas etapas de atualização com os novos dados, em vez de treinar do zero.

4.9.4 Uma taxonomia de Problemas de Aprendizagem

Munidos do conhecimento sobre como lidar com as mudanças nas distribuições, podemos agora considerar alguns outros aspectos da formulação do problema de *machine learning*.

Aprendizagem em Lote

Na *aprendizagem em lote*, temos acesso aos recursos de treinamento e rótulos $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, que usamos para treinar um modelo $f(\mathbf{x})$. Posteriormente, implementamos esse modelo para pontuar novos dados $(\mathbf{x}, y)$ extraídos da mesma distribuição. Esta é a suposição padrão para qualquer um dos problemas que discutimos aqui. Por exemplo, podemos treinar um detector de gatos com base em muitas fotos de cães e gatos. Depois de treiná-lo, nós o enviamos como parte de um sistema de visão computacional de porta de gato inteligente que permite a entrada apenas de gatos. Ele é então instalado na casa do cliente e nunca mais atualizado (exceto em circunstâncias extremas).

Aprendizado *Online*

Agora imagine que os dados $(\mathbf{x}_i, y_i)$ chegam em uma amostra de cada vez. Mais especificamente, suponha que primeiro observamos $\mathbf{x}_i$, então precisamos chegar a uma estimativa $f(\mathbf{x}_i)$ e somente depois de fazermos isso, observamos y_i e com isso, recebemos uma recompensa ou incorremos em uma perda, dada a nossa decisão. Muitos problemas reais se enquadram nesta categoria. Por exemplo, precisamos prever o preço das ações de amanhã, o que nos permite negociar com base nessa estimativa e, no final do dia, descobrimos se nossa estimativa nos permitiu obter lucro. Em outras palavras, em *aprendizagem online*, temos o seguinte ciclo, onde estamos continuamente melhorando nosso modelo a partir de novas observações.

$$\mathrm{model} \ f_t \longrightarrow \mathrm{data} \ \mathbf{x}_t \longrightarrow \mathrm{estimate} \ f_t(\mathbf{x}_t) \longrightarrow \mathrm{observation} \ y_t \longrightarrow \mathrm{loss} \ l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \mathrm{model} \ f_{t+1} \tag{4.9.11}$$

Bandits

Bandits são um caso especial do problema acima. Enquanto na maioria dos problemas de aprendizagem temos uma função parametrizada continuamente f onde queremos aprender seus parâmetros (por exemplo, uma rede profunda), em um problema *bandit* temos apenas um número finito de braços que podemos puxar, ou seja, um número finito de ações que podemos realizar. Não é muito surpreendente que, para este problema mais simples, possam ser obtidas garantias teóricas mais fortes em termos de otimização. Listamos principalmente porque esse problema é frequentemente (confusamente) tratado como se fosse um ambiente de aprendizagem distinto.

Controle

Em muitos casos, o ambiente lembra o que fizemos. Não necessariamente de maneira adversa, mas apenas lembrará e a resposta dependerá do que aconteceu antes. Por exemplo, um controlador de caldeira de café observará diferentes temperaturas dependendo se estava aquecendo a caldeira anteriormente. Os algoritmos do controlador PID (proporcional derivativo e integral) são uma escolha popular. Da mesma forma, o comportamento de um usuário em um site de notícias dependerá do que mostramos a ele anteriormente (por exemplo, ele lerá a maioria das notícias apenas uma vez). Muitos desses algoritmos formam um modelo do ambiente no qual atuam, de modo que suas decisões parecem menos aleatórias. Recentemente, a teoria de controle (por exemplo, variantes PID) também foi usada para ajustar hiperparâmetros automaticamente para obter uma melhor qualidade de desemaranhamento e reconstrução, e melhorar a diversidade do texto gerado e a qualidade da reconstrução das imagens geradas (Shao et al., 2020).

Aprendizagem por Reforço

No caso mais geral de um ambiente com memória, podemos encontrar situações em que o ambiente está tentando cooperar conosco (jogos cooperativos, em particular para jogos de soma não zero), ou outras em que o ambiente tentará vencer. Xadrez, Go, Backgammon ou StarCraft são alguns dos casos de *aprendizagem por reforço*. Da mesma forma, podemos querer construir um bom controlador para carros autônomos. Os outros carros tendem a responder ao estilo de direção do carro autônomo de maneiras não triviais, por exemplo, tentando evitá-lo, tentando causar um acidente e tentando cooperar com ele.

Considerando o Ambiente

Uma distinção importante entre as diferentes situações acima é que a mesma estratégia que pode ter funcionado no caso de um ambiente estacionário pode não funcionar quando o ambiente pode se adaptar. Por exemplo, uma oportunidade de arbitragem descoberta por um comerciante provavelmente desaparecerá assim que ele começar a explorá-la. A velocidade e a maneira como o ambiente muda determinam em grande parte o tipo de algoritmos que podemos utilizar. Por exemplo, se sabemos que as coisas só podem mudar lentamente, podemos forçar qualquer estimativa a mudar apenas lentamente também. Se soubermos que o ambiente pode mudar instantaneamente, mas muito raramente, podemos fazer concessões a isso. Esses tipos de conhecimento são cruciais para o aspirante a cientista de dados lidar com a mudança de conceito, ou seja, quando o problema que ele está tentando resolver muda ao longo do tempo.

4.9.5 Justiça, Responsabilidade e Transparência no *Machine Learning*

Finalmente, é importante lembrar que quando você implanta sistemas de *machine learning* você não está apenas otimizando um modelo preditivo — você normalmente fornece uma ferramenta que irá ser usada para automatizar (parcial ou totalmente) as decisões. Esses sistemas técnicos podem impactar as vidas de indivíduos sujeitos às decisões resultantes. O salto da consideração das previsões para as decisões levanta não apenas novas questões técnicas, mas também uma série de questões éticas isso deve ser considerado cuidadosamente. Se estivermos implantando um sistema de diagnóstico médico, precisamos saber para quais populações pode funcionar e pode não funcionar. Negligenciar riscos previsíveis para o bem-estar de uma subpopulação pode nos levar a administrar cuidados inferiores. Além disso, uma vez que contemplamos os sistemas de

tomada de decisão, devemos recuar e reconsiderar como avaliamos nossa tecnologia. Entre outras consequências desta mudança de escopo, descobriremos que a *exatidão* raramente é a medida certa. Por exemplo, ao traduzir previsões em ações, muitas vezes queremos levar em consideração a potencial sensibilidade ao custo de errar de várias maneiras. Se uma maneira de classificar erroneamente uma imagem pode ser percebida como um truque racial, enquanto a classificação incorreta para uma categoria diferente seria inofensiva, então podemos querer ajustar nossos limites de acordo, levando em consideração os valores sociais na concepção do protocolo de tomada de decisão. Também queremos ter cuidado com como os sistemas de previsão podem levar a ciclos de *feedback*. Por exemplo, considere sistemas de policiamento preditivo, que alocam policiais de patrulha para áreas com alta previsão de crime. É fácil ver como um padrão preocupante pode surgir:

1. Bairros com mais crimes recebem mais patrulhas.
2. Consequentemente, mais crimes são descobertos nesses bairros, inserindo os dados de treinamento disponíveis para iterações futuras.
3. Exposto a mais aspectos positivos, o modelo prevê ainda mais crimes nesses bairros.
4. Na próxima iteração, o modelo atualizado visa a mesma vizinhança ainda mais fortemente, levando a ainda mais crimes descobertos, etc.

Freqüentemente, os vários mecanismos pelos quais as previsões de um modelo são acopladas a seus dados de treinamento não são contabilizados no processo de modelagem. Isso pode levar ao que os pesquisadores chamam de *ciclos de feedback descontrolados*. Além disso, queremos ter cuidado com se estamos tratando do problema certo em primeiro lugar. Algoritmos preditivos agora desempenham um papel descomunal na mediação da disseminação de informações. A notícia de que um indivíduo encontra ser determinado pelo conjunto de páginas do Facebook de que *Gostou*? Estes são apenas alguns entre os muitos dilemas éticos urgentes que você pode encontrar em uma carreira em *machine learning*.

4.9.6 Resumo

- Em muitos casos, os conjuntos de treinamento e teste não vêm da mesma distribuição. Isso é chamado de mudança de distribuição.
- O risco é a expectativa de perda sobre toda a população de dados extraídos de sua distribuição real. No entanto, toda essa população geralmente não está disponível. O risco empírico é uma perda média sobre os dados de treinamento para aproximar o risco. Na prática, realizamos a minimização empírica do risco.
- De acordo com as premissas correspondentes, a covariável e a mudança de rótulo podem ser detectadas e corrigidas no momento do teste. Deixar de levar em consideração esse *bias* pode se tornar problemático no momento do teste.
- Em alguns casos, o ambiente pode se lembrar de ações automatizadas e responder de maneiras surpreendentes. Devemos levar em conta essa possibilidade ao construir modelos e continuar a monitorar sistemas ativos, abertos à possibilidade de que nossos modelos e o ambiente se enredem de maneiras imprevistas.

4.9.7 Exercícios

1. O que pode acontecer quando mudamos o comportamento de um mecanismo de pesquisa? O que os usuários podem fazer? E os anunciantes?
2. Implemente um detector de deslocamento covariável. Dica: construa um classificador.
3. Implemente um corretor de mudança covariável.
4. Além da mudança de distribuição, o que mais poderia afetar a forma como o risco empírico se aproxima do risco?

Discussions[55]

4.10 Previsão de Preços de Imóveis no Kaggle

Agora que apresentamos algumas ferramentas básicas para construir e treinar redes profundas e regularizá-las com técnicas, incluindo queda de peso e abandono, estamos prontos para colocar todo esse conhecimento em prática participando de uma competição *Kaggle*. A competição de previsão de preços de casas é um ótimo lugar para começar. Os dados são bastante genéricos e não exibem uma estrutura exótica que podem exigir modelos especializados (como áudio ou vídeo). Este conjunto de dados, coletado por Bart de Cock em 2011 (DeCock, 2011), cobre os preços da habitação em Ames, IA do período de 2006-2010. É consideravelmente maior do que o famoso conjunto de dados de habitação de Boston[56] de Harrison e Rubinfeld (1978), ostentando mais exemplos e mais recursos.

Nesta seção, iremos orientá-lo nos detalhes de pré-processamento de dados, design de modelo e seleção de hiperparâmetros. Esperamos que, por meio de uma abordagem prática, você ganhe algumas intuições que irão guiá-lo em sua carreira como cientista de dados.

4.10.1 *Download* e *Cache* de *datasets*

Ao longo do livro, treinaremos e testaremos modelos em vários conjuntos de dados baixados. Aqui, implementamos várias funções utilitárias para facilitar o *download* de dados. Primeiro, mantemos um dicionário `DATA_HUB` que mapeia uma string (o *nome* do conjunto de dados) a uma tupla contendo o URL para localizar o conjunto de dados e a chave SHA-1 que verifica a integridade do arquivo. Todos esses conjuntos de dados são hospedados no site cujo endereço é `DATA_URL`.

```
import hashlib
import os
import tarfile
import zipfile
import requests

#@save
DATA_HUB = dict()
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

A seguinte função `download` baixa um conjunto de dados, armazenando em cache em um diretório local (`../data` por padrão) e retorna o nome do arquivo baixado. Se um arquivo correspondente a

[55] https://discuss.d2l.ai/t/105
[56] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

este conjunto de dados já existe no diretório de cache e seu SHA-1 corresponde ao armazenado em `DATA_HUB`, nosso código usará o arquivo em cache para evitar obstruir sua internet com *downloads* redundantes.

```
def download(name, cache_dir=os.path.join('..', 'data')):  #@save
    """Download a file inserted into DATA_HUB, return the local filename."""
    assert name in DATA_HUB, f"{name} does not exist in {DATA_HUB}."
    url, sha1_hash = DATA_HUB[name]
    os.makedirs(cache_dir, exist_ok=True)
    fname = os.path.join(cache_dir, url.split('/')[-1])
    if os.path.exists(fname):
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname  # Hit cache
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname
```

Também implementamos duas funções utilitárias adicionais: uma é baixar e extrair um arquivo zip ou tar e o outro para baixar todos os conjuntos de dados usados neste livro de `DATA_HUB` para o diretório de cache.

```
def download_extract(name, folder=None):  #@save
    """Download and extract a zip/tar file."""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir

def download_all():  #@save
    """Download all files in the DATA_HUB."""
    for name in DATA_HUB:
        download(name)
```

4.10.2 *Kaggle*

Kaggle[57] é uma plataforma popular que hospeda competições de *machine learning*. Cada competição se concentra em um conjunto de dados e muitos são patrocinados por interessados que oferecem prêmios para as soluções vencedoras. A plataforma ajuda os usuários a interagir por meio de fóruns e código compartilhado, fomentando a colaboração e a competição. Embora a perseguição ao placar muitas vezes saia do controle, com pesquisadores focando miopicamente nas etapas de pré-processamento em vez de fazer perguntas fundamentais, também há um enorme valor na objetividade de uma plataforma que facilita comparações quantitativas diretas entre abordagens concorrentes, bem como compartilhamento de código para que todos possam aprender o que funcionou e o que não funcionou. Se você quiser participar de uma competição *Kaggle*, primeiro você precisa se registrar para uma conta (veja Fig. 4.10.1).

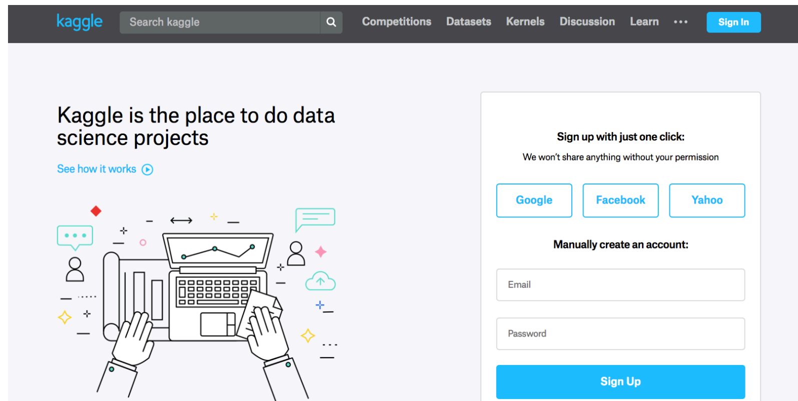


Fig. 4.10.1: Site do Kaggle.

Na página de competição de previsão de preços de casas, conforme ilustrado em Fig. 4.10.2, você pode encontrar o conjunto de dados (na guia "Dados"), enviar previsões e ver sua classificação, A URL está bem aqui:

> https://www.kaggle.com/c/house-prices-advanced-regression-techniques

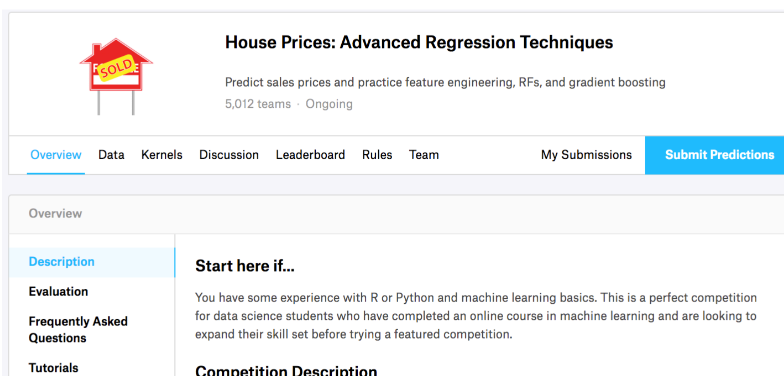


Fig. 4.10.2: A página da competição de previsão de preços de casas.

[57] https://www.kaggle.com

4.10.3 Acessando e Lendo o Conjunto de Dados

Observe que os dados da competição são separados em conjuntos de treinamento e teste. Cada registro inclui o valor da propriedade da casa e atributos como tipo de rua, ano de construção, tipo de telhado, condição do porão, etc. Os recursos consistem em vários tipos de dados. Por exemplo, o ano de construção é representado por um número inteiro, o tipo de telhado por atribuições categóricas discretas, e outros recursos por números de ponto flutuante. E é aqui que a realidade complica as coisas: para alguns exemplos, alguns dados estão ausentes com o valor ausente marcado simplesmente como "na". O preço de cada casa está incluído para o conjunto de treinamento apenas (afinal, é uma competição). Queremos particionar o conjunto de treinamento para criar um conjunto de validação, mas só podemos avaliar nossos modelos no conjunto de teste oficial depois de enviar previsões para Kaggle. A guia "Dados" na guia da competição em Fig. 4.10.2 tem links para baixar os dados.

Para começar, vamos ler e processar os dados usando pandas, que introduzimos em Section 2.2. Então, você vai querer ter certeza de que instalou o pandas antes de prosseguir. Felizmente, se você estiver no Jupyter, podemos instalar pandas sem nem mesmo sair do notebook.

```
# If pandas is not installed, please uncomment the following line:
# !pip install pandas

%matplotlib inline
import numpy as np
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

Por conveniência, podemos baixar e armazenar em cache o conjunto de dados de habitação *Kaggle* usando o *script* que definimos acima.

```
DATA_HUB['kaggle_house_train'] = (  #@save
    DATA_URL + 'kaggle_house_pred_train.csv',
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')

DATA_HUB['kaggle_house_test'] = (  #@save
    DATA_URL + 'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

Usamos pandas para carregar os dois arquivos csv contendo dados de treinamento e teste, respectivamente.

```
train_data = pd.read_csv(download('kaggle_house_train'))
test_data = pd.read_csv(download('kaggle_house_test'))
```

```
Downloading ../data/kaggle_house_pred_train.csv from http://d2l-data.s3-accelerate.amazonaws.
↪com/kaggle_house_pred_train.csv...
Downloading ../data/kaggle_house_pred_test.csv from http://d2l-data.s3-accelerate.amazonaws.
↪com/kaggle_house_pred_test.csv...
```

O conjunto de dados de treinamento inclui 1460 exemplos, 80 características e 1 rótulo, enquanto os dados de teste contém 1459 exemplos e 80 características.

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

Vamos dar uma olhada nos primeiros quatro e nos dois últimos recursos bem como o rótulo (SalePrice) dos primeiros quatro exemplos.

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

Podemos ver que em cada exemplo, a primeira característica é o ID. Isso ajuda o modelo a identificar cada exemplo de treinamento. Embora seja conveniente, ele não carrega qualquer informação para fins de previsão. Portanto, nós o removemos do conjunto de dados antes de alimentar os dados no modelo.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

4.10.4 Pré-processamento de Dados

Conforme declarado acima, temos uma grande variedade de tipos de dados. Precisaremos pré-processar os dados antes de começarmos a modelagem. Vamos começar com as *features* numéricas. Primeiro, aplicamos uma heurística, substituindo todos os valores ausentes pela média da feature correspondente. Então, para colocar todos os recursos em uma escala comum, nós *padronizamos* os dados redimensionando recursos para média zero e variancia unitária:

$$x \leftarrow \frac{x - \mu}{\sigma}, \tag{4.10.1}$$

onde μ e σ denotam média e desvio padrão, respectivamente. Para verificar se isso realmente transforma nossa *feature* (variável) de modo que tenha média zero e variância unitária, observe que $E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$ e que $E[(x-\mu)^2] = (\sigma^2+\mu^2)-2\mu^2+\mu^2 = \sigma^2$. Intuitivamente, padronizamos os dados por duas razões. Primeiro, se mostra conveniente para otimização. Em segundo lugar, porque não sabemos *a priori* quais *features* serão relevantes, não queremos penalizar coeficientes atribuídos a uma *feature* mais do que a qualquer outra.

```
# If test data were inaccessible, mean and standard deviation could be
# calculated from training data
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

Em seguida, lidamos com valores discretos. Isso inclui recursos como "MSZoning". Nós os substituímos por uma codificação one-hot da mesma forma que transformamos anteriormente rótulos multiclasse em vetores (veja Section 3.4.1). Por exemplo, "MSZoning" assume os valores "RL" e "RM". Eliminando a *feature* "MSZoning", duas novas *features* de indicador "MSZoning_RL" e "MSZoning_RM" são criadas com valores 0 ou 1. De acordo com a codificação one-hot, se o valor original de "MSZoning" for "RL", então "MSZoning_RL" é 1 e "MSZoning_RM" é 0. O pacote `pandas` faz isso automaticamente para nós.

```
# `Dummy_na=True` considers "na" (missing value) as a valid feature value, and
# creates an indicator feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

Você pode ver que essa conversão aumenta o número de *features* de 79 a 331. Finalmente, por meio do atributo `values`, podemos extrair o formato NumPy do formato `pandas` e convertê-lo no tensor representação para treinamento.

```
n_train = train_data.shape[0]
train_features = torch.tensor(all_features[:n_train].values, dtype=torch.float32)
test_features = torch.tensor(all_features[n_train:].values, dtype=torch.float32)
train_labels = torch.tensor(
    train_data.SalePrice.values.reshape(-1, 1), dtype=torch.float32)
```

4.10.5 Treinamento

Para começar, treinamos um modelo linear com perda quadrática. Não surpreendentemente, nosso modelo linear não conduzirá para uma inscrição vencedora de competição mas fornece uma verificação de sanidade para ver se há informações significativas nos dados. Se não podemos fazer melhor do que adivinhação aleatória aqui, então pode haver uma boa chance que temos um bug de processamento de dados. E se as coisas funcionarem, o modelo linear servirá como base dando-nos alguma intuição sobre o quão próximo o modelo simples chega aos melhores modelos relatados, dando-nos uma ideia de quanto ganho devemos esperar de modelos mais sofisticados.

```
loss = nn.MSELoss()
in_features = train_features.shape[1]

def get_net():
    net = nn.Sequential(nn.Linear(in_features,1))
    return net
```

Com os preços das casas, assim como com os preços das ações, nós nos preocupamos com as quantidades relativas mais do que quantidades absolutas. Assim, tendemos a nos preocupar mais com o erro relativo $\frac{y-\hat{y}}{y}$ do que sobre o erro absoluto $y - \hat{y}$. Por exemplo, se nossa previsão estiver errada em US $ 100.000 ao estimar o preço de uma casa na zona rural de Ohio, onde o valor de uma casa típica é 125.000 USD, então provavelmente estamos fazendo um trabalho horrível. Por outro lado, se errarmos neste valor em Los Altos Hills, Califórnia, isso pode representar uma previsão incrivelmente precisa (lá, o preço médio da casa excede 4 milhões de dólares).

Uma maneira de resolver este problema é medir a discrepância no logaritmo das estimativas de

preços. Na verdade, esta é também a medida de erro oficial usado pela competição para avaliar a qualidade dos envios. Afinal, um pequeno valor δ for $|\log y - \log \hat{y}| \leq \delta$ se traduz em $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. Isso leva ao seguinte erro de raiz quadrada média entre o logaritmo do preço previsto e o logaritmo do preço do rótulo:

$$\sqrt{\frac{1}{n}\sum_{i=1}^n \left(\log y_i -\log \hat{y}_i\right)^2}. \tag{4.10.2}$$

```
def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the
    # value less than 1 as 1
    clipped_preds = torch.clamp(net(features), 1, float('inf'))
    rmse = torch.sqrt(loss(torch.log(clipped_preds),
                           torch.log(labels)))
    return rmse.item()
```

Ao contrário das seções anteriores, nossas funções de treinamento contarão com o otimizador Adam (iremos descrevê-lo em maiores detalhes posteriormente). O principal apelo deste otimizador é que, apesar de não melhorar (e às vezes piorar) dados recursos ilimitados para otimização de hiperparâmetros, as pessoas tendem a achar que é significativamente menos sensível à taxa de aprendizagem inicial.

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # The Adam optimization algorithm is used here
    optimizer = torch.optim.Adam(net.parameters(),
                                 lr = learning_rate,
                                 weight_decay = weight_decay)
    for epoch in range(num_epochs):
        for X, y in train_iter:
            optimizer.zero_grad()
            l = loss(net(X), y)
            l.backward()
            optimizer.step()
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

4.10.6 Validação Cruzada K-Fold

Você deve se lembrar que introduzimos validação cruzada K-fold na seção onde discutimos como lidar com seleção de modelo (Section 4.4). Faremos um bom uso disso para selecionar o design do modelo e ajustar os hiperparâmetros. Primeiro precisamos de uma função que retorna a ith dobra dos dados em um procedimento de validação cruzada K-fold. Ela continua cortando o segmento i^th como dados de validação e retornando o resto como dados de treinamento. Observe que esta não é a maneira mais eficiente de lidar com dados e definitivamente faríamos algo muito mais inteligente se nosso conjunto de dados fosse consideravelmente maior. Mas essa complexidade adicional pode ofuscar nosso código desnecessariamente portanto, podemos omiti-lo com segurança devido à simplicidade do nosso problema.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = torch.cat([X_train, X_part], 0)
            y_train = torch.cat([y_train, y_part], 0)
    return X_train, y_train, X_valid, y_valid
```

As médias de erro de treinamento e verificação são retornadas quando treinamos K vezes na validação cruzada de K.

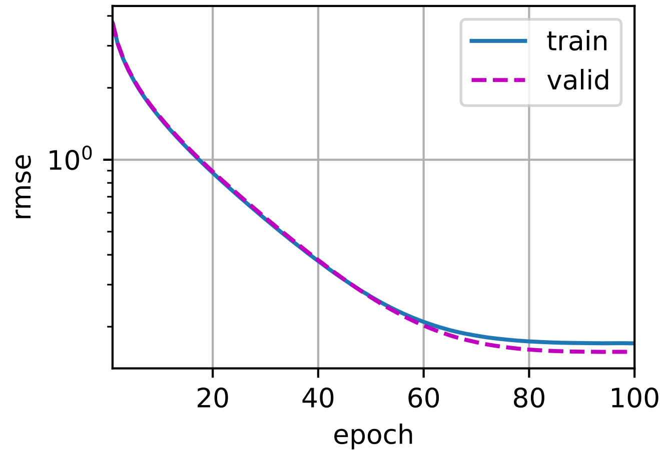
```
def k_fold(k, X_train, y_train, num_epochs, learning_rate, weight_decay,
           batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs + 1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse', xlim=[1, num_epochs],
                     legend=['train', 'valid'], yscale='log')
        print(f'fold {i + 1}, train log rmse {float(train_ls[-1]):f}, '
              f'valid log rmse {float(valid_ls[-1]):f}')
    return train_l_sum / k, valid_l_sum / k
```

4.10.7 Seleção de Modelo

Neste exemplo, escolhemos um conjunto desafinado de hiperparâmetros e deixamos para o leitor melhorar o modelo. Encontrar uma boa escolha pode levar tempo, dependendo de quantas variáveis alguém otimiza. Com um conjunto de dados grande o suficiente, e os tipos normais de hiperparâmetros, Validação cruzada K-fold tende a ser razoavelmente resiliente contra vários testes. No entanto, se tentarmos um número excessivamente grande de opções podemos apenas ter sorte e descobrir que nossa validação o desempenho não é mais representativo do verdadeiro erro.

```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print(f'{k}-fold validation: avg train log rmse: {float(train_l):f}, '
      f'avg valid log rmse: {float(valid_l):f}')
```

```
fold 1, train log rmse 0.170435, valid log rmse 0.157006
fold 2, train log rmse 0.162808, valid log rmse 0.192237
fold 3, train log rmse 0.163597, valid log rmse 0.168006
fold 4, train log rmse 0.168409, valid log rmse 0.154589
fold 5, train log rmse 0.163689, valid log rmse 0.183117
5-fold validation: avg train log rmse: 0.165788, avg valid log rmse: 0.170991
```



Observe que às vezes o número de erros de treinamento para um conjunto de hiperparâmetros pode ser muito baixo, mesmo com o número de erros na validação cruzada de K-fold consideravelmente maior. Isso indica que estamos com *overfitting*. Durante o treinamento, você desejará monitorar os dois números. Menos *overfitting* pode indicar que nossos dados podem suportar um modelo mais poderoso. O *verfitting* maciço pode sugerir que podemos ganhar incorporando técnicas de regularização.

4.10.8 Enviando Previsões no Kaggle

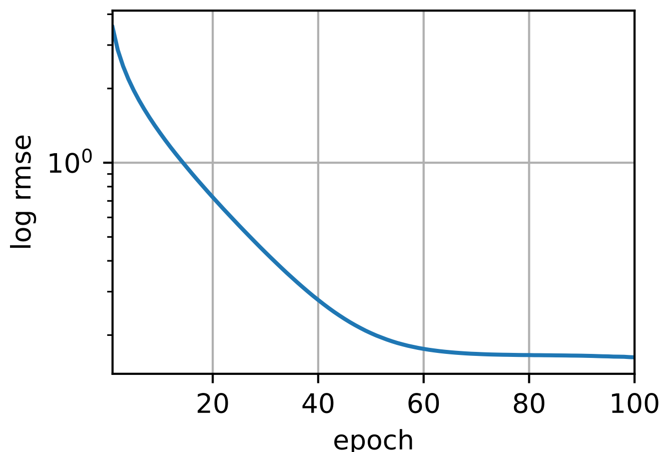
Agora que sabemos qual deve ser uma boa escolha de hiperparâmetros, podemos também usar todos os dados para treinar nele (em vez de apenas $1 - 1/K$ dos dados que são usados nas fatias de validação cruzada). O modelo que obtemos desta forma pode então ser aplicado ao conjunto de teste. Salvar as previsões em um arquivo csv irá simplificar o upload dos resultados para o Kaggle.

```python
def train_and_pred(train_features, test_feature, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='log rmse', xlim=[1, num_epochs], yscale='log')
    print(f'train log rmse {float(train_ls[-1]):f}')
    # Apply the network to the test set
    preds = net(test_features).detach().numpy()
    # Reformat it to export to Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

Uma boa verificação de sanidade é ver se as previsões no conjunto de teste assemelham-se aos do processo de validação cruzada K-fold. Se o fizerem, é hora de enviá-los para o Kaggle. O código a seguir irá gerar um arquivo chamado `submit.csv`.

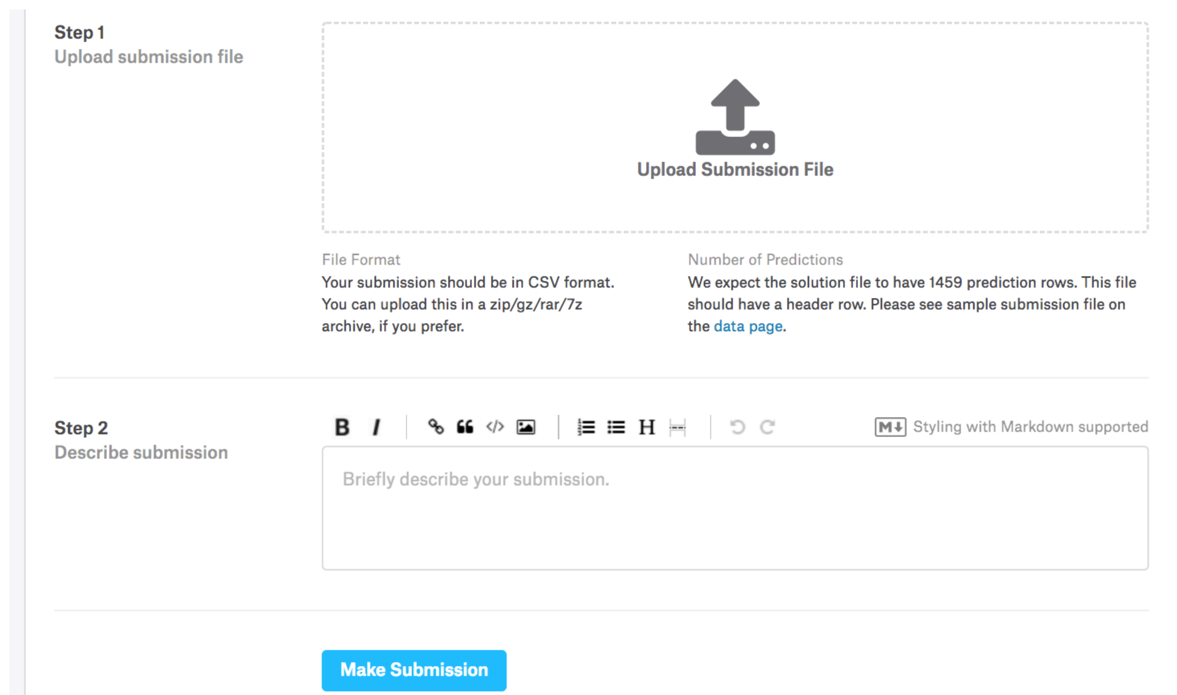
```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train log rmse 0.162520
```



A seguir, conforme demonstrado em:numref:`fig_kaggle_submit2`, podemos enviar nossas previsões no Kaggle e veja como elas se comparam aos preços reais das casas (rótulos) no conjunto de teste. As etapas são bastante simples:

- Faça login no site do Kaggle e visite a página de competição de previsão de preços de casas.
- Clique no botão "Enviar previsões" ou "Envio tardio" (no momento da redação deste botão, o botão está localizado à direita).
- Clique no botão "Enviar arquivo de envio" na caixa tracejada na parte inferior da página e selecione o arquivo de previsão que deseja enviar.
- Clique no botão "Fazer envio" na parte inferior da página para ver seus resultados.
-



width 400px

4.10.9 Resumo

- Os dados reais geralmente contêm uma combinação de diferentes tipos de dados e precisam ser pré-processados.
- Redimensionar dados de valor real para média zero e variância unitária é um bom padrão. O mesmo ocorre com a substituição dos valores ausentes por sua média.
- Transformar características categóricas em características de indicadores nos permite tratá-las como vetores únicos.
- Podemos usar a validação cruzada de K para selecionar o modelo e ajustar os hiperparâmetros.
- Os logaritmos são úteis para erros relativos.

4.10.10 Exercícios

1. Envie suas previsões para esta seção para o Kaggle. Quão boas são suas previsões?
2. Você pode melhorar seu modelo minimizando o logaritmo dos preços diretamente? O que acontece se você tentar prever o logaritmo do preço em vez do preço?
3. É sempre uma boa ideia substituir os valores ausentes por sua média? Dica: você pode construir uma situação em que os valores não faltem aleatoriamente?
4. Melhore a pontuação no Kaggle ajustando os hiperparâmetros por meio da validação cruzada de K.
5. Melhore a pontuação melhorando o modelo (por exemplo, camadas, *weight decay* e eliminação).

6. O que acontece se não padronizarmos as características numéricas contínuas como fizemos nesta seção?

Discussions[58]

[58] https://discuss.d2l.ai/t/107

5 | *Deep Learning* Computacional

Junto com conjuntos de dados gigantes e hardware poderoso, ótimas ferramentas de software desempenharam um papel indispensável no rápido progresso do *Deep Learning*. Começando com a revolucionária biblioteca Theano lançada em 2007, ferramentas de código aberto flexíveis têm permitido aos pesquisadores para prototipar modelos rapidamente, evitando trabalho repetitivo ao reciclar componentes padrão ao mesmo tempo em que mantém a capacidade de fazer modificações de baixo nível. Com o tempo, as bibliotecas de *Deep Learning* evoluíram para oferecer abstrações cada vez mais grosseiras. Assim como os designers de semicondutores passaram a especificar transistores para circuitos lógicos para escrever código, pesquisadores de redes neurais deixaram de pensar sobre o comportamento de neurônios artificiais individuais para conceber redes em termos de camadas inteiras, e agora frequentemente projeta arquiteturas com *blocos* muito mais grosseiros em mente.

Até agora, apresentamos alguns conceitos básicos de aprendizado de máquina, evoluindo para modelos de *Deep Learning* totalmente funcionais. No último capítulo, implementamos cada componente de um MLP do zero e até mostrou como aproveitar APIs de alto nível para lançar os mesmos modelos sem esforço. Para chegar tão longe tão rápido, nós *chamamos* as bibliotecas, mas pulei detalhes mais avançados sobre *como eles funcionam*. Neste capítulo, vamos abrir a cortina, aprofundando os principais componentes da computação de *Deep Learning*, ou seja, construção de modelo, acesso de parâmetro e inicialização, projetando camadas e blocos personalizados, lendo e gravando modelos em disco, e aproveitando GPUs para obter acelerações dramáticas. Esses *insights* o moverão de *usuário final* para *usuário avançado*, dando a você as ferramentas necessárias para colher os benefícios de uma biblioteca de *Deep Learning* madura, mantendo a flexibilidade para implementar modelos mais complexos, incluindo aqueles que você mesmo inventa! Embora este capítulo não introduza nenhum novo modelo ou conjunto de dados, os capítulos de modelagem avançada que se seguem dependem muito dessas técnicas.

5.1 Camadas e Blocos

Quando introduzimos as redes neurais pela primeira vez, focamos em modelos lineares com uma única saída. Aqui, todo o modelo consiste em apenas um único neurônio. Observe que um único neurônio (i) leva algum conjunto de entradas; (ii) gera uma saída escalar correspondente; e (iii) tem um conjunto de parâmetros associados que podem ser atualizados para otimizar alguma função objetivo de interesse. Então, quando começamos a pensar em redes com múltiplas saídas, nós alavancamos a aritmética vetorizada para caracterizar uma camada inteira de neurônios. Assim como os neurônios individuais, camadas (i) recebem um conjunto de entradas, (ii) gerar resultados correspondentes, e (iii) são descritos por um conjunto de parâmetros ajustáveis. Quando trabalhamos com a regressão softmax, uma única camada era ela própria o modelo. No entanto, mesmo quando subsequentemente introduziu MLPs, ainda podemos pensar no modelo como mantendo esta mesma estrutura básica.

Curiosamente, para MLPs, todo o modelo e suas camadas constituintes compartilham essa estrutura. Todo o modelo recebe entradas brutas (os recursos), gera resultados (as previsões), e possui parâmetros (os parâmetros combinados de todas as camadas constituintes). Da mesma forma, cada camada individual ingere entradas (fornecido pela camada anterior) gera saídas (as entradas para a camada subsequente), e possui um conjunto de parâmetros ajustáveis que são atualizados de acordo com o sinal que flui para trás da camada subsequente.

Embora você possa pensar que neurônios, camadas e modelos dê-nos abstrações suficientes para cuidar de nossos negócios, Acontece que muitas vezes achamos conveniente para falar sobre componentes que são maior do que uma camada individual mas menor do que o modelo inteiro. Por exemplo, a arquitetura ResNet-152, que é muito popular na visão computacional, possui centenas de camadas. Essas camadas consistem em padrões repetidos de *grupos de camadas*. Implementar uma camada de rede por vez pode se tornar tedioso. Essa preocupação não é apenas hipotética — tal padrões de projeto são comuns na prática. A arquitetura ResNet mencionada acima venceu as competições de visão computacional ImageNet e COCO 2015 para reconhecimento e detecção (He et al., 2016) e continua sendo uma arquitetura indispensável para muitas tarefas de visão. Arquiteturas semelhantes nas quais as camadas são organizadas em vários padrões repetidos agora são onipresentes em outros domínios, incluindo processamento de linguagem natural e fala.

Para implementar essas redes complexas, introduzimos o conceito de uma rede neural *block*. Um bloco pode descrever uma única camada, um componente que consiste em várias camadas, ou o próprio modelo inteiro! Uma vantagem de trabalhar com a abstração de bloco é que eles podem ser combinados em artefatos maiores, frequentemente recursivamente. Isso é ilustrado em Fig. 5.1.1. Definindo o código para gerar blocos de complexidade arbitrária sob demanda, podemos escrever código surpreendentemente compacto e ainda implementar redes neurais complexas.

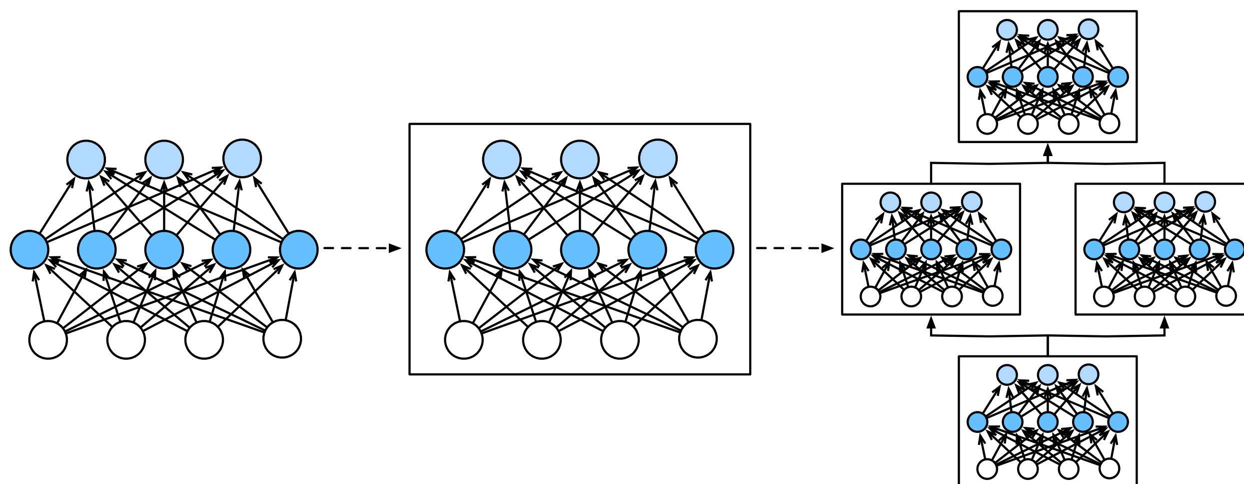


Fig. 5.1.1: Múltiplas camadas são combinadas em blocos, formando padrões repetitivos de um modelo maior.

Do ponto de vista da programação, um bloco é representado por uma *classe*. Qualquer subclasse dele deve definir uma função de propagação direta que transforma sua entrada em saída e deve armazenar todos os parâmetros necessários. Observe que alguns blocos não requerem nenhum parâmetro. Finalmente, um bloco deve possuir uma função de retropropagação, para fins de cálculo de gradientes. Felizmente, devido a alguma magia dos bastidores fornecido pela diferenciação automática (introduzido em Section 2.5) ao definir nosso próprio bloco, só precisamos nos preocupar com os parâmetros e a função de propagação direta.

Para começar, revisitamos o código que usamos para implementar MLPs (Section 4.3). O código a

seguir gera uma rede com uma camada oculta totalmente conectada com 256 unidades e ativação ReLU, seguido por uma camada de saída totalmente conectada com 10 unidades (sem função de ativação).

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X)
```

```
tensor([[-0.0256,  0.1279,  0.0679,  0.0072, -0.2088,  0.0412, -0.2575, -0.0133,
          0.0521,  0.0697],
        [ 0.0747,  0.1671,  0.1247, -0.1827, -0.1590,  0.0773, -0.2884, -0.1681,
          0.0335,  0.2242]], grad_fn=<AddmmBackward>)
```

Neste exemplo, nós construímos nosso modelo instanciando um `nn.Sequential`, com camadas na ordem que eles devem ser executados passados como argumentos. Em suma, `nn.Sequential` define um tipo especial de `Module`, a classe que apresenta um bloco em PyTorch. Ele mantém uma lista ordenada de `Module` constituintes. Observe que cada uma das duas camadas totalmente conectadas é uma instância da classe `Linear` que é uma subclasse de `Module`. A função de propagação direta (`forward`) também é notavelmente simples: ele encadeia cada bloco da lista, passando a saída de cada um como entrada para o próximo. Observe que, até agora, temos invocado nossos modelos através da construção `net (X)` para obter seus resultados. Na verdade, isso é apenas um atalho para `net.__call__(X)`.

5.1.1 Um Bloco Personalizado

Talvez a maneira mais fácil de desenvolver intuição sobre como funciona um bloco é implementar um nós mesmos. Antes de implementar nosso próprio bloco personalizado, resumimos brevemente a funcionalidade básica que cada bloco deve fornecer:

1. Ingerir dados de entrada como argumentos para sua função de propagação direta.
2. Gere uma saída fazendo com que a função de propagação direta retorne um valor. Observe que a saída pode ter uma forma diferente da entrada. Por exemplo, a primeira camada totalmente conectada em nosso modelo acima ingere uma entrada de dimensão arbitrária, mas retorna uma saída de dimensão 256.
3. Calcule o gradiente de sua saída em relação à sua entrada, que pode ser acessado por meio de sua função de retropropagação. Normalmente, isso acontece automaticamente.
4. Armazene e forneça acesso aos parâmetros necessários para executar o cálculo de propagação direta.
5. Inicialize os parâmetros do modelo conforme necessário.

No seguinte trecho de código, nós codificamos um bloco do zero correspondendo a um MLP com uma camada oculta com 256 unidades ocultas, e uma camada de saída de 10 dimensões. Observe que a classe `MLP` abaixo herda a classe que representa um bloco. Vamos contar muito com as funções da classe pai, fornecendo apenas nosso próprio construtor (a função `__init__` em Python) e a função de propagação direta.

```
class MLP(nn.Module):
    # Declare uma camada com parâmetros de modelo. Aqui, declaramos duas
    # camadas totalmente conectadas
    def __init__(self):
        # Chame o construtor da classe pai `MLP` `Block` para realizar
        # a inicialização necessária. Desta forma, outros argumentos de função
        # também podem ser especificado durante a instanciação da classe, como
        # os parametros do modelo, `params` (a ser descritos posteriormente)
        super().__init__()
        self.hidden = nn.Linear(20, 256)  # Hidden layer
        self.out = nn.Linear(256, 10)  # Output layer

    # Defina a propagação direta do modelo, ou seja, como retornar a
    # saída do modelo necessária com base na entrada `X`
    def forward(self, X):
        # Observe aqui que usamos a versão funcional do ReLU definida no
        # módulo nn.functional.
        return self.out(F.relu(self.hidden(X)))
```

Vamos primeiro nos concentrar na função de propagação direta. Observe que leva `X` como entrada, calcula a representação oculta com a função de ativação aplicada, e produz seus *logits*. Nesta implementação `MLP`, ambas as camadas são variáveis de instância. Para ver por que isso é razoável, imagine instanciando dois MLPs, `net1` e`net2`, e treiná-los em dados diferentes. Naturalmente, esperaríamos que eles para representar dois modelos aprendidos diferentes.

Nós instanciamos as camadas do MLP no construtor e posteriormente invocar essas camadas em cada chamada para a função de propagação direta. Observe alguns detalhes importantes: Primeiro, nossa função `__init__` personalizada invoca a função `__init__` da classe pai via `super().__ init __()` poupando-nos da dor de reafirmar o código padrão aplicável à maioria dos blocos. Em seguida, instanciamos nossas duas camadas totalmente conectadas, atribuindo-os a `self.hidden` e`self.out`. Observe que, a menos que implementemos um novo operador, não precisamos nos preocupar com a função de *backpropagation* ou inicialização de parâmetro. O sistema irá gerar essas funções automaticamente. Vamos tentar fazer isso.

```
net = MLP()
net(X)
```

```
tensor([[-0.0627, -0.0958, -0.0792,  0.2375,  0.3284,  0.1038, -0.0707, -0.0726,
         -0.0738,  0.0958],
        [ 0.0125, -0.1072,  0.0780,  0.1053,  0.2692,  0.1731, -0.0750, -0.1358,
         -0.0111, -0.0523]], grad_fn=<AddmmBackward>)
```

Uma virtude fundamental da abstração em bloco é sua versatilidade. Podemos criar uma subclasse de um bloco para criar camadas (como a classe de camada totalmente conectada), modelos inteiros (como a classe `MLP` acima), ou vários componentes de complexidade intermediária. Nós exploramos essa versatilidade ao longo dos capítulos seguintes, como ao abordar redes neurais convolucionais.

5.1.2 O Bloco Sequencial

Agora podemos dar uma olhada mais de perto em como a classe Sequential funciona. Lembre-se de que Sequential foi projetado para conectar outros blocos em série. Para construir nosso próprio MySequential simplificado, só precisamos definir duas funções principais: 1. Uma função para anexar um blocos a uma lista. 2. Uma função de propagação direta para passar uma entrada através da cadeia de blocos, na mesma ordem em que foram acrescentados.

A seguinte classe MySequential oferece o mesmo funcionalidade da classe Sequential padrão.

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            # Here, `module` is an instance of a `Module` subclass. We save it
            #
            # in the member variable `_modules` of the `Module` class, and its
            #
            # type is OrderedDict
            #
            self._modules[str(idx)] = module

    def forward(self, X):
        # OrderedDict guarantees that members will be traversed in the order
        #
        # they were added
        #
        for block in self._modules.values():
            X = block(X)
        return X
```

No método __init__, adicionamos todos os módulos para o dicionário ordenado _modules um por um. Você pode se perguntar por que todo Module possui um atributo _modules e por que o usamos em vez de apenas definir uma lista Python nós mesmos. Em suma, a principal vantagem de _modules é que durante a inicialização do parâmetro do nosso módulo, o sistema sabe olhar dentro do _modules dicionário para encontrar submódulos cujo os parâmetros também precisam ser inicializados.

Quando a função de propagação direta de nosso MySequential é invocada, cada bloco adicionado é executado na ordem em que foram adicionados. Agora podemos reimplementar um MLP usando nossa classe MySequential.

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X)
```

```
tensor([[-0.1604,  0.0811, -0.1572,  0.0730,  0.1101, -0.1630,  0.0652,  0.0542,
          0.3904,  0.3679],
        [-0.2220, -0.0184, -0.0323,  0.1314,  0.2649, -0.0249, -0.1741, -0.0846,
          0.2679,  0.3209]], grad_fn=<AddmmBackward>)
```

Observe que este uso de MySequential é idêntico ao código que escrevemos anteriormente para a classe Sequential (conforme descrito em Section 4.3).

5.1.3 Execução de Código na Função de Propagação Direta

A classe `Sequential` facilita a construção do modelo, nos permitindo montar novas arquiteturas sem ter que definir nossa própria classe. No entanto, nem todas as arquiteturas são cadeias simples. Quando uma maior flexibilidade é necessária, vamos querer definir nossos próprios blocos. Por exemplo, podemos querer executar o controle de fluxo do Python dentro da função de propagação direta. Além disso, podemos querer realizar operações matemáticas arbitrárias, não simplesmente depender de camadas de rede neural predefinidas.

Você deve ter notado que até agora, todas as operações em nossas redes agiram de acordo com as ativações de nossa rede e seus parâmetros. Às vezes, no entanto, podemos querer incorporar termos que não são resultado de camadas anteriores nem parâmetros atualizáveis. Chamamos isso de *parâmetros constantes*. Digamos, por exemplo, que queremos uma camada que calcula a função $f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$, onde $\mathbf{x}$ é a entrada, $\mathbf{w}$ é nosso parâmetro, e c é alguma constante especificada que não é atualizado durante a otimização. Portanto, implementamos uma classe `FixedHiddenMLP` como a seguir.

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Parâmetros de peso aleatórios que não computarão gradientes e
        # portanto, mantem-se constante durante o treinamento
        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)

    def forward(self, X):
        X = self.linear(X)
        # Use os parâmetros constantes criados, bem como as funções `relu` e` mm`
        X = F.relu(torch.mm(X, self.rand_weight) + 1)
        # Reutilize a camada totalmente conectada. Isso é equivalente a compartilhar
        # parâmetros com duas camadas totalmente conectadas
        X = self.linear(X)
        # Controle de fluxo
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

Neste modelo `FixedHiddenMLP`, implementamos uma camada oculta cujos pesos (`self.rand_weight`) são inicializados aleatoriamente na instanciação e daí em diante constantes. Este peso não é um parâmetro do modelo e, portanto, nunca é atualizado por *backpropagation*. A rede então passa a saída desta camada "fixa" através de uma camada totalmente conectada.

Observe que antes de retornar a saída, nosso modelo fez algo incomum. Executamos um *loop while*, testando na condição de que sua norma L_1 seja maior que 1, e dividindo nosso vetor de produção por 2 até que satisfizesse a condição. Finalmente, retornamos a soma das entradas em `X`. Até onde sabemos, nenhuma rede neural padrão executa esta operação. Observe que esta operação em particular pode não ser útil em qualquer tarefa do mundo real. Nosso objetivo é apenas mostrar como integrar código arbitrário no fluxo de seu cálculos de rede neural.

```
net = FixedHiddenMLP()
net(X)
```

```
tensor(0.1221, grad_fn=<SumBackward0>)
```

Podemos misturar e combinar vários maneiras de montar blocos juntos. No exemplo a seguir, aninhamos blocos de algumas maneiras criativas.

```
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(20, 64), nn.ReLU(),
                                 nn.Linear(64, 32), nn.ReLU())
        self.linear = nn.Linear(32, 16)

    def forward(self, X):
        return self.linear(self.net(X))

chimera = nn.Sequential(NestMLP(), nn.Linear(16, 20), FixedHiddenMLP())
chimera(X)
```

```
tensor(-0.0955, grad_fn=<SumBackward0>)
```

5.1.4 Eficiência

O leitor ávido pode começar a se preocupar sobre a eficiência de algumas dessas operações. Afinal, temos muitas pesquisas de dicionário, execução de código e muitas outras coisas Pythônicas ocorrendo no que deveria ser uma biblioteca de *Deep Learning* de alto desempenho. Os problemas do bloqueio do interpretador global[59] do Python são bem conhecidos. No contexto de *Deep Learning*, podemos nos preocupar que nossas GPU(s) extremamente rápidas pode ter que esperar até uma CPU insignificante executa o código Python antes de obter outro trabalho para ser executado.

5.1.5 Sumário

- Camadas são blocos.
- Muitas camadas podem incluir um bloco.
- Muitos blocos podem incluir um bloco.
- Um bloco pode conter código.
- Os blocos cuidam de muitas tarefas domésticas, incluindo inicialização de parâmetros e *backpropagation*.
- As concatenações sequenciais de camadas e blocos são tratadas pelo bloco `Sequencial`.

[59] https://wiki.python.org/moin/GlobalInterpreterLock

5.1.6 Exercícios

1. Que tipos de problemas ocorrerão se você alterar `MySequential` para armazenar blocos em uma lista Python?
2. Implemente um bloco que tenha dois blocos como argumento, digamos `net1` e `net2` e retorne a saída concatenada de ambas as redes na propagação direta. Isso também é chamado de bloco paralelo.
3. Suponha que você deseja concatenar várias instâncias da mesma rede. Implemente uma função de fábrica que gere várias instâncias do mesmo bloco e construa uma rede maior a partir dele.

Discussions[60]

5.2 Gerenciamento de Parâmetros

Depois de escolher uma arquitetura e definir nossos hiperparâmetros, passamos para o ciclo de treinamento, onde nosso objetivo é encontrar valores de parâmetro que minimizam nossa função de perda. Após o treinamento, precisaremos desses parâmetros para fazer previsões futuras. Além disso, às vezes desejamos para extrair os parâmetros seja para reutilizá-los em algum outro contexto, para salvar nosso modelo em disco para que pode ser executado em outro *software*, ou para exame na esperança de ganhar compreensão científica.

Na maioria das vezes, seremos capazes de ignorar os detalhes essenciais de como os parâmetros são declarados e manipulado, contando com estruturas de *Deep Learning* para fazer o trabalho pesado. No entanto, quando nos afastamos de arquiteturas empilhadas com camadas padrão, às vezes precisaremos declarar e manipular parâmetros. Nesta seção, cobrimos o seguinte:

- Parâmetros de acesso para depuração, diagnóstico e visualizações.
- Inicialização de parâmetros.
- Parâmetros de compartilhamento em diferentes componentes do modelo.

Começamos nos concentrando em um MLP com uma camada oculta.

```
import torch
from torch import nn

net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
X = torch.rand(size=(2, 4))
net(X)
```

```
tensor([[0.2046],
        [0.4232]], grad_fn=<AddmmBackward>)
```

[60] https://discuss.d2l.ai/t/55

5.2.1 Acesso a Parâmetros

Vamos começar explicando como acessar os parâmetros dos modelos que você já conhece. Quando um modelo é definido por meio da classe `Sequential`, podemos primeiro acessar qualquer camada indexando no modelo como se fosse uma lista. Os parâmetros de cada camada são convenientemente localizado em seu atributo. Podemos inspecionar os parâmetros da segunda camada totalmente conectada da seguinte maneira.

```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[ 0.2166, -0.3507, -0.1181,  0.2769, -0.1432,  0.2410,  0.
↪0740,  0.2221]])), ('bias', tensor([0.3118]))])
```

A saída nos diz algumas coisas importantes. Primeiro, esta camada totalmente conectada contém dois parâmetros, correspondendo aos pesos e vieses, respectivamente. Ambos são armazenados como *floats* de precisão simples (float32). Observe que os nomes dos parâmetros nos permitem identificar de forma única parâmetros de cada camada, mesmo em uma rede contendo centenas de camadas.

Parâmetros Direcionados

Observe que cada parâmetro é representado como uma instância da classe de parâmetro. Para fazer algo útil com os parâmetros, primeiro precisamos acessar os valores numéricos subjacentes. Existem várias maneiras de fazer isso. Alguns são mais simples, enquanto outros são mais gerais. O código a seguir extrai o viés da segunda camada de rede neural, que retorna uma instância de classe de parâmetro, e acessa posteriormente o valor desse parâmetro.

```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([0.3118], requires_grad=True)
tensor([0.3118])
```

Os parâmetros são objetos complexos, contendo valores, gradientes, e informações adicionais. É por isso que precisamos solicitar o valor explicitamente.

Além do valor, cada parâmetro também nos permite acessar o gradiente. Como ainda não invocamos a *backpropagation* para esta rede, ela está em seu estado inicial.

```
net[2].weight.grad == None
```

```
True
```

Todos os Parâmetros de Uma Vez

Quando precisamos realizar operações em todos os parâmetros, acessá-los um por um pode se tornar tedioso. A situação pode ficar especialmente complicada quando trabalhamos com blocos mais complexos (por exemplo, blocos aninhados), uma vez que precisaríamos voltar recursivamente através de toda a árvore para extrair parâmetros de cada sub-bloco. Abaixo, demonstramos como acessar os parâmetros da primeira camada totalmente conectada versus acessar todas as camadas.

```
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
```

```
('weight', torch.Size([8, 4])) ('bias', torch.Size([8]))
('0.weight', torch.Size([8, 4])) ('0.bias', torch.Size([8])) ('2.weight', torch.Size([1,␣
↪8])) ('2.bias', torch.Size([1]))
```

Isso nos fornece outra maneira de acessar os parâmetros da rede como segue.

```
net.state_dict()['2.bias'].data
```

```
tensor([0.3118])
```

Coletando Parâmetros de Blocos Aninhados

Vamos ver como funcionam as convenções de nomenclatura de parâmetros se aninharmos vários blocos uns dentro dos outros. Para isso, primeiro definimos uma função que produz blocos (uma fábrica de blocos, por assim dizer) e então combine-os dentro de blocos ainda maiores.

```
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                         nn.Linear(8, 4), nn.ReLU())

def block2():
    net = nn.Sequential()
    for i in range(4):
        # Nested here
        net.add_module(f'block {i}', block1())
    return net

rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

```
tensor([[0.1150],
        [0.1150]], grad_fn=<AddmmBackward>)
```

Agora que projetamos a rede, vamos ver como está organizado.

```
print(rgnet)
```

```
Sequential(
  (0): Sequential(
    (block 0): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 1): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 2): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 3): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
  )
  (1): Linear(in_features=4, out_features=1, bias=True)
)
```

Uma vez que as camadas são aninhadas hierarquicamente, também podemos acessá-los como se indexação por meio de listas aninhadas. Por exemplo, podemos acessar o primeiro bloco principal, dentro dele o segundo sub-bloco, e dentro disso o viés da primeira camada, com o seguinte.

```
rgnet[0][1][0].bias.data
```

```
tensor([ 0.3746,  0.1523, -0.2427, -0.0837, -0.3223, -0.3808, -0.1753,  0.4000])
```

5.2.2 Inicialização de Parâmetros

Agora que sabemos como acessar os parâmetros, vamos ver como inicializá-los corretamente. Discutimos a necessidade de inicialização adequada em Section 4.8. A estrutura de *Deep Learning* fornece inicializações aleatórias padrão para suas camadas. No entanto, muitas vezes queremos inicializar nossos pesos de acordo com vários outros protocolos. A estrutura fornece mais comumente protocolos usados e também permite criar um inicializador personalizado.

Por padrão, o PyTorch inicializa matrizes de ponderação e polarização uniformemente extraindo de um intervalo que é calculado de acordo com a dimensão de entrada e saída. O módulo `nn.init` do PyTorch oferece uma variedade de métodos de inicialização predefinidos.

Inicialização *Built-in*

Vamos começar chamando inicializadores integrados. O código abaixo inicializa todos os parâmetros de peso como variáveis aleatórias gaussianas com desvio padrão de 0,01, enquanto os parâmetros de polarização são zerados.

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([ 0.0045, -0.0082, -0.0063,  0.0008]), tensor(0.))
```

Também podemos inicializar todos os parâmetros a um determinado valor constante (digamos, 1).

```
def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)
        nn.init.zeros_(m.bias)
net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

Também podemos aplicar inicializadores diferentes para certos blocos. Por exemplo, abaixo inicializamos a primeira camada com o inicializador Xavier e inicializar a segunda camada para um valor constante de 42.

```
def xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)

net[0].apply(xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([ 0.5433, -0.5930, -0.6287, -0.6148])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

Inicialização Customizada

Às vezes, os métodos de inicialização de que precisamos não são fornecidos pela estrutura de *Deep Learning*. No exemplo abaixo, definimos um inicializador para qualquer parâmetro de peso w usando a seguinte distribuição estranha:

$$w \sim \begin{cases} U(5, 10) & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U(-10, -5) & \text{with probability } \frac{1}{4} \end{cases} \tag{5.2.1}$$

Novamente, implementamos uma função `my_init` para aplicar a `net`.

```
def my_init(m):
    if type(m) == nn.Linear:
        print("Init", *[(name, param.shape)
                        for name, param in m.named_parameters()][0])
        nn.init.uniform_(m.weight, -10, 10)
        m.weight.data *= m.weight.data.abs() >= 5

net.apply(my_init)
net[0].weight[:2]
```

```
Init weight torch.Size([8, 4])
Init weight torch.Size([1, 8])
```

```
tensor([[ 6.2500,  9.0335,  6.4244, -6.6561],
        [-5.9088,  9.8715,  7.8404,  0.0000]], grad_fn=<SliceBackward>)
```

Observe que sempre temos a opção de definir parâmetros diretamente.

```
net[0].weight.data[:] += 1
net[0].weight.data[0, 0] = 42
net[0].weight.data[0]
```

```
tensor([42.0000, 10.0335,  7.4244, -5.6561])
```

5.2.3 Parâmetros *Tied*

Frequentemente, queremos compartilhar parâmetros em várias camadas. Vamos ver como fazer isso com elegância. A seguir, alocamos uma camada densa e usar seus parâmetros especificamente para definir os de outra camada.

```
# Precisamos dar as camadas compartilhadas um nome
# para que possamos referenciar seus parâmetros

#
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
```

(continues on next page)

(continued from previous page)

```
                    nn.Linear(8, 1))
net(X)
# Checar se são os mesmos parâmetros
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Garantindo que são o mesmo objeto ao invés de ter
# apenas o mesmo valor
print(net[2].weight.data[0] == net[4].weight.data[0])
```

```
tensor([True, True, True, True, True, True, True, True])
tensor([True, True, True, True, True, True, True, True])
```

Este exemplo mostra que os parâmetros da segunda e terceira camadas são amarrados. Eles não são apenas iguais, eles são representado pelo mesmo tensor exato. Assim, se mudarmos um dos parâmetros, o outro também muda. Você pode se perguntar, quando os parâmetros são amarrados o que acontece com os gradientes? Uma vez que os parâmetros do modelo contêm gradientes, os gradientes da segunda camada oculta e a terceira camada oculta são adicionadas juntas durante a retropropagação.

5.2.4 Sumário

- Temos várias maneiras de acessar, inicializar e vincular os parâmetros do modelo.
- Podemos usar inicialização personalizada.

5.2.5 Exercícios

1. Use o modelo `FancyMLP` definido em Section 5.1 e acesse os parâmetros das várias camadas.
2. Observe o documento do módulo de inicialização para explorar diferentes inicializadores.
3. Construa um MLP contendo uma camada de parâmetros compartilhados e treine-o. Durante o processo de treinamento, observe os parâmetros do modelo e gradientes de cada camada.
4. Por que compartilhar parâmetros é uma boa ideia?

Discussão[61]

5.3 Camadas Personalizadas

Um fator por trás do sucesso do *Deep Learning* é a disponibilidade de uma ampla gama de camadas que pode ser composto de maneiras criativas projetar arquiteturas adequadas para uma ampla variedade de tarefas. Por exemplo, os pesquisadores inventaram camadas especificamente para lidar com imagens, texto, loop sobre dados sequenciais, e realizando programação dinâmica. Mais cedo ou mais tarde, você encontrará ou inventará uma camada que ainda não existe na estrutura de *Deep Learning*. Nesses casos, você deve construir uma camada personalizada. Nesta seção, mostramos como.

[61] https://discuss.d2l.ai/t/57

5.3.1 Camadas Sem Parâmetros

Para começar, construímos uma camada personalizada que não possui parâmetros próprios. Isso deve parecer familiar, se você se lembra de nosso introdução ao bloco em Section 5.1. A seguinte classe `CenteredLayer` simplesmente subtrai a média de sua entrada. Para construí-lo, simplesmente precisamos herdar da classe da camada base e implementar a função de propagação direta.

```
import torch
from torch import nn
from torch.nn import functional as F


class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

Vamos verificar se nossa camada funciona conforme o esperado, alimentando alguns dados por meio dela.

```
layer = CenteredLayer()
layer(torch.FloatTensor([1, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

Agora podemos incorporar nossa camada como um componente na construção de modelos mais complexos.

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

Como uma verificação extra de sanidade, podemos enviar dados aleatórios através da rede e verificar se a média é de fato 0. Porque estamos lidando com números de ponto flutuante, ainda podemos ver um número muito pequeno diferente de zero devido à quantização.

```
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(-4.1910e-09, grad_fn=<MeanBackward0>)
```

5.3.2 Camadas com Parâmetros

Agora que sabemos como definir camadas simples, vamos prosseguir para a definição de camadas com parâmetros que pode ser ajustado por meio de treinamento. Podemos usar funções integradas para criar parâmetros, que fornecem algumas funcionalidades básicas de manutenção. Em particular, eles governam o acesso, inicialização, compartilhar, salvar e carregar parâmetros do modelo. Dessa forma, entre outros benefícios, não precisaremos escrever rotinas de serialização personalizadas para cada camada personalizada.

Agora, vamos implementar nossa própria versão da camada totalmente conectada. Lembre-se de que esta camada requer dois parâmetros, um para representar o peso e outro para o viés. Nesta implementação, preparamos a ativação do ReLU como padrão. Esta camada requer a entrada de argumentos: `in_units` e `units`, que denotam o número de entradas e saídas, respectivamente.

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))
    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

Em seguida, instanciamos a classe `MyDense` e acessar seus parâmetros de modelo.

```
dense = MyLinear(5, 3)
dense.weight
```

```
Parameter containing:
tensor([[-0.4359, -0.8619, -0.4170],
        [-0.3191,  0.3732,  0.7508],
        [-0.1494, -0.8253, -0.7071],
        [ 1.0940, -0.4332, -2.2712],
        [ 1.9598, -0.2122,  1.8529]], requires_grad=True)
```

Podemos realizar cálculos de propagação direta usando camadas personalizadas.

```
dense(torch.rand(2, 5))
```

```
tensor([[1.1498, 0.0000, 0.0000],
        [1.9590, 0.0000, 0.0000]])
```

Também podemos construir modelos usando camadas personalizadas. Assim que tivermos isso, podemos usá-lo como a camada totalmente conectada integrada.

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

```
tensor([[0.],
        [0.]])
```

5.3.3 Sumário

- Podemos projetar camadas personalizadas por meio da classe de camada básica. Isso nos permite definir novas camadas flexíveis que se comportam de maneira diferente de quaisquer camadas existentes na biblioteca.
- Uma vez definidas, as camadas personalizadas podem ser chamadas em contextos e arquiteturas arbitrários.
- As camadas podem ter parâmetros locais, que podem ser criados por meio de funções integradas.

5.3.4 Exercícios

3. Projete uma camada que recebe uma entrada e calcula uma redução de tensor, ou seja, ele retorna $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
4. Projete uma camada que retorne a metade anterior dos coeficientes de Fourier dos dados.

Discussão[62]

5.4 Entrada e Saída de Arquivos

Até agora, discutimos como processar dados e como para construir, treinar e testar modelos de *Deep Learning*. No entanto, em algum momento, esperamos ser felizes o suficiente com os modelos aprendidos que queremos para salvar os resultados para uso posterior em vários contextos (talvez até mesmo para fazer previsões na implantação). Além disso, ao executar um longo processo de treinamento, a prática recomendada é salvar resultados intermediários periodicamente (pontos de verificação) para garantir que não perdemos vários dias de computação se tropeçarmos no cabo de alimentação do nosso servidor. Portanto, é hora de aprender como carregar e armazenar ambos os vetores de peso individuais e modelos inteiros. Esta seção aborda ambos os problemas.

5.4.1 Carregando e Salvando Tensores

Para tensores individuais, podemos diretamente invocar as funções `load` e `save` para ler e escrever respectivamente. Ambas as funções exigem que forneçamos um nome, e `save` requer como entrada a variável a ser salva.

```
import torch
from torch import nn
from torch.nn import functional as F

x = torch.arange(4)
torch.save(x, 'x-file')
```

Agora podemos ler os dados do arquivo armazenado de volta na memória.

[62] https://discuss.d2l.ai/t/59

```
x2 = torch.load("x-file")
x2
```

```
tensor([0, 1, 2, 3])
```

Podemos armazenar uma lista de tensores e lê-los de volta na memória.

```
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

```
(tensor([0, 1, 2, 3]), tensor([0., 0., 0., 0.]))
```

Podemos até escrever e ler um dicionário que mapeia de *strings* a tensores. Isso é conveniente quando queremos ler ou escrever todos os pesos em um modelo.

```
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

5.4.2 Carregando e Salvando Parâmetros de Modelos

Salvar vetores de peso individuais (ou outros tensores) é útil, mas fica muito tedioso se quisermos salvar (e depois carregar) um modelo inteiro. Afinal, podemos ter centenas de grupos de parâmetros espalhados por toda parte. Por esta razão, a estrutura de *Deep Learning* fornece funcionalidades integradas para carregar e salvar redes inteiras. Um detalhe importante a notar é que este salva o modelo *parâmetros* e não o modelo inteiro. Por exemplo, se tivermos um MLP de 3 camadas, precisamos especificar a arquitetura separadamente. A razão para isso é que os próprios modelos podem conter código arbitrário, portanto, eles não podem ser serializados naturalmente. Assim, para restabelecer um modelo, precisamos para gerar a arquitetura em código e carregue os parâmetros do disco. Vamos começar com nosso MLP familiar.

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
```

A seguir, armazenamos os parâmetros do modelo como um arquivo com o nome "mlp.params".

```
torch.save(net.state_dict(), 'mlp.params')
```

Para recuperar o modelo, instanciamos um clone do modelo MLP original. Em vez de inicializar aleatoriamente os parâmetros do modelo, lemos os parâmetros armazenados no arquivo diretamente.

```
clone = MLP()
clone.load_state_dict(torch.load("mlp.params"))
clone.eval()
```

```
MLP(
  (hidden): Linear(in_features=20, out_features=256, bias=True)
  (output): Linear(in_features=256, out_features=10, bias=True)
)
```

Uma vez que ambas as instâncias têm os mesmos parâmetros de modelo, o resultado computacional da mesma entrada `X` deve ser o mesmo. Deixe-nos verificar isso.

```
Y_clone = clone(X)
Y_clone == Y
```

```
tensor([[True, True, True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True, True, True]])
```

5.4.3 Sumário

- As funções `save` e `load` podem ser usadas para executar E/S de arquivo para objetos tensores.
- Podemos salvar e carregar todos os conjuntos de parâmetros de uma rede por meio de um dicionário de parâmetros.
- Salvar a arquitetura deve ser feito em código e não em parâmetros.

5.4.4 Exercícios

1. Mesmo se não houver necessidade de implantar modelos treinados em um dispositivo diferente, quais são os benefícios práticos de armazenar parâmetros de modelo?
2. Suponha que desejamos reutilizar apenas partes de uma rede para serem incorporadas a uma rede de arquitetura diferente. Como você usaria, digamos, as duas primeiras camadas de uma rede anterior em uma nova rede?
3. Como você salvaria a arquitetura e os parâmetros da rede? Que restrições você imporia à arquitetura?

Discussão[63]

[63] https://discuss.d2l.ai/t/61

5.5 GPUs

Em tab_intro_decade, discutimos o rápido crescimento de computação nas últimas duas décadas. Em suma, o desempenho da GPU aumentou por um fator de 1000 a cada década desde 2000. Isso oferece ótimas oportunidades, mas também sugere uma necessidade significativa de fornecer tal desempenho.

Nesta seção, começamos a discutir como aproveitar este desempenho computacional para sua pesquisa. Primeiro usando GPUs únicas e, posteriormente, como usar várias GPUs e vários servidores (com várias GPUs).

Especificamente, discutiremos como para usar uma única GPU NVIDIA para cálculos. Primeiro, certifique-se de ter pelo menos uma GPU NVIDIA instalada. Em seguida, baixe o NVIDIA driver e CUDA[64]. e siga as instruções para definir o caminho apropriado. Assim que esses preparativos forem concluídos, o comando nvidia-smi pode ser usado para ver as informações da placa gráfica.

```
!nvidia-smi
```

```
Sat Dec 11 06:56:32 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   43C    P0    37W / 300W |     11MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   58C    P0    45W / 300W |     11MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   44C    P0    41W / 300W |     11MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   61C    P0    62W / 300W |     11MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

No PyTorch, cada array possui um dispositivo, frequentemente o referimos como um contexto. Até agora, por padrão, todas as variáveis e computação associada foram atribuídos à CPU. Normalmente, outros contextos podem ser várias GPUs. As coisas podem ficar ainda mais complicadas quando nós implantamos trabalhos em vários servidores. Ao atribuir matrizes a contextos de forma inteligente, podemos minimizar o tempo gasto transferência de dados entre dispositivos. Por exemplo, ao treinar redes neurais em um servidor com uma GPU, normalmente preferimos que os parâmetros do modelo residam na GPU.

[64] https://developer.nvidia.com/cuda-downloads

Em seguida, precisamos confirmar que a versão GPU do PyTorch está instalada. Se uma versão CPU do PyTorch já estiver instalada, precisamos desinstalá-lo primeiro. Por exemplo, use o comando `pip uninstall torch`, em seguida, instale a versão correspondente do PyTorch de acordo com sua versão CUDA. Supondo que você tenha o CUDA 10.0 instalado, você pode instalar a versão PyTorch compatível com CUDA 10.0 via `pip install torch-cu100`.

Para executar os programas desta seção, você precisa de pelo menos duas GPUs. Observe que isso pode ser extravagante para a maioria dos computadores desktop mas está facilmente disponível na nuvem, por exemplo, usando as instâncias multi-GPU do AWS EC2. Quase todas as outras seções * não * requerem várias GPUs. Em vez disso, isso é simplesmente para ilustrar como os dados fluem entre diferentes dispositivos.

5.5.1 Dispositivos Computacionais

Podemos especificar dispositivos, como CPUs e GPUs, para armazenamento e cálculo. Por padrão, os tensores são criados na memória principal e, em seguida, use a CPU para calculá-lo.

No PyTorch, a CPU e a GPU podem ser indicadas por `torch.device('cpu')` e `torch.cuda.device('cuda')`. Deve-se notar que o dispositivo cpu significa todas as CPUs físicas e memória. Isso significa que os cálculos de PyTorch tentará usar todos os núcleos da CPU. No entanto, um dispositivo gpu representa apenas uma placa e a memória correspondente. Se houver várias GPUs, usamos `torch.cuda.device(f'cuda: {i}')` para representar a ith GPU (i começa em 0). Além disso, gpu:0 e gpu são equivalentes.

```
import torch
from torch import nn

torch.device('cpu'), torch.cuda.device('cuda'), torch.cuda.device('cuda:1')
```

```
(device(type='cpu'),
 <torch.cuda.device at 0x7f5ec461f9d0>,
 <torch.cuda.device at 0x7f5ec4645ca0>)
```

Podemos consultar o número de GPUs disponíveis.

```
torch.cuda.device_count()
```

```
2
```

Agora definimos duas funções convenientes que nos permitem para executar o código mesmo que as GPUs solicitadas não existam.

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')

def try_all_gpus():  #@save
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    devices = [torch.device(f'cuda:{i}')
```

(continues on next page)

(continued from previous page)

```
            for i in range(torch.cuda.device_count())]
    return devices if devices else [torch.device('cpu')]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

5.5.2 Tensores e GPUs

Por padrão, tensores são criados na CPU. Podemos consultar o dispositivo onde o tensor está localizado.

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

É importante notar que sempre que quisermos para operar em vários termos, eles precisam estar no mesmo dispositivo. Por exemplo, se somarmos dois tensores, precisamos ter certeza de que ambos os argumentos estão no mesmo dispositivo — caso contrário, a estrutura não saberia onde armazenar o resultado ou mesmo como decidir onde realizar o cálculo.

Armazenamento na GPU

Existem várias maneiras de armazenar um tensor na GPU. Por exemplo, podemos especificar um dispositivo de armazenamento ao criar um tensor. A seguir, criamos a variável tensorial `X` no primeiro gpu. O tensor criado em uma GPU consome apenas a memória desta GPU. Podemos usar o comando `nvidia-smi` para ver o uso de memória da GPU. Em geral, precisamos ter certeza de não criar dados que excedam o limite de memória da GPU.

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

Supondo que você tenha pelo menos duas GPUs, o código a seguir criará um tensor aleatório na segunda GPU.

```
Y = torch.rand(2, 3, device=try_gpu(1))
Y
```

```
tensor([[0.0183, 0.8929, 0.2991],
        [0.5415, 0.1923, 0.6762]], device='cuda:1')
```

Copiando

Se quisermos calcular `X + Y`, precisamos decidir onde realizar esta operação. Por exemplo, como mostrado em Fig. 5.5.1, podemos transferir `X` para a segunda GPU e realizar a operação lá. *Não* simplesmente adicione `X` e`Y`, pois isso resultará em uma exceção. O mecanismo de tempo de execução não saberia o que fazer: ele não consegue encontrar dados no mesmo dispositivo e falha. Já que `Y` vive na segunda GPU, precisamos mover `X` para lá antes de podermos adicionar os dois.

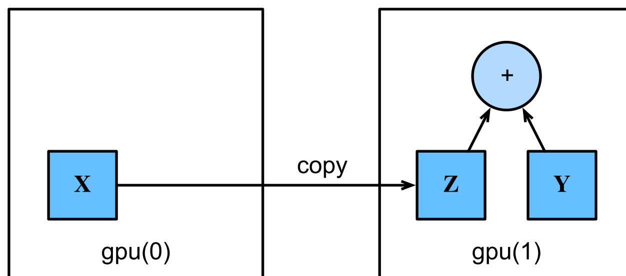


Fig. 5.5.1: Copiar dados para realizar uma operação no mesmo dispositivo.

```
Z = X.cuda(1)
print(X)
print(Z)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:1')
```

Agora que os dados estão na mesma GPU (ambos são `Z` e`Y`), podemos somá-los.

```
Y + Z
```

```
tensor([[1.0183, 1.8929, 1.2991],
        [1.5415, 1.1923, 1.6762]], device='cuda:1')
```

Imagine que sua variável `Z` já esteja em sua segunda GPU. O que acontece se ainda chamarmos `Z.cuda(1)`? Ele retornará `Z` em vez de fazer uma cópia e alocar nova memória.

```
Z.cuda(1) is Z
```

```
True
```

Informações extra

As pessoas usam GPUs para fazer aprendizado de máquina porque eles esperam que ela seja rápida. Mas a transferência de variáveis entre dispositivos é lenta. Então, queremos que você tenha 100% de certeza que você deseja fazer algo lento antes de deixá-lo fazer. Se a estrutura de *Deep Learning* apenas fizesse a cópia automaticamente sem bater, então você pode não perceber que você escreveu algum código lento.

Além disso, a transferência de dados entre dispositivos (CPU, GPUs e outras máquinas) é algo muito mais lento do que a computação. Também torna a paralelização muito mais difícil, já que temos que esperar que os dados sejam enviados (ou melhor, para serem recebidos) antes de prosseguirmos com mais operações. É por isso que as operações de cópia devem ser realizadas com muito cuidado. Como regra geral, muitas pequenas operações são muito piores do que uma grande operação. Além disso, várias operações ao mesmo tempo são muito melhores do que muitas operações simples intercaladas no código a menos que você saiba o que está fazendo. Este é o caso, uma vez que tais operações podem bloquear se um dispositivo tem que esperar pelo outro antes de fazer outra coisa. É um pouco como pedir seu café em uma fila em vez de pré-encomendá-lo por telefone e descobrir que ele está pronto quando você estiver.

Por último, quando imprimimos tensores ou convertemos tensores para o formato NumPy, se os dados não estiverem na memória principal, o framework irá copiá-lo para a memória principal primeiro, resultando em sobrecarga de transmissão adicional. Pior ainda, agora está sujeito ao temido bloqueio de intérprete global isso faz tudo esperar que o Python seja concluído.

5.5.3 Redes Neurais e GPUs

Da mesma forma, um modelo de rede neural pode especificar dispositivos. O código a seguir coloca os parâmetros do modelo na GPU.

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
```

Veremos muitos mais exemplos de como executar modelos em GPUs nos capítulos seguintes, simplesmente porque eles se tornarão um pouco mais intensivos em termos de computação.

Quando a entrada é um tensor na GPU, o modelo calculará o resultado na mesma GPU.

```
net(X)
```

```
tensor([[1.5026],
        [1.5026]], device='cuda:0', grad_fn=<AddmmBackward>)
```

Vamos confirmar se os parâmetros do modelo estão armazenados na mesma GPU.

```
net[0].weight.data.device
```

```
device(type='cuda', index=0)
```

Resumindo, contanto que todos os dados e parâmetros estejam no mesmo dispositivo, podemos aprender modelos com eficiência. Nos próximos capítulos, veremos vários desses exemplos.

5.5.4 Sumário

- Podemos especificar dispositivos para armazenamento e cálculo, como CPU ou GPU. Por padrão, os dados são criados na memória principal e então usa-se a CPU para cálculos.
- A estrutura de *Deep Learning* requer todos os dados de entrada para cálculo estar no mesmo dispositivo, seja CPU ou a mesma GPU.
- Você pode perder um desempenho significativo movendo dados sem cuidado. Um erro típico é o seguinte: calcular a perda para cada minibatch na GPU e relatando de volta para o usuário na linha de comando (ou registrando-o em um NumPy `ndarray`) irá disparar um bloqueio global do interpretador que paralisa todas as GPUs. É muito melhor alocar memória para registrar dentro da GPU e apenas mover registros maiores.

5.5.5 Exercícios

1. Tente uma tarefa de computação maior, como a multiplicação de grandes matrizes, e veja a diferença de velocidade entre a CPU e a GPU. Que tal uma tarefa com uma pequena quantidade de cálculos?
2. Como devemos ler e escrever os parâmetros do modelo na GPU?
3. Meça o tempo que leva para calcular 1000 multiplicações matriz-matriz de 100×100 matrizes e registrar a norma de Frobenius da matriz de saída, um resultado de cada vez vs. manter um registro na GPU e transferir apenas o resultado final.
4. Meça quanto tempo leva para realizar duas multiplicações matriz-matriz em duas GPUs ao mesmo tempo vs. em sequência em uma GPU. Dica: você deve ver uma escala quase linear.

Discussão[65]

[65] https://discuss.d2l.ai/t/63

6 | Convolutional Neural Networks

In earlier chapters, we came up against image data, for which each example consists of a two-dimensional grid of pixels. Depending on whether we are handling black-and-white or color images, each pixel location might be associated with either one or multiple numerical values, respectively. Until now, our way of dealing with this rich structure was deeply unsatisfying. We simply discarded each image's spatial structure by flattening them into one-dimensional vectors, feeding them through a fully-connected MLP. Because these networks are invariant to the order of the features, we could get similar results regardless of whether we preserve an order corresponding to the spatial structure of the pixels or if we permute the columns of our design matrix before fitting the MLP's parameters. Preferably, we would leverage our prior knowledge that nearby pixels are typically related to each other, to build efficient models for learning from image data.

This chapter introduces *convolutional neural networks* (CNNs), a powerful family of neural networks that are designed for precisely this purpose. CNN-based architectures are now ubiquitous in the field of computer vision, and have become so dominant that hardly anyone today would develop a commercial application or enter a competition related to image recognition, object detection, or semantic segmentation, without building off of this approach.

Modern CNNs, as they are called colloquially owe their design to inspirations from biology, group theory, and a healthy dose of experimental tinkering. In addition to their sample efficiency in achieving accurate models, CNNs tend to be computationally efficient, both because they require fewer parameters than fully-connected architectures and because convolutions are easy to parallelize across GPU cores. Consequently, practitioners often apply CNNs whenever possible, and increasingly they have emerged as credible competitors even on tasks with a one-dimensional sequence structure, such as audio, text, and time series analysis, where recurrent neural networks are conventionally used. Some clever adaptations of CNNs have also brought them to bear on graph-structured data and in recommender systems.

First, we will walk through the basic operations that comprise the backbone of all convolutional networks. These include the convolutional layers themselves, nitty-gritty details including padding and stride, the pooling layers used to aggregate information across adjacent spatial regions, the use of multiple channels at each layer, and a careful discussion of the structure of modern architectures. We will conclude the chapter with a full working example of LeNet, the first convolutional network successfully deployed, long before the rise of modern deep learning. In the next chapter, we will dive into full implementations of some popular and comparatively recent CNN architectures whose designs represent most of the techniques commonly used by modern practitioners.

6.1 De Camadas Totalmente Conectadas às Convoluções

Até hoje, os modelos que discutimos até agora permanecem opções apropriadas quando estamos lidando com dados tabulares. Por tabular, queremos dizer que os dados consistem de linhas correspondentes a exemplos e colunas correspondentes a *features*. Com dados tabulares, podemos antecipar que os padrões que buscamos podem envolver interações entre as características, mas não assumimos nenhuma estrutura *a priori* sobre como as características interagem.

Às vezes, realmente não temos conhecimento para orientar a construção de arquiteturas mais artesanais. Nestes casos, um MLP pode ser o melhor que podemos fazer. No entanto, para dados perceptivos de alta dimensão, essas redes sem estrutura podem se tornar difíceis de manejar.

Por exemplo, vamos voltar ao nosso exemplo de execução de distinguir gatos de cães. Digamos que fazemos um trabalho completo na coleta de dados, coletando um conjunto de dados anotado de fotografias de um megapixel. Isso significa que cada entrada na rede tem um milhão de dimensões. De acordo com nossas discussões sobre custo de parametrização de camadas totalmente conectadas em Section 3.4.3, até mesmo uma redução agressiva para mil dimensões ocultas exigiria uma camada totalmente conectada caracterizada por $10^6 \times 10^3 = 10^9$ parâmetros. A menos que tenhamos muitas GPUs, um talento para otimização distribuída, e uma quantidade extraordinária de paciência, aprender os parâmetros desta rede pode acabar sendo inviável.

Um leitor cuidadoso pode objetar a este argumento na base de que a resolução de um megapixel pode não ser necessária. No entanto, embora possamos ser capazes de escapar com apenas cem mil pixels, nossa camada oculta de tamanho 1000 subestima grosseiramente o número de unidades ocultas que leva para aprender boas representações de imagens, portanto, um sistema prático ainda exigirá bilhões de parâmetros. Além disso, aprender um classificador ajustando tantos parâmetros pode exigir a coleta de um enorme conjunto de dados. E ainda hoje tanto os humanos quanto os computadores são capazes de distinguir gatos de cães muito bem, aparentemente contradizendo essas intuições. Isso ocorre porque as imagens exibem uma estrutura rica que pode ser explorada por humanos e modelos de aprendizado de máquina semelhantes. Redes neurais convolucionais (CNNs) são uma forma criativa que o *machine learning* adotou para explorar algumas das estruturas conhecidas em imagens naturais.

6.1.1 Invariância

Imagine que você deseja detectar um objeto em uma imagem. Parece razoável que qualquer método que usamos para reconhecer objetos não deveria se preocupar demais com a localização precisa do objeto na imagem. Idealmente, nosso sistema deve explorar esse conhecimento. Os porcos geralmente não voam e os aviões geralmente não nadam. No entanto, devemos ainda reconhecer um porco era aquele que aparecia no topo da imagem. Podemos tirar alguma inspiração aqui do jogo infantil "Cadê o Wally" (representado em Fig. 6.1.1). O jogo consiste em várias cenas caóticas repletas de atividades. Wally aparece em algum lugar em cada uma, normalmente à espreita em algum local improvável. O objetivo do leitor é localizá-lo. Apesar de sua roupa característica, isso pode ser surpreendentemente difícil, devido ao grande número de distrações. No entanto, *a aparência do Wally* não depende de *onde o Wally está localizado*. Poderíamos varrer a imagem com um detector Wally que poderia atribuir uma pontuação a cada *patch*, indicando a probabilidade de o *patch* conter Wally. CNNs sistematizam essa ideia de *invariância espacial*, explorando para aprender representações úteis com menos parâmetros.

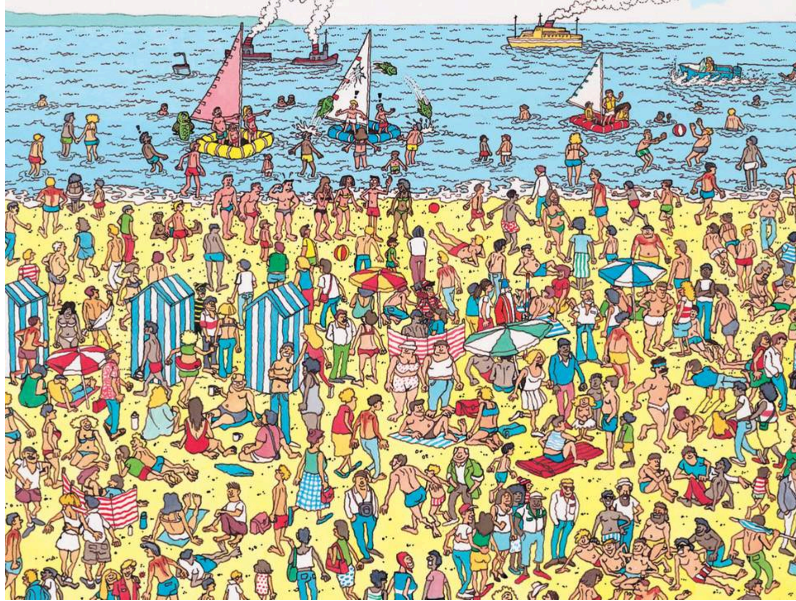


Fig. 6.1.1: Uma imagem do jogo "Onde está Wally".

Agora podemos tornar essas intuições mais concretas enumerando alguns desiderATOS para orientar nosso design de uma arquitetura de rede neural adequada para visão computacional:

1. Nas primeiras camadas, nossa rede deve responder de forma semelhante ao mesmo *patch*, independentemente de onde aparece na imagem. Este princípio é denominado *invariância da tradução*.
2. As primeiras camadas da rede devem se concentrar nas regiões locais, sem levar em conta o conteúdo da imagem em regiões distantes. Este é o princípio de *localidade*. Eventualmente, essas representações locais podem ser agregadas para fazer previsões em todo o nível da imagem.

Vamos ver como isso se traduz em matemática.

6.1.2 Restringindo o MLP

Para começar, podemos considerar um MLP com imagens bidimensionais $\mathbf{X}$ como entradas e suas representações ocultas imediatas $\mathbf{H}$ similarmente representadas como matrizes em matemática e como tensores bidimensionais em código, onde $\mathbf{X}$ e $\mathbf{H}$ têm a mesma forma. Deixe isso penetrar. Agora concebemos não apenas as entradas, mas também as representações ocultas como possuidoras de estrutura espacial.

Deixe $[\mathbf{X}]_{i,j}$ e $[\mathbf{H}]_{i,j}$ denotarem o pixel no local (i, j) na imagem de entrada e representação oculta, respectivamente. Consequentemente, para que cada uma das unidades ocultas receba entrada de cada um dos pixels de entrada, nós deixaríamos de usar matrizes de peso (como fizemos anteriormente em MLPs) para representar nossos parâmetros como tensores de peso de quarta ordem W. Suponha que $\mathbf{U}$ contenha *bias*, poderíamos expressar formalmente a camada totalmente conectada como

$$\begin{aligned} \left[\mathbf{H}\right]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l}\\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned}, \tag{6.1.1}$$

onde a mudança de W para V é inteiramente cosmética por enquanto uma vez que existe uma correspondência um-para-um entre coeficientes em ambos os tensores de quarta ordem. Nós simplesmente reindexamos os subscritos (k, l) de modo que $k = i + a$ and $l = j + b$. Em outras palavras, definimos $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$. Os índices a e b ultrapassam os deslocamentos positivos e negativos, cobrindo toda a imagem. Para qualquer localização dada (i, j) na representação oculta $[\mathbf{H}]_{i,j}$, calculamos seu valor somando os pixels em x, centralizado em torno de (i, j) e ponderado por $[\mathsf{V}]_{i,j,a,b}$.

Invariância de Tradução

Agora vamos invocar o primeiro princípio estabelecido acima: invariância de tradução. Isso implica que uma mudança na entrada $\mathbf{X}$ deve simplesmente levar a uma mudança na representação oculta $\mathbf{H}$. Isso só é possível se V e $\mathbf{U}$ não dependem realmente de (i, j), ou seja, temos $[\mathsf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ e **U**$ é uma constante, digamos u. Como resultado, podemos simplificar a definição de $\mathbf{H}$:

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.1.2}$$

Esta é uma *convolução*! Estamos efetivamente ponderando pixels em $(i+a, j+b)$ nas proximidades da localização (i, j) com coeficientes$[\mathbf{V}]_{a,b}$ para obter o valor $[\mathbf{H}]_{i,j}$. Observe que $[\mathbf{V}]_{a,b}$ precisa de muito menos coeficientes do que $[:raw-latex:`\mathsf{V}`]_{i, j, a, b}, pois ele não depende mais da localização na imagem. Fizemos um progresso significativo!

Localidade

Agora, vamos invocar o segundo princípio: localidade. Conforme motivado acima, acreditamos que não devemos ter parecer muito longe do local (i, j) a fim de coletar informações relevantes para avaliar o que está acontecendo em $[\mathbf{H}]_{i,j}$. Isso significa que fora de algum intervalo $|a| > \Delta$ or $|b| > \Delta$, devemos definir $[\mathbf{V}]_{a,b} = 0$. Equivalentemente, podemos reescrever $[\mathbf{H}]_{i,j}$ como

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.1.3}$$

Observe que (6.1.3), em poucas palavras, é uma *camada convolucional. Redes neurais convolucionais* (CNNs[1]) são uma família especial de redes neurais que contêm camadas convolucionais. Na comunidade de pesquisa de *deep learning*, $\mathbf{V}$ é referido como um *kernel de convolução*, um *filtro*, ou simplesmente os *pesos* da camada que são parâmetros frequentemente aprendíveis. Quando a região local é pequena, a diferença em comparação com uma rede totalmente conectada pode ser dramática. Embora anteriormente, pudéssemos ter exigido bilhões de parâmetros para representar apenas uma única camada em uma rede de processamento de imagem, agora precisamos de apenas algumas centenas, sem alterar a dimensionalidade de qualquer as entradas ou as representações ocultas. O preço pago por esta redução drástica de parâmetros é que nossos recursos agora são invariantes de tradução e que nossa camada só pode incorporar informações locais, ao determinar o valor de cada ativação oculta. Todo aprendizado depende da imposição de *bias* indutivos. Quando esses *bias* concordam com a realidade, obtemos modelos com amostras eficientes que generalizam bem para dados invisíveis. Mas é claro, se esses *bias* não concordam com a realidade, por exemplo, se as imagens acabassem não sendo invariantes à tradução, nossos modelos podem ter dificuldade até mesmo para se ajustar aos nossos dados de treinamento.

[1] *Convolutional Neural Networks.*

6.1.3 Convoluções

Antes de prosseguir, devemos revisar brevemente porque a operação acima é chamada de convolução. Em matemática, a *convolução* entre duas funções, digamos que $f, g : \mathbb{R}^d \to \mathbb{R}$ é definida como

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}. \tag{6.1.4}$$

Ou seja, medimos a sobreposição entre f e g quando uma função é "invertida" e deslocada por $\mathbf{x}$. Sempre que temos objetos discretos, a integral se transforma em uma soma. Por exemplo, para vetores do conjunto de vetores dimensionais infinitos somados ao quadrado com o índice acima de $\mathbb{Z}$, obtemos a seguinte definição:

$$(f * g)(i) = \sum_a f(a) g(i - a). \tag{6.1.5}$$

Para tensores bidimensionais, temos uma soma correspondente com índices (a, b) para f e $(i - a, j - b)$ para g, respectivamente:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b). \tag{6.1.6}$$

Isso é semelhante a (6.1.3), com uma grande diferença. Em vez de usar $(i + a, j + b)$, estamos usando a diferença. Observe, porém, que esta distinção é principalmente cosmética uma vez que sempre podemos combinar a notação entre (6.1.3) e (6.1.6). Nossa definição original em (6.1.3) mais apropriadamente descreve uma *correlação cruzada*. Voltaremos a isso na seção seguinte.

6.1.4 "Onde está Wally" Revisitado

Voltando ao nosso detector Wally, vamos ver como é. A camada convolucional escolhe janelas de um determinado tamanho e pesa as intensidades de acordo com o filtro V, conforme demonstrado em Fig. 6.1.2. Podemos ter como objetivo aprender um modelo para que onde quer que a "Wallyneza" seja mais alta, devemos encontrar um pico nas representações das camadas ocultas.

Fig. 6.1.2: Detectar Wally.

Canais

Existe apenas um problema com essa abordagem. Até agora, felizmente ignoramos que as imagens consistem de 3 canais: vermelho, verde e azul. Na realidade, as imagens não são objetos bidimensionais mas sim tensores de terceira ordem, caracterizados por uma altura, largura e canal, por exemplo, com forma $1024 \times 1024 \times 3$ pixels. Enquanto os dois primeiros desses eixos dizem respeito às relações espaciais, o terceiro pode ser considerado como atribuição uma representação multidimensional para cada localização de pixel. Assim, indexamos X como $[\mathsf{X}]_{i,j,k}$. O filtro convolucional deve se adaptar em conformidade. Em vez de $[\mathbf{V}]_{a,b}$, agora temos $[\mathsf{V}]_{a,b,c}$.

Além disso, assim como nossa entrada consiste em um tensor de terceira ordem, é uma boa ideia formular de forma semelhante nossas representações ocultas como tensores de terceira ordem H. Em outras palavras, em vez de apenas ter uma única representação oculta correspondendo a cada localização espacial, queremos todo um vetor de representações ocultas correspondente a cada localização espacial. Poderíamos pensar nas representações ocultas como abrangendo várias grades bidimensionais empilhadas umas sobre as outras. Como nas entradas, às vezes são chamados de *canais*. Eles também são chamados de *mapas de características*, já que cada um fornece um conjunto espacializado de recursos aprendidos para a camada subsequente. Intuitivamente, você pode imaginar que nas camadas inferiores que estão mais próximas das entradas, alguns canais podem se tornar especializados para reconhecer bordas enquanto outros podem reconhecer texturas.

Para suportar canais múltiplos em ambas as entradas (X) e representações ocultas (H), podemos adicionar uma quarta coordenada a V: $[\mathsf{V}]_{a,b,c,d}$. Juntando tudo, temos:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{6.1.7}$$

onde d indexa os canais de saída nas representações ocultas H. A camada convolucional subsequente irá tomar um tensor de terceira ordem, H, como entrada. Sendo mais geral, (6.1.7) é a definição de uma camada convolucional para canais múltiplos, onde V é um *kernel* ou filtro da camada.

Ainda existem muitas operações que precisamos abordar. Por exemplo, precisamos descobrir como combinar todas as representações ocultas para uma única saída, por exemplo, se há um Wally *em qualquer lugar* da imagem. Também precisamos decidir como computar as coisas de forma eficiente, como combinar várias camadas, funções de ativação apropriadas, e como fazer escolhas de design razoáveis para produzir redes eficazes na prática. Voltaremos a essas questões no restante do capítulo.

6.1.5 Resumo

- A invariância da tradução nas imagens implica que todas as manchas de uma imagem serão tratadas da mesma maneira.
- Localidade significa que apenas uma pequena vizinhança de pixels será usada para calcular as representações ocultas correspondentes.
- No processamento de imagem, as camadas convolucionais geralmente requerem muito menos parâmetros do que as camadas totalmente conectadas.
- CNNS são uma família especial de redes neurais que contêm camadas convolucionais.

- Os canais de entrada e saída permitem que nosso modelo capture vários aspectos de uma imagem em cada localização espacial.

6.1.6 Exercícios

1. Suponha que o tamanho do *kernel* de convolução seja $\Delta = 0$. Mostre que, neste caso, o *kernel* de convolução implementa um MLP independentemente para cada conjunto de canais.
2. Por que a invariância da tradução pode não ser uma boa ideia, afinal?
3. Com quais problemas devemos lidar ao decidir como tratar representações ocultas correspondentes a localizações de pixels na fronteira de uma imagem?
4. Descreva uma camada convolucional análoga para áudio.
5. Você acha que as camadas convolucionais também podem ser aplicáveis para dados de texto? Por que ou por que não?
6. Prove que $f * g = g * f$.

Discussions[66]

6.2 Convolução para Imagens

Agora que entendemos como as camadas convolucionais funcionam na teoria, estamos prontos para ver como eles funcionam na prática. Com base na nossa motivação de redes neurais convolucionais como arquiteturas eficientes para explorar a estrutura em dados de imagem, usamos imagens como nosso exemplo de execução.

6.2.1 A Operação de Correlação Cruzada

Lembre-se de que, estritamente falando, as camadas convolucionais são um nome impróprio, uma vez que as operações que elas expressam são descritos com mais precisão como correlações cruzadas. Com base em nossas descrições de camadas convolucionais em Section 6.1, em tal camada, um tensor de entrada e um tensor de *kernel* são combinados para produzir um tensor de saída por meio de uma operação de correlação cruzada.

Vamos ignorar os canais por enquanto e ver como isso funciona com dados bidimensionais e representações ocultas. Em Fig. 6.2.1, a entrada é um tensor bidimensional com altura de 3 e largura de 3. Marcamos a forma do tensor como 3×3 or $(3, 3)$. A altura e a largura do *kernel* são 2. A forma da *janela do kernel* (ou *janela de convolução*) é dada pela altura e largura do *kernel* (aqui é 2×2).

[66] https://discuss.d2l.ai/t/64

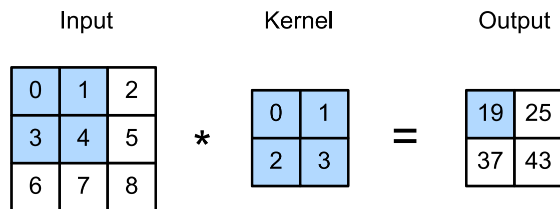


Fig. 6.2.1: Operação de correlação cruzada bidimensional. As partes sombreadas são o primeiro elemento de saída, bem como os elementos tensores de entrada e *kernel* usados para o cálculo de saída: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

Na operação de correlação cruzada bidimensional, começamos com a janela de convolução posicionada no canto superior esquerdo do tensor de entrada e o deslizamos pelo tensor de entrada, ambos da esquerda para a direita e de cima para baixo. Quando a janela de convolução desliza para uma determinada posição, o subtensor de entrada contido nessa janela e o tensor do *kernel* são multiplicados elemento a elemento e o tensor resultante é resumido produzindo um único valor escalar. Este resultado fornece o valor do tensor de saída no local correspondente. Aqui, o tensor de saída tem uma altura de 2 e largura de 2 e os quatro elementos são derivados de a operação de correlação cruzada bidimensional:

$$\begin{aligned} 0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43. \end{aligned} \tag{6.2.1}$$

Observe que ao longo de cada eixo, o tamanho da saída é ligeiramente menor que o tamanho de entrada. Como o *kernel* tem largura e altura maiores que um, só podemos calcular corretamente a correlação cruzada para locais onde o *kernel* se encaixa totalmente na imagem, o tamanho da saída é dado pelo tamanho da entrada $n_h \times n_w$ menos o tamanho do *kernel* de convolução $k_h \times k_w$ através da

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{6.2.2}$$

Este é o caso, pois precisamos de espaço suficiente para "deslocar" o *kernel* de convolução na imagem. Mais tarde, veremos como manter o tamanho inalterado preenchendo a imagem com zeros em torno de seu limite para que haja espaço suficiente para mudar o *kernel*. Em seguida, implementamos este processo na função `corr2d`, que aceita um tensor de entrada `X` e um tensor de *kernel* `K` e retorna um tensor de saída `Y`.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

Podemos construir o tensor de entrada `X` e o tensor do kernel`K` from Fig. 6.2.1 para validar o resultado da implementação acima da operação de correlação cruzada bidimensional.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

6.2.2 Camadas Convolucionais

Uma camada convolucional correlaciona a entrada e o *kernel* e adiciona um *bias* escalar para produzir uma saída. Os dois parâmetros de uma camada convolucional são o *kernel* e o *bias* escalar. Ao treinar modelos com base em camadas convolucionais, normalmente inicializamos os *kernels* aleatoriamente, assim como faríamos com uma camada totalmente conectada.

Agora estamos prontos para implementar uma camada convolucional bidimensional com base na função `corr2d` definida acima. Na função construtora `__init__`, declaramos `weight` e`bias` como os dois parâmetros do modelo. A função de propagação direta chama a função `corr2d` e adiciona o viés.

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Na convolução $h \times w$ ou um *kernel* de convolução $h \times w$ a altura e a largura do *kernel* de convolução são h e w, respectivamente. Também nos referimos a uma camada convolucional com um kernel de convolução $h \times w$ simplesmente como uma camada convolucional $h \times w$

6.2.3 Detecção de Borda de Objeto em Imagens

Vamos analisar uma aplicação simples de uma camada convolucional: detectar a borda de um objeto em uma imagem encontrando a localização da mudança de pixel. Primeiro, construímos uma "imagem" de 6×8 pixels. As quatro colunas do meio são pretas (0) e as demais são brancas (1).

```
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
```

(continues on next page)

(continued from previous page)

```
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

Em seguida, construímos um kernel `K` com uma altura de 1 e uma largura de 2. Quando realizamos a operação de correlação cruzada com a entrada, se os elementos horizontalmente adjacentes forem iguais, a saída é 0. Caso contrário, a saída é diferente de zero.

```
K = torch.tensor([[1.0, -1.0]])
```

Estamos prontos para realizar a operação de correlação cruzada com os argumentos `X` (nossa entrada) e`K` (nosso kernel). Como você pode ver, detectamos 1 para a borda do branco ao preto e -1 para a borda do preto ao branco. Todas as outras saídas assumem o valor 0.

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

Agora podemos aplicar o kernel à imagem transposta. Como esperado, ele desaparece. O kernel `K` detecta apenas bordas verticais.

```
corr2d(X.t(), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

6.2.4 Aprendendo um Kernel

Projetar um detector de borda por diferenças finitas `[1, -1]` é legal se sabemos que é exatamente isso que estamos procurando. No entanto, quando olhamos para *kernels* maiores, e considere camadas sucessivas de convoluções, pode ser impossível especificar exatamente o que cada filtro deve fazer manualmente.

Agora vamos ver se podemos aprender o *kernel* que gerou `Y` de`X` olhando apenas para os pares de entrada–saída. Primeiro construímos uma camada convolucional e inicializamos seu *kernel* como um tensor aleatório. A seguir, em cada iteração, usaremos o erro quadrático para comparar `Y` com a saída da camada convolucional. Podemos então calcular o gradiente para atualizar o *ker-*

nel. Por uma questão de simplicidade, na sequência nós usamos a classe embutida para camadas convolucionais bidimensionais e ignorar o *bias*.

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.Conv2d(1,1, kernel_size=(1, 2), bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))

for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    # Update the kernel
    conv2d.weight.data[:] -= 3e-2 * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'batch {i + 1}, loss {l.sum():.3f}')
```

```
batch 2, loss 9.670
batch 4, loss 2.786
batch 6, loss 0.944
batch 8, loss 0.354
batch 10, loss 0.139
```

Observe que o erro caiu para um valor pequeno após 10 iterações. Agora daremos uma olhada no tensor do *kernel* que aprendemos.

```
conv2d.weight.data.reshape((1, 2))
```

```
tensor([[ 1.0265, -0.9505]])
```

Indeed, the learned kernel tensor is remarkably close to the kernel tensor `K` we defined earlier.

6.2.5 Correlação Cruzada e Convolução

Lembre-se de nossa observação de Section 6.1 da correspondência entre as operações de correlação cruzada e convolução. Aqui, vamos continuar a considerar as camadas convolucionais bidimensionais. E se essas camadas realizar operações de convolução estritas conforme definido em (6.1.6) em vez de correlações cruzadas? Para obter a saída da operação de *convolução* estrita, precisamos apenas inverter o tensor do *kerne*l bidimensional tanto horizontal quanto verticalmente e, em seguida, executar a operação de *correlação cruzada* com o tensor de entrada.

É digno de nota que, uma vez que os *kernels* são aprendidos a partir de dados no aprendizado profundo, as saídas das camadas convolucionais permanecem inalteradas não importa se tais camadas executam as operações de convolução estrita ou as operações de correlação cruzada.

Para ilustrar isso, suponha que uma camada convolucional execute *correlação cruzada* e aprenda o *kernel* em Fig. 6.2.1, que é denotado como a matriz $\mathbf{K}$ aqui. Supondo que outras condições per-

maneçam inalteradas, quando esta camada executa *convolução* estrita em vez disso, o *kernel* aprendido $\mathbf{K}'$ será o mesmo que $\mathbf{K}$ depois que $\mathbf{K}'$ is é invertido horizontalmente e verticalmente. Quer dizer, quando a camada convolucional executa *convolução* estrita para a entrada em Fig. 6.2.1 e $\mathbf{K}'$, a mesma saída em Fig. 6.2.1 (correlação cruzada da entrada e $\mathbf{K}$) será obtida.

De acordo com a terminologia padrão da literatura de *deep learning*, continuaremos nos referindo à operação de correlação cruzada como uma convolução, embora, estritamente falando, seja ligeiramente diferente. Além do mais, usamos o termo *elemento* para nos referirmos a uma entrada (ou componente) de qualquer tensor que representa uma representação de camada ou um *kernel* de convolução.

6.2.6 Mapa de Características e Campo Receptivo

Conforme descrito em Section 6.1.4, a saída da camada convolucional em Fig. 6.2.1 às vezes é chamada de *mapa de características*, pois pode ser considerado como as representações aprendidas (características) nas dimensões espaciais (por exemplo, largura e altura) para a camada subsequente. Nas CNNs, para qualquer elemento x de alguma camada, seu *campo receptivo* refere-se a todos os elementos (de todas as camadas anteriores) que pode afetar o cálculo de x durante a propagação direta. Observe que o campo receptivo pode ser maior do que o tamanho real da entrada.

Vamos continuar a usar Fig. 6.2.1 para explicar o campo receptivo. Dado o *kernel* de convolução 2×2 o campo receptivo do elemento de saída sombreado (de valor 19) são os quatro elementos na parte sombreada da entrada. Agora, vamos denotar a saída 2×2 como $\mathbf{Y}$ e considere uma CNN mais profunda com uma camada convolucional adicional 2×2 que leva $\mathbf{Y}$ como sua entrada, produzindo um único elemento z. Nesse caso, o campo receptivo de z em $\mathbf{Y}$ inclui todos os quatro elementos de $\mathbf{Y}$, enquanto o campo receptivo na entrada inclui todos os nove elementos de entrada. Por isso, quando qualquer elemento em um mapa de recursos precisa de um campo receptivo maior para detectar recursos de entrada em uma área mais ampla, podemos construir uma rede mais profunda.

6.2.7 Resumo

- O cálculo central de uma camada convolucional bidimensional é uma operação de correlação cruzada bidimensional. Em sua forma mais simples, isso executa uma operação de correlação cruzada nos dados de entrada bidimensionais e no *kernel* e, em seguida, adiciona um *bias*.
- Podemos projetar um *kernel* para detectar bordas em imagens.
- Podemos aprender os parâmetros do *kernel* a partir de dados.
- Com os *kernels* aprendidos a partir dos dados, as saídas das camadas convolucionais permanecem inalteradas, independentemente das operações realizadas por essas camadas (convolução estrita ou correlação cruzada).
- Quando qualquer elemento em um mapa de características precisa de um campo receptivo maior para detectar características mais amplas na entrada, uma rede mais profunda pode ser considerada.

6.2.8 Exercícios

1. Construa uma imagem `X` com bordas diagonais.
    1. O que acontece se você aplicar o *kernel* `K` nesta seção a ele?
    2. O que acontece se você transpõe `X`?
    3. O que acontece se você transpõe `K`?
2. Quando você tenta encontrar automaticamente o gradiente para a classe `Conv2D` que criamos, que tipo de mensagem de erro você vê?
3. Como você representa uma operação de correlação cruzada como uma multiplicação de matriz, alterando os tensores de entrada e *kernel*?
4. Projete alguns *kernels* manualmente.
    1. Qual é a forma de um *kernel* para a segunda derivada?
    2. Qual é o *kernel* de uma integral?
    3. Qual é o tamanho mínimo de um *kernel* para obter uma derivada de grau d?

Discussions[67]

6.3 Preenchimento e Saltos

No exemplo anterior de Fig. 6.2.1, nossa entrada tinha altura e largura de 3 e nosso núcleo de convolução tinha altura e largura de 2, produzindo uma representação de saída com dimensão 2×2. Como generalizamos em Section 6.2, assumindo que a forma de entrada é $n_h \times n_w$ e a forma do kernel de convolução é $k_h \times k_w$, então a forma de saída será $(n_h - k_h + 1) \times (n_w - k_w + 1)$. Portanto, a forma de saída da camada convolucional é determinada pela forma da entrada e a forma do núcleo de convolução.

Em vários casos, incorporamos técnicas, incluindo preenchimento e convoluções com saltos, que afetam o tamanho da saída. Como motivação, note que uma vez que os *kernels* geralmente têm largura e altura maiores que 1, depois de aplicar muitas convoluções sucessivas, tendemos a acabar com resultados que são consideravelmente menor do que nossa entrada. Se começarmos com uma imagem de 240×240 pixels, 10 camadas de 5×5 convoluções reduzem a imagem para 200×200 pixels, cortando 30% da imagem e com ela obliterando qualquer informação interessante nos limites da imagem original. *Preenchimento* é a ferramenta mais popular para lidar com esse problema.

In other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be unwieldy. *Strided convolutions* are a popular technique that can help in these instances.

[67] https://discuss.d2l.ai/t/66

6.3.1 Preenchimento

Conforme descrito acima, um problema complicado ao aplicar camadas convolucionais é que tendemos a perder pixels no perímetro de nossa imagem. Uma vez que normalmente usamos pequenos *kernels*, para qualquer convolução dada, podemos perder apenas alguns pixels, mas isso pode somar conforme aplicamos muitas camadas convolucionais sucessivas. Uma solução direta para este problema é adicionar pixels extras de preenchimento ao redor do limite de nossa imagem de entrada, aumentando assim o tamanho efetivo da imagem. Normalmente, definimos os valores dos pixels extras para zero. Em Fig. 6.3.1, preenchemos uma entrada 3×3, aumentando seu tamanho para 5×5. A saída correspondente então aumenta para uma matriz 4×4 As partes sombreadas são o primeiro elemento de saída, bem como os elementos tensores de entrada e kernel usados para o cálculo de saída: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

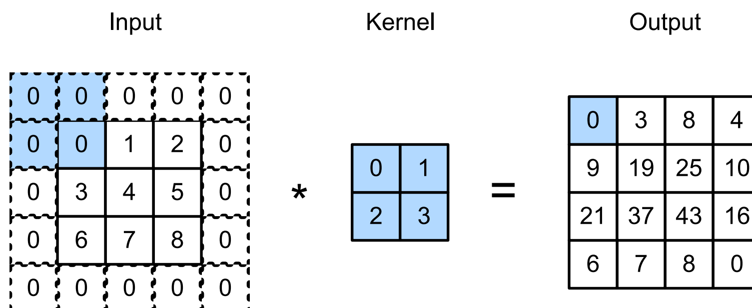


Fig. 6.3.1: Correlação cruzada bidimensional com preenchimento.

Em geral, se adicionarmos um total de p_h linhas de preenchimento (cerca de metade na parte superior e metade na parte inferior) e um total de p_w colunas de preenchimento (cerca de metade à esquerda e metade à direita), a forma de saída será

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \tag{6.3.1}$$

Isso significa que a altura e largura da saída aumentará em p_h e p_w, respectivamente.

Em muitos casos, queremos definir $p_h = k_h - 1$ e $p_w = k_w - 1$ para dar à entrada e saída a mesma altura e largura. Isso tornará mais fácil prever a forma de saída de cada camada ao construir a rede. Supondo que k_h seja estranho aqui, vamos preencher $p_h/2$ linhas em ambos os lados da altura. Se k_h for par, uma possibilidade é juntar $\lceil p_h/2 \rceil$ linhas no topo da entrada e $\lfloor p_h/2 \rfloor$ linhas na parte inferior. Vamos preencher ambos os lados da largura da mesma maneira.

CNNs geralmente usam *kernels* de convolução com valores de altura e largura ímpares, como 1, 3, 5 ou 7. Escolher tamanhos ímpares de *kernel* tem o benefício que podemos preservar a dimensionalidade espacial enquanto preenche com o mesmo número de linhas na parte superior e inferior, e o mesmo número de colunas à esquerda e à direita.

Além disso, esta prática de usar *kernels* estranhos e preenchimento para preservar precisamente a dimensionalidade oferece um benefício administrativo. Para qualquer tensor bidimensional `X`, quando o tamanho do *kernel* é estranho e o número de linhas e colunas de preenchimento em todos os lados são iguais, produzindo uma saída com a mesma altura e largura da entrada, sabemos que a saída `Y [i, j]` é calculada por correlação cruzada do kernel de entrada e convolução com a janela centralizada em `X [i, j]`.

No exemplo a seguir, criamos uma camada convolucional bidimensional com altura e largura de 3 e aplique 1 pixel de preenchimento em todos os lados. Dada uma entrada com altura e largura de 8, descobrimos que a altura e a largura da saída também é 8.

```python
import torch
from torch import nn


# We define a convenience function to calculate the convolutional layer. This
# function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    # Here (1, 1) indicates that the batch size and the number of channels
    # are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: examples and
    # channels
    return Y.reshape(Y.shape[2:])
# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

Quando a altura e largura do núcleo de convolução são diferentes, podemos fazer com que a saída e a entrada tenham a mesma altura e largura definindo diferentes números de preenchimento para altura e largura.

```python
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on either side of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

6.3.2 Saltos

Ao calcular a correlação cruzada, começamos com a janela de convolução no canto superior esquerdo do tensor de entrada, e o deslizamos sobre todos os locais para baixo e para a direita. Nos exemplos anteriores, optamos por deslizar um elemento de cada vez. No entanto, às vezes, seja para eficiência computacional ou porque desejamos reduzir a resolução, movemos nossa janela mais de um elemento por vez, pulando os locais intermediários.

Nos referimos ao número de linhas e colunas percorridas por slide como o *salto*. Até agora, usamos saltos de 1, tanto para altura quanto para largura. Às vezes, podemos querer dar um salto maior. Fig. 6.3.2 mostra uma operação de correlação cruzada bidimensional com um salto de 3 na vertical e 2 na horizontal. As partes sombreadas são os elementos de saída, bem como os elementos tensores de entrada e *kernel* usados para o cálculo de saída: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$. Podemos ver que quando o segundo elemento da primeira coluna é gerado, a janela de convolução desliza três fileiras para baixo. A janela de convolução desliza duas

colunas para a direita quando o segundo elemento da primeira linha é gerado. Quando a janela de convolução continua a deslizar duas colunas para a direita na entrada, não há saída porque o elemento de entrada não pode preencher a janela (a menos que adicionemos outra coluna de preenchimento).

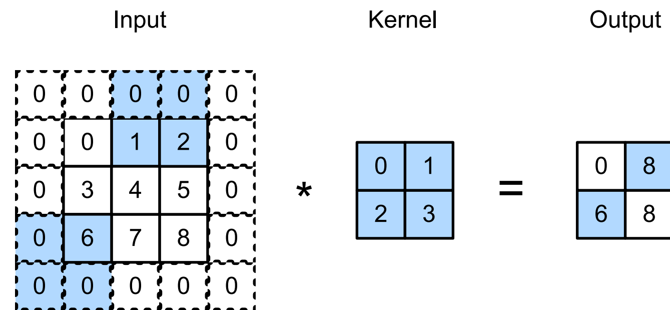


Fig. 6.3.2: Correlação cruzada com passos de 3 e 2 para altura e largura, respectivamente.

Em geral, quando o salto para a altura é s_h e a distância para a largura é s_w, a forma de saída é

$$\lfloor(n_h-k_h+p_h+s_h)/s_h\rfloor \times \lfloor(n_w-k_w+p_w+s_w)/s_w\rfloor. \tag{6.3.2}$$

Se definirmos $p_h=k_h-1$ e $p_w=k_w-1$, então a forma de saída será simplificada para $\lfloor(n_h+s_h-1)/s_h\rfloor \times \lfloor(n_w+s_w-1)/s_w\rfloor$. Indo um passo adiante, se a altura e largura de entrada são divisíveis pelos saltos na altura e largura, então a forma de saída será $(n_h/s_h) \times (n_w/s_w)$.

Abaixo, definimos os saltos de altura e largura para 2, reduzindo assim pela metade a altura e a largura da entrada.

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

A seguir, veremos um exemplo um pouco mais complicado.

```
conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

Por uma questão de brevidade, quando o número de preenchimento em ambos os lados da altura e largura de entrada são p_h ep_w respectivamente, chamamos o preenchimento (p_h, p_w). Especificamente, quando $p_h = p_w = p$, o preenchimento é p. Quando os saltos de altura e largura são s_h e s_w, respectivamente, chamamos o salto de (s_h, s_w). Especificamente, quando $s_h = s_w = s$, , o salto é s. Por padrão, o preenchimento é 0 e a salto é 1. Na prática, raramente usamos saltos não homogêneos ou preenchimento, ou seja, geralmente temos $p_h = p_w$ e $s_h = s_w$.

6.3.3 Resumo

- O preenchimento pode aumentar a altura e a largura da saída. Isso geralmente é usado para dar à saída a mesma altura e largura da entrada.
- Os saltos podem reduzir a resolução da saída, por exemplo, reduzindo a altura e largura da saída para apenas $1/n$ da altura e largura da entrada (n é um número inteiro maior que 1).
- Preenchimento e saltos podem ser usados para ajustar a dimensionalidade dos dados de forma eficaz.

Exercises

1. Para o último exemplo nesta seção, use matemática para calcular a forma de saída para ver se é consistente com o resultado experimental.
2. Experimente outras combinações de preenchimento e saltos nos experimentos desta seção.
3. Para sinais de áudio, a que corresponde um salto de 2?
4. Quais são os benefícios computacionais de uma salto maior que 1?

Discussions[68]

6.4 Canais de Múltiplas Entradas e Saídas

Embora tenhamos descrito os vários canais que compõem cada imagem (por exemplo, imagens coloridas têm os canais RGB padrão para indicar a quantidade de vermelho, verde e azul) e camadas convolucionais para vários canais em Section 6.1.4, até agora, simplificamos todos os nossos exemplos numéricos trabalhando com apenas uma única entrada e um único canal de saída. Isso nos permitiu pensar em nossas entradas, *kernels* de convolução, e saídas, cada um como tensores bidimensionais.

Quando adicionamos canais a isto, nossas entradas e representações ocultas ambas se tornam tensores tridimensionais. Por exemplo, cada imagem de entrada RGB tem a forma $3 \times h \times w$. Referimo-nos a este eixo, com um tamanho de 3, como a dimensão do *canal*. Nesta seção, daremos uma olhada mais detalhada em núcleos de convolução com múltiplos canais de entrada e saída.

6.4.1 Canais de Entrada Múltiplos

Quando os dados de entrada contêm vários canais, precisamos construir um *kernel* de convolução com o mesmo número de canais de entrada que os dados de entrada, para que possa realizar correlação cruzada com os dados de entrada. Supondo que o número de canais para os dados de entrada seja c_i, o número de canais de entrada do *kernel* de convolução também precisa ser c_i. Se a forma da janela do nosso kernel de convolução é $k_h \times k_w$, então quando $c_i = 1$, podemos pensar em nosso kernel de convolução apenas como um tensor bidimensional de forma $k_h \times k_w$.

No entanto, quando $c_i > 1$, precisamos de um kernel que contém um tensor de forma $k_h \times k_w$ para *cada* canal de entrada. Concatenando estes c_i tensores juntos produz um kernel de convolução de

[68] https://discuss.d2l.ai/t/68

forma $c_i \times k_h \times k_w$. Uma vez que o *kernel* de entrada e convolução tem cada um c_i canais, podemos realizar uma operação de correlação cruzada no tensor bidimensional da entrada e o tensor bidimensional do núcleo de convolução para cada canal, adicionando os resultados c_i juntos (somando os canais) para produzir um tensor bidimensional. Este é o resultado de uma correlação cruzada bidimensional entre uma entrada multicanal e um *kernel* de convolução com vários canais de entrada.

Em Fig. 6.4.1, demonstramos um exemplo de uma correlação cruzada bidimensional com dois canais de entrada. As partes sombreadas são o primeiro elemento de saída bem como os elementos tensores de entrada e kernel usados para o cálculo de saída: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

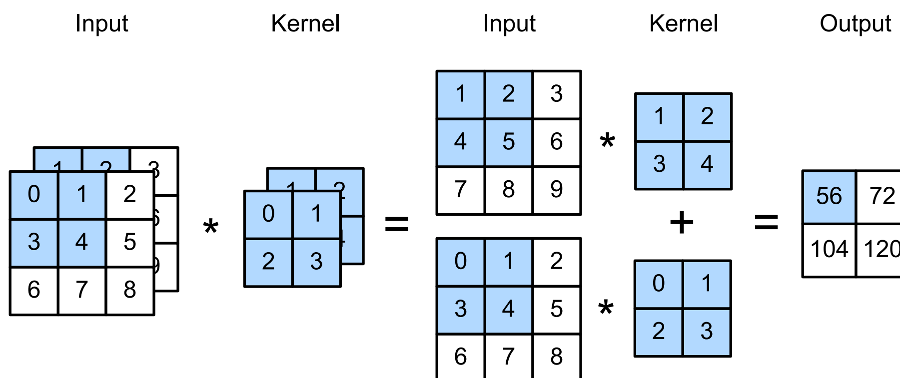


Fig. 6.4.1: Cálculo de correlação cruzada com 2 canais de entrada.

Para ter certeza de que realmente entendemos o que está acontecendo aqui, podemos implementar operações de correlação cruzada com vários canais de entrada. Observe que tudo o que estamos fazendo é realizar uma operação de correlação cruzada por canal e depois somando os resultados.

```
import torch
from d2l import torch as d2l
```

```
def corr2d_multi_in(X, K):
    # First, iterate through the 0th dimension (channel dimension) of `X` and
    # `K`. Then, add them together
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

Podemos construir o tensor de entrada `X` e o tensor do kernel`K` correspondendo aos valores em Fig. 6.4.1 para validar a saída da operação de correlação cruzada.

```
X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
               [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

6.4.2 Canais de Saída Múltiplos

Independentemente do número de canais de entrada, até agora acabamos sempre com um canal de saída. No entanto, como discutimos em Section 6.1.4, é essencial ter vários canais em cada camada. Nas arquiteturas de rede neural mais populares, na verdade, aumentamos a dimensão do canal à medida que subimos na rede neural, normalmente reduzindo as amostras para compensar a resolução espacial para maior *profundidade do canal*. Intuitivamente, você pode pensar em cada canal como respondendo a algum conjunto diferente de *features*. A realidade é um pouco mais complicada do que as interpretações mais ingênuas dessa intuição, uma vez que as representações não são aprendidas de forma independente, mas sim otimizadas para serem úteis em conjunto. Portanto, pode não ser que um único canal aprenda um detector de bordas, mas sim que alguma direção no espaço do canal corresponde à detecção de bordas.

Denote por c_i e c_o o número dos canais de entrada e saída, respectivamente, e sejam k_h e k_w a altura e a largura do *kernel*. Para obter uma saída com vários canais, podemos criar um tensor de kernel da forma $c_i \times k_h \times k_w$ para *cada* canal de saída. Nós os concatenamos na dimensão do canal de saída, de modo que a forma do núcleo de convolução é $c_o \times c_i \times k_h \times k_w$. Em operações de correlação cruzada, o resultado em cada canal de saída é calculado do *kernel* de convolução correspondente a esse canal de saída e recebe a entrada de todos os canais no tensor de entrada.

Implementamos uma função de correlação cruzada para calcular a saída de vários canais, conforme mostrado abaixo.

```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of `K`, and each time, perform
    # cross-correlation operations with input `X`. All of the results are
    # stacked together
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

Construímos um *kernel* de convolução com 3 canais de saída concatenando o tensor do kernel `K` com`K + 1` (mais um para cada elemento em `K`) e`K + 2`.

```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

Abaixo, realizamos operações de correlação cruzada no tensor de entrada `X` com o tensor do kernel`K`. Agora a saída contém 3 canais. O resultado do primeiro canal é consistente com o resultado do tensor de entrada anterior `X` e o canal de múltiplas entradas, *kernel* do canal de saída única.

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

6.4.3 Camada Convolucional 1×1

No início, uma convolução 1×1, ou seja, $k_h = k_w = 1$, não parece fazer muito sentido. Afinal, uma convolução correlaciona pixels adjacentes. Uma convolução 1×1 obviamente não faz isso. No entanto, são operações populares que às vezes são incluídas nos projetos de redes profundas complexas. Vejamos com alguns detalhes o que ele realmente faz.

Como a janela mínima é usada, a convolução 1×1 perde a capacidade de camadas convolucionais maiores para reconhecer padrões que consistem em interações entre os elementos adjacentes nas dimensões de altura e largura. O único cálculo da convolução 1×1 ocorre na dimensão do canal.

Fig. 6.4.2 mostra o cálculo de correlação cruzada usando o kernel de convolução 1×1 com 3 canais de entrada e 2 canais de saída. Observe que as entradas e saídas têm a mesma altura e largura. Cada elemento na saída é derivado de uma combinação linear de elementos *na mesma posição* na imagem de entrada. Você poderia pensar na camada convolucional 1×1 como constituindo uma camada totalmente conectada aplicada em cada localização de pixel para transformar os valores de entrada correspondentes c_i em valores de saída c_o. Porque esta ainda é uma camada convolucional, os pesos são vinculados à localização do pixel. Assim, a camada convolucional 1×1 requer pesos $c_o \times c_i$ (mais o *bias*).

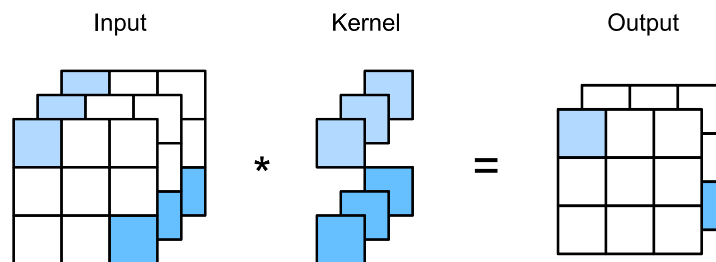


Fig. 6.4.2: O cálculo de correlação cruzada usa o *kernel* de convolução 1×1 com 3 canais de entrada e 2 canais de saída. A entrada e a saída têm a mesma altura e largura.

Vamos verificar se isso funciona na prática: implementamos uma convolução 1×1 usando uma camada totalmente conectada. A única coisa é que precisamos fazer alguns ajustes para a forma de dados antes e depois da multiplicação da matriz.

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    Y = torch.matmul(K, X)  # Matrix multiplication in the fully-connected layer
    return Y.reshape((c_o, h, w))
```

Ao realizar convolução 1×1 , a função acima é equivalente à função de correlação cruzada implementada anteriormente `corr2d_multi_in_out`. Vamos verificar isso com alguns dados de amostra.

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
```

```
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

6.4.4 Resumo

- Vários canais podem ser usados para estender os parâmetros do modelo da camada convolucional.
- A camada convolucional 1×1 é equivalente à camada totalmente conectada, quando aplicada por pixel.
- A camada convolucional 1×1 é normalmente usada para ajustar o número de canais entre as camadas de rede e para controlar a complexidade do modelo.

6.4.5 Exercícios

1. Suponha que temos dois *kernels* de convolução de tamanho k_1 e k_2, respectivamente (sem não linearidade entre eles).
    1. Prove que o resultado da operação pode ser expresso por uma única convolução.
    2. Qual é a dimensionalidade da convolução única equivalente?
    3. O inverso é verdadeiro?
2. Assuma uma entrada de forma $c_i \times h \times w$ e um *kernel* de convolução de forma $c_o \times c_i \times k_h \times k_w$, preenchimento de (p_h, p_w), e passo de (s_h, s_w).
    1. Qual é o custo computacional (multiplicações e adições) para a propagação direta?
    2. Qual é a pegada de memória?
    3. Qual é a pegada de memória para a computação reversa?
    4. Qual é o custo computacional para a retropropagação?
3. Por que fator o número de cálculos aumenta se dobrarmos o número de canais de entrada c_i e o número de canais de saída c_o? O que acontece se dobrarmos o preenchimento?
4. Se a altura e largura de um *kernel* de convolução é $k_h = k_w = 1$,, qual é a complexidade computacional da propagação direta?
5. As variáveis `Y1` e`Y2` no último exemplo desta seção são exatamente as mesmas? Porque?
6. Como você implementaria convoluções usando a multiplicação de matrizes quando a janela de convolução não é 1×1?

Discussions[69]

[69] https://discuss.d2l.ai/t/70

6.5 *Pooling*

Muitas vezes, conforme processamos imagens, queremos gradualmente reduzir a resolução espacial de nossas representações ocultas, agregando informações para que quanto mais alto subimos na rede, maior o campo receptivo (na entrada) ao qual cada nó oculto é sensível.

Muitas vezes, nossa tarefa final faz alguma pergunta global sobre a imagem, por exemplo, *contém um gato?* Então, normalmente, as unidades de nossa camada final devem ser sensíveis para toda a entrada. Ao agregar informações gradualmente, produzindo mapas cada vez mais grosseiros, alcançamos esse objetivo de, em última análise, aprendendo uma representação global, enquanto mantém todas as vantagens das camadas convolucionais nas camadas intermediárias de processamento.

Além disso, ao detectar recursos de nível inferior, como bordas (conforme discutido em Section 6.2), frequentemente queremos que nossas representações sejam um tanto invariáveis à tradução. Por exemplo, se pegarmos a imagem `X` com uma delimitação nítida entre preto e branco e deslocarmos a imagem inteira em um pixel para a direita, ou seja, `Z [i, j] = X [i, j + 1]`, então a saída para a nova imagem `Z` pode ser muito diferente. A borda terá deslocado um pixel. Na realidade, os objetos dificilmente ocorrem exatamente no mesmo lugar. Na verdade, mesmo com um tripé e um objeto estacionário, a vibração da câmera devido ao movimento do obturador pode mudar tudo em um pixel ou mais (câmeras de última geração são carregadas com recursos especiais para resolver esse problema).

Esta seção apresenta *camadas de pooling*, que servem ao duplo propósito de mitigando a sensibilidade das camadas convolucionais à localização e de representações de *downsampling* espacialmente.

6.5.1 *Pooling* Máximo e *Pooling* Médio

Como camadas convolucionais, operadores de *pooling* consistem em uma janela de formato fixo que é deslizada todas as regiões na entrada de acordo com seu passo, computando uma única saída para cada local percorrido pela janela de formato fixo (também conhecida como *janela de pooling*). No entanto, ao contrário do cálculo de correlação cruzada das entradas e grãos na camada convolucional, a camada de *pooling* não contém parâmetros (não há *kernel*). Em vez disso, os operadores de *pooling* são determinísticos, normalmente calculando o valor máximo ou médio dos elementos na janela de *pooling*. Essas operações são chamadas de *pooling máximo* (*pooling máximo* para breve) e *pooling médio*, respectivamente.

Em ambos os casos, como com o operador de correlação cruzada, podemos pensar na janela de *pooling* começando da parte superior esquerda do tensor de entrada e deslizando pelo tensor de entrada da esquerda para a direita e de cima para baixo. Em cada local que atinge a janela de *pooling*, ele calcula o máximo ou o médio valor do subtensor de entrada na janela, dependendo se o *pooling* máximo ou médio é empregado.

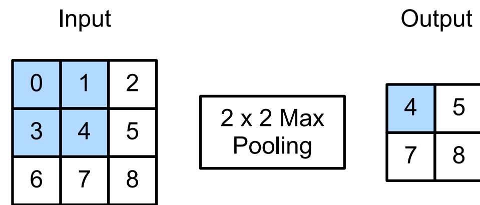


Fig. 6.5.1: Pooling máximo com uma forma de janela de pool de 2×2. As partes sombreadas são o primeiro elemento de saída, bem como os elementos tensores de entrada usados para o cálculo de saída: $\max(0, 1, 3, 4) = 4$.

O tensor de saída em Fig. 6.5.1 tem uma altura de 2 e uma largura de 2. Os quatro elementos são derivados do valor máximo em cada janela de *pooling*:

$$\begin{aligned} \max(0, 1, 3, 4) = 4, \\ \max(1, 2, 4, 5) = 5, \\ \max(3, 4, 6, 7) = 7, \\ \max(4, 5, 7, 8) = 8. \end{aligned} \tag{6.5.1}$$

Uma camada de *pooling* com uma forma de janela de pool de $p \times q$ é chamado de camada de *pooling* $p \times q$ A operação de *pooling* é chamada $p \times q$ *pooling*.

Vamos retornar ao exemplo de detecção de borda do objeto mencionado no início desta seção. Agora vamos usar a saída da camada convolucional como entrada para 2×2 *pooling* máximo. Defina a entrada da camada convolucional como `X` e a saída da camada de pooling como`Y`. Se os valores de `X [i, j]` e `X [i, j + 1]` são ou não diferentes, ou `X [i, j + 1]` e `X [i, j + 2]` são diferentes, a camada de pool sempre produz `Y [i, j] = 1`. Ou seja, usando a camada de pooling máxima 2×2 ainda podemos detectar se o padrão reconhecido pela camada convolucional move no máximo um elemento em altura ou largura.

No código abaixo, implementamos a propagação direta da camada de *pooling* na função `pool2d`. Esta função é semelhante à função `corr2d` in: numref: `sec_conv_layer`. No entanto, aqui não temos *kernel*, computando a saída como o máximo ou a média de cada região na entrada.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

Podemos construir o tensor de entrada `X` em Fig. 6.5.1 para validar a saída da camada de *pooling* máximo bidimensional.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

Além disso, experimentamos a camada de *pooling* média.

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

6.5.2 Preenchimento e Passos

Tal como acontece com as camadas convolucionais, camadas de *pooling* também podem alterar a forma de saída. E como antes, podemos alterar a operação para obter uma forma de saída desejada preenchendo a entrada e ajustando o passo. Podemos demonstrar o uso de preenchimento e passos em camadas de agrupamento por meio da camada de agrupamento máximo bidimensional integrada do framework de *deep learning*. Primeiro construímos um tensor de entrada `X` cuja forma tem quatro dimensões, onde o número de exemplos (tamanho do lote) e o número de canais são ambos 1.

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Por padrão, o passo e a janela de *pooling* na instância da classe interna do *framework* têm a mesma forma. Abaixo, usamos uma janela de *pooling* de forma `(3, 3)`, portanto, obtemos uma forma de passo de `(3, 3)` por padrão.

```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

```
tensor([[[[10.]]]])
```

O passo e o preenchimento podem ser especificados manualmente.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

Claro, podemos especificar uma janela de *pooling* retangular arbitrária e especificar o preenchimento e o passo para altura e largura, respectivamente. Para `nn.MaxPool2D`, o preenchimento deve ser menor que a metade do kernel_size. Se a condição não for atendida, podemos primeiro preenchemos a entrada usando `nn.functional.pad` e, em seguida, o passamos para a camada de *pooling*.

Claro, podemos especificar uma janela de *pooling* retangular arbitrária e especificar o preenchimento e o passo para altura e largura, respectivamente. No TensorFlow, para implementar um preenchimento de 1 em todo o tensor, uma função projetada para preenchimento deve ser invocada usando `tf.pad`. Isso implementará o preenchimento necessário e permitirá que o supracitado (3, 3) agrupamento com uma passada (2, 2) para realizar semelhantes aos do PyTorch e MXNet. Ao preencher desta forma, a variável embutida `padding` deve ser definida como `válida`.

```
X_pad = nn.functional.pad(X, (2, 2, 1, 1))
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3))
pool2d(X_pad)
```

```
tensor([[[[ 0.,  3.],
          [ 8., 11.],
          [12., 15.]]]])
```

6.5.3 Canais Múltiplos

Ao processar dados de entrada multicanal, a camada de *pooling* agrupa cada canal de entrada separadamente, em vez de somar as entradas nos canais como em uma camada convolucional. Isso significa que o número de canais de saída para a camada de *pooling* é igual ao número de canais de entrada. Abaixo, vamos concatenar os tensores `X` e`X + 1` na dimensão do canal para construir uma entrada com 2 canais.

Observe que isso exigirá um concatenação ao longo da última dimensão do TensorFlow devido à sintaxe dos últimos canais.

```
X = torch.cat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

Como podemos ver, o número de canais de saída ainda é 2 após o *pooling*.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

Observe que a saída para o *pooling* de tensorflow parece à primeira vista ser diferente, no entanto numericamente, os mesmos resultados são apresentados como MXNet e PyTorch. A diferença está na dimensionalidade, e na leitura do a saída verticalmente produz a mesma saída que as outras implementações.

6.5.4 Resumo

- Pegando os elementos de entrada na janela de agrupamento, a operação de agrupamento máxima atribui o valor máximo como a saída e a operação de agrupamento média atribui o valor médio como a saída.
- Um dos principais benefícios de uma camada de *pooling* é aliviar a sensibilidade excessiva da camada convolucional ao local.
- Podemos especificar o preenchimento e a passada para a camada de *pooling*.
- O agrupamento máximo, combinado com uma passada maior do que 1, pode ser usado para reduzir as dimensões espaciais (por exemplo, largura e altura).
- O número de canais de saída da camada de *pooling* é igual ao número de canais de entrada.

6.5.5 Exercícios

1. Você pode implementar o *pooling* médio como um caso especial de uma camada de convolução? Se sim, faça.
2. Você pode implementar o *pooling* máximo como um caso especial de uma camada de convolução? Se for assim, faça.
3. Qual é o custo computacional da camada de *pooling*? Suponha que a entrada para a camada de *pooling* seja do tamanho $c \times h \times w$, a janela de *pooling* tem um formato de $p_h \times p_w$ com um preenchimento de (p_h, p_w) e um passo de (s_h, s_w).
4. Por que você espera que o *pooling* máximo e o *pooling* médio funcionem de maneira diferente?
5. Precisamos de uma camada mínima de *pooling* separada? Você pode substituí-la por outra operação?
6. Existe outra operação entre o *pooling* médio e máximo que você possa considerar (dica: lembre-se do *softmax*)? Por que não é tão popular?

Discussions[70]

[70] https://discuss.d2l.ai/t/72

6.6 Redes Neurais Convolucionais (LeNet)

Agora temos todos os ingredientes necessários para montar uma CNN totalmente funcional. Em nosso encontro anterior com dados de imagem, nós aplicamos um modelo de regressão *softmax* (:numref: `sec_softmax_scratch`) e um modelo MLP (Section 4.2) a fotos de roupas no conjunto de dados Fashion-MNIST. Para tornar esses dados passíveis de regressão *softmax* e MLPs, primeiro nivelamos cada imagem de uma matriz 28×28 em um vetor de comprimento fixo 784-dimensional, e depois os processamos com camadas totalmente conectadas. Agora que temos um controle sobre as camadas convolucionais, podemos reter a estrutura espacial em nossas imagens. Como um benefício adicional de substituir camadas totalmente conectadas por camadas convolucionais, desfrutaremos de modelos mais parcimoniosos que requerem muito menos parâmetros.

Nesta seção, apresentaremos *LeNet*, entre as primeiras CNNs publicadas para chamar a atenção para seu desempenho em tarefas de visão computacional. O modelo foi apresentado por (e nomeado em homenagem a) Yann LeCun, em seguida, um pesquisador da AT&T Bell Labs, para fins de reconhecimento de dígitos manuscritos em imagens (LeCun et al., 1998). Este trabalho representou o culminar de uma década de pesquisa desenvolvendo a tecnologia. Em 1989, LeCun publicou o primeiro estudo com sucesso treinar CNNs via retropropagação.

Na época, LeNet alcançou resultados excelentes combinando o desempenho das máquinas de vetores de suporte, em seguida, uma abordagem dominante na aprendizagem supervisionada. LeNet foi eventualmente adaptado para reconhecer dígitos para processar depósitos em caixas eletrônicos. Até hoje, alguns caixas eletrônicos ainda executam o código que Yann e seu colega Leon Bottou escreveram na década de 1990!

6.6.1 LeNet

Em um alto nível, LeNet (LeNet-5) consiste em duas partes: (i) um codificador convolucional que consiste em duas camadas convolucionais; e (ii) um bloco denso que consiste em três camadas totalmente conectadas; A arquitetura é resumida em Fig. 6.6.1.

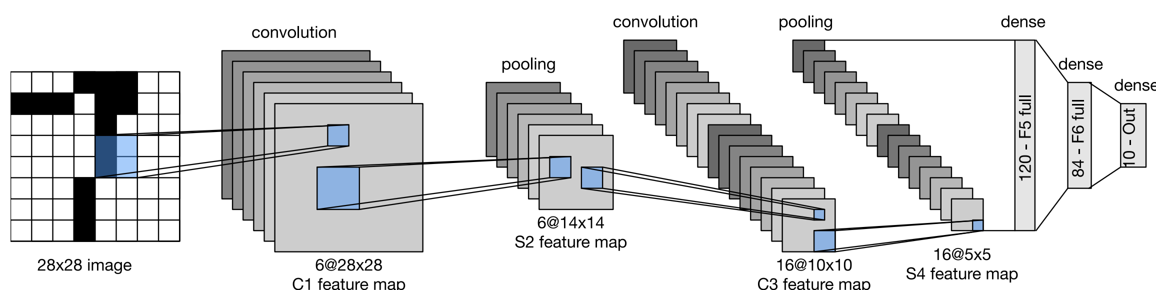


Fig. 6.6.1: Fluxo de dados em LeNet. A entrada é um dígito escrito à mão, a saída uma probabilidade de mais de 10 resultados possíveis.

As unidades básicas em cada bloco convolucional são uma camada convolucional, uma função de ativação *sigmoid* e uma subsequente operação média de *pooling*. Observe que, embora ReLUs

e *max-pooling* funcionem melhor, essas descobertas ainda não haviam sido feitas na década de 1990. Cada camada convolucional usa um *kernel* 5×5 e uma função de ativação *sigmoid*. Essas camadas mapeiam entradas organizadas espacialmente a uma série de mapas de recursos bidimensionais, normalmente aumentando o número de canais. A primeira camada convolucional tem 6 canais de saída, enquanto o segundo tem 16. Cada operação de *pooling* 2×2 (passo 2) reduz a dimensionalidade por um fator de 4 por meio da redução da resolução espacial. O bloco convolucional emite uma saída com forma dada por (tamanho do lote, número de canal, altura, largura).

Para passar a saída do bloco convolucional para o bloco denso, devemos nivelar cada exemplo no *minibatch*. Em outras palavras, pegamos essa entrada quadridimensional e a transformamos na entrada bidimensional esperada por camadas totalmente conectadas: como um lembrete, a representação bidimensional que desejamos usa a primeira dimensão para indexar exemplos no *minibatch* e o segundo para dar a representação vetorial plana de cada exemplo. O bloco denso do LeNet tem três camadas totalmente conectadas, com 120, 84 e 10 saídas, respectivamente. Porque ainda estamos realizando a classificação, a camada de saída de 10 dimensões corresponde ao número de classes de saída possíveis.

Chegar ao ponto em que você realmente entende o que está acontecendo dentro do LeNet pode dar um pouco de trabalho, mas espero que o seguinte *snippet* de código o convença que a implementação de tais modelos com estruturas modernas de *deep learning* é extremamente simples. Precisamos apenas instanciar um bloco `Sequential` e encadear as camadas apropriadas.

```
import torch
from torch import nn
from d2l import torch as d2l


class Reshape(torch.nn.Module):
    def forward(self, x):
        return x.view(-1, 1, 28, 28)

net = torch.nn.Sequential(
    Reshape(),
    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Tomamos uma pequena liberdade com o modelo original, removendo a ativação gaussiana na camada final. Fora isso, esta rede corresponde a arquitetura LeNet-5 original.

Ao passar por um canal único (preto e branco) 28×28 imagem através da rede e imprimir a forma de saída em cada camada, podemos inspecionar o modelo para ter certeza que suas operações se alinham com o que esperamos de Fig. 6.6.2.

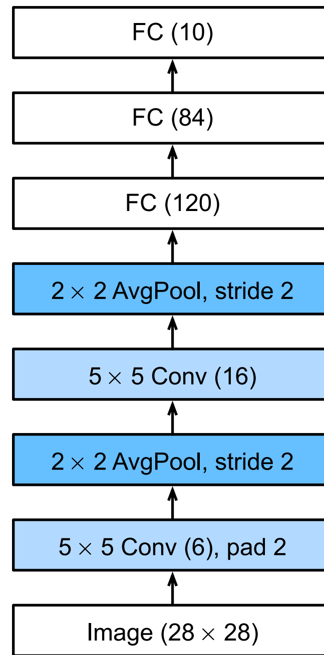


Fig. 6.6.2: Compressed notation for LeNet-5.

```
X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape: \t',X.shape)
```

```
Reshape output shape:        torch.Size([1, 1, 28, 28])
Conv2d output shape:         torch.Size([1, 6, 28, 28])
Sigmoid output shape:        torch.Size([1, 6, 28, 28])
AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
Conv2d output shape:         torch.Size([1, 16, 10, 10])
Sigmoid output shape:        torch.Size([1, 16, 10, 10])
AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
Flatten output shape:        torch.Size([1, 400])
Linear output shape:         torch.Size([1, 120])
Sigmoid output shape:        torch.Size([1, 120])
Linear output shape:         torch.Size([1, 84])
Sigmoid output shape:        torch.Size([1, 84])
Linear output shape:         torch.Size([1, 10])
```

Observe que a altura e largura da representação em cada camada ao longo do bloco convolucional é reduzido (em comparação com a camada anterior). A primeira camada convolucional usa 2 pixels de preenchimento para compensar a redução de altura e largura que de outra forma resultaria do uso de um *kernel* 5×5. Em contraste, a segunda camada convolucional dispensa o preenchimento, e, portanto, a altura e a largura são reduzidas em 4 pixels. Conforme subimos na pilha de camadas, o número de canais aumenta camada sobre camada de 1 na entrada a 6 após a primeira camada convolucional e 16 após a segunda camada convolucional. No entanto, cada camada de *pooling* divide a altura e a largura pela metade. Finalmente, cada camada totalmente conectada reduz a dimensionalidade, finalmente emitindo uma saída cuja dimensão corresponde ao número de classes.

6.6.2 Trainamento

Agora que implementamos o modelo, vamos fazer um experimento para ver como o LeNet se sai no Fashion-MNIST.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

Embora as CNNs tenham menos parâmetros, eles ainda podem ser mais caros para computar do que MLPs igualmente profundas porque cada parâmetro participa de muitas multiplicações a mais. Se você tiver acesso a uma GPU, este pode ser um bom momento para colocá-la em ação para acelerar o treinamento.

Para avaliação, precisamos fazer uma pequena modificação para a função `evaluate_accuracy` que descrevemos em Section 3.6. Uma vez que o conjunto de dados completo está na memória principal, precisamos copiá-lo para a memória da GPU antes que o modelo use a GPU para calcular com o conjunto de dados.

```
def evaluate_accuracy_gpu(net, data_iter, device=None): #@save
    """Compute the accuracy for a model on a dataset using a GPU."""
    if isinstance(net, torch.nn.Module):
        net.eval()  # Set the model to evaluation mode
        if not device:
            device = next(iter(net.parameters())).device
    # No. of correct predictions, no. of predictions
    metric = d2l.Accumulator(2)
    for X, y in data_iter:
        if isinstance(X, list):
            # Required for BERT Fine-tuning (to be covered later)
            X = [x.to(device) for x in X]
        else:
            X = X.to(device)
        y = y.to(device)
        metric.add(d2l.accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

Também precisamos atualizar nossa função de treinamento para lidar com GPUs. Ao contrário do `train_epoch_ch3` definido em Section 3.6, agora precisamos mover cada *minibatch* de dados para o nosso dispositivo designado (esperançosamente, a GPU) antes de fazer as propagações para frente e para trás.

A função de treinamento `train_ch6` também é semelhante para `train_ch3` definido em Section 3.6. Como iremos implementar redes com muitas camadas daqui para frente, contaremos principalmente com APIs de alto nível. A função de treinamento a seguir pressupõe um modelo criado a partir de APIs de alto nível como entrada e é otimizado em conformidade. Inicializamos os parâmetros do modelo no dispositivo indicado pelo argumento `device`, usando a inicialização do Xavier conforme apresentado em Section 4.8.2. Assim como com MLPs, nossa função de perda é entropia cruzada, e o minimizamos por meio da descida gradiente estocástica de *minibatch*. Como cada época leva dezenas de segundos para ser executada, visualizamos a perda de treinamento com mais frequência.

```
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
```

(continues on next page)

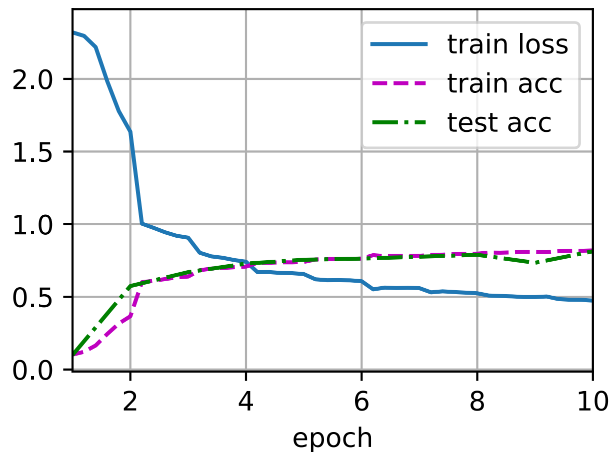
(continued from previous page)

```
    """Train a model with a GPU (defined in Chapter 6)."""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    print('training on', device)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

Agora vamos treinar e avaliar o modelo LeNet-5.

```
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.474, train acc 0.820, test acc 0.814
85084.5 examples/sec on cuda:0
```



6.6.3 Resumo

- A CNN é uma rede que emprega camadas convolucionais.
- Em uma CNN, intercalamos convoluções, não linearidades e (frequentemente) operações de agrupamento.
- Em uma CNN, as camadas convolucionais são normalmente organizadas de forma que diminuam gradualmente a resolução espacial das representações, enquanto aumentam o número de canais.
- Em CNNs tradicionais, as representações codificadas pelos blocos convolucionais são processadas por uma ou mais camadas totalmente conectadas antes de emitir a saída.
- LeNet foi indiscutivelmente a primeira implantação bem-sucedida de tal rede.

6.6.4 Exercícios

1. Substitua o *pooling* médio pelo *pooling* máximo. O que acontece?
2. Tente construir uma rede mais complexa baseada em LeNet para melhorar sua precisão.
    1. Ajuste o tamanho da janela de convolução.
    2. Ajuste o número de canais de saída.
    3. Ajuste a função de ativação (por exemplo, ReLU).
    4. Ajuste o número de camadas de convolução.
    5. Ajuste o número de camadas totalmente conectadas.
    6. Ajuste as taxas de aprendizagem e outros detalhes de treinamento (por exemplo, inicialização e número de épocas).
3. Experimente a rede aprimorada no conjunto de dados MNIST original.
4. Exibir as ativações da primeira e da segunda camada do LeNet para diferentes entradas (por exemplo, suéteres e casacos).

Discussions[71]

[71] https://discuss.d2l.ai/t/74

7 | Modern Convolutional Neural Networks

Agora que entendemos o básico de conectar CNNs, vamos levá-lo por um tour pelas arquiteturas modernas da CNN. Neste capítulo, cada seção corresponde a uma arquitetura significativa da CNN que foi em algum ponto (ou atualmente) o modelo básico sobre o qual muitos projetos de pesquisa e sistemas implantados foram construídos. Cada uma dessas redes foi brevemente uma arquitetura dominante e muitos foram vencedores ou segundos classificados na competição ImageNet, que tem servido como um barômetro do progresso na aprendizagem supervisionada em visão computacional desde 2010.

Esses modelos incluem AlexNet, a primeira rede em grande escala implantada a vencer os métodos convencionais de visão por computador em um desafio de visão em grande escala; a rede VGG, que faz uso de vários blocos repetidos de elementos; a rede na rede (NiN) que convolve redes neurais inteiras com patch por meio de entradas; GoogLeNet, que usa redes com concatenações paralelas; redes residuais (ResNet), que continuam sendo as mais populares arquitetura pronta para uso em visão computacional; e redes densamente conectadas (DenseNet), que são caros para calcular, mas estabeleceram alguns benchmarks recentes.

Embora a ideia de redes neurais *profundas* seja bastante simples (empilhar um monte de camadas), o desempenho pode variar muito entre as arquiteturas e as opções de hiperparâmetros. As redes neurais descritas neste capítulo são o produto da intuição, alguns insights matemáticos, e muita tentativa e erro. Apresentamos esses modelos em ordem cronológica, em parte para transmitir um sentido da história para que você possa formar suas próprias intuições sobre para onde o campo está indo e talvez desenvolva suas próprias arquiteturas. Por exemplo, a normalização em lote e as conexões residuais descritas neste capítulo ofereceram duas ideias populares para treinar e projetar modelos profundos.

7.1 Redes Neurais Convolucionais Profundas (AlexNet)

Embora as CNNs fossem bem conhecidas nas comunidades de visão computacional e aprendizado de máquina após a introdução do LeNet, eles não dominaram imediatamente o campo. Embora LeNet tenha alcançado bons resultados em pequenos conjuntos de dados iniciais, o desempenho e a viabilidade de treinamento de CNNs em conjuntos de dados maiores e mais realistas ainda não foram estabelecidos. Na verdade, durante grande parte do tempo intermediário entre o início da década de 1990 e os resultados do divisor de águas de 2012, redes neurais muitas vezes eram superadas por outros métodos de aprendizado de máquina, como máquinas de vetores de suporte.

Para a visão computacional, essa comparação talvez não seja justa. Isso embora as entradas para redes convolucionais consistam em valores de pixel brutos ou levemente processados (por exemplo, pela centralização), os profissionais nunca alimentariam pixels brutos em modelos tradi-

cionais. Em vez disso, pipelines típicos de visão computacional consistiam em pipelines de extração de recursos de engenharia manual. Em vez de *aprender os recursos*, os recursos foram *criados*. A maior parte do progresso veio de ter ideias mais inteligentes para recursos, e o algoritmo de aprendizagem foi frequentemente relegado a uma reflexão tardia.

Embora alguns aceleradores de rede neural estivessem disponíveis na década de 1990, eles ainda não eram suficientemente poderosos para fazer CNNs multicanal e multicamadas profundas com um grande número de parâmetros. Além disso, os conjuntos de dados ainda eram relativamente pequenos. Somados a esses obstáculos, truques-chave para treinar redes neurais incluindo heurísticas de inicialização de parâmetros, variantes inteligentes de descida gradiente estocástica, funções de ativação não esmagadoras, e ainda faltavam técnicas de regularização eficazes.

Assim, em vez de treinar sistemas *ponta a ponta* (pixel para classificação), pipelines clássicos pareciam mais com isto:

1. Obtenha um conjunto de dados interessante. No início, esses conjuntos de dados exigiam sensores caros (na época, as imagens de 1 megapixel eram de última geração).
2. Pré-processe o conjunto de dados com recursos feitos à mão com base em algum conhecimento de ótica, geometria, outras ferramentas analíticas e, ocasionalmente, nas descobertas fortuitas de alunos de pós-graduação sortudos.
3. Alimente os dados por meio de um conjunto padrão de extratores de recursos, como o SIFT (transformação de recurso invariante de escala) (Lowe, 2004), o SURF (recursos robustos acelerados) [Bay.Tuytelaars.Van- Gool.2006], ou qualquer outro duto ajustado manualmente.
4. 1.espeje as representações resultantes em seu classificador favorito, provavelmente um modelo linear ou método de kernel, para treinar um classificador.

Se você conversasse com pesquisadores de aprendizado de máquina, eles acreditavam que o aprendizado de máquina era importante e bonito. Teorias elegantes provaram as propriedades de vários classificadores. O campo do aprendizado de máquina era próspero, rigoroso e eminentemente útil. No entanto, se você falou com um pesquisador de visão computacional, você ouviria uma história muito diferente. A verdade suja do reconhecimento de imagem, eles diriam a você, é que os recursos, e não os algoritmos de aprendizagem, impulsionaram o progresso. Pesquisadores de visão computacional acreditavam com razão que um conjunto de dados ligeiramente maior ou mais limpo ou um pipeline de extração de recursos ligeiramente melhorado importava muito mais para a precisão final do que qualquer algoritmo de aprendizado.

7.1.1 Representaçao do Aprendizado

Outra forma de definir o estado de coisas é que a parte mais importante do pipeline foi a representação. E até 2012 a representação era calculada mecanicamente. Na verdade, desenvolver um novo conjunto de funções de recursos, melhorar os resultados e escrever o método era um gênero de papel proeminente. SIFT (Lowe, 2004), SURF (Bay et al., 2006), HOG (histogramas de gradiente orientado) :cite: `Dalal.Triggs.2005`, pacotes de palavras visuais[72] e extratores de recursos semelhantes são os mais usados.

Outro grupo de pesquisadores, incluindo Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari e Juergen Schmidhuber, tinha planos diferentes. Eles acreditavam que as próprias características deveriam ser aprendidas. Além disso, eles acreditavam que era razoavelmente complexo, os recursos devem ser compostos hierarquicamente com várias camadas aprendidas em conjunto, cada uma com parâmetros aprendíveis. No caso de uma imagem, as camadas

[72] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

mais baixas podem vir para detectar bordas, cores e texturas. De fato, Alex Krizhevsky, Ilya Sutskever e Geoff Hinton propôs uma nova variante de uma CNN, *AlexNet*, que obteve excelente desempenho no desafio ImageNet de 2012. AlexNet foi nomeado após Alex Krizhevsky, o primeiro autor do inovador artigo de classificação ImageNet (Krizhevsky et al., 2012).

Curiosamente, nas camadas mais baixas da rede, o modelo aprendeu extratores de recursos que se assemelhavam a alguns filtros tradicionais. Fig. 7.1.1 é reproduzido do artigo AlexNet (Krizhevsky et al., 2012) e descreve descritores de imagem de nível inferior.

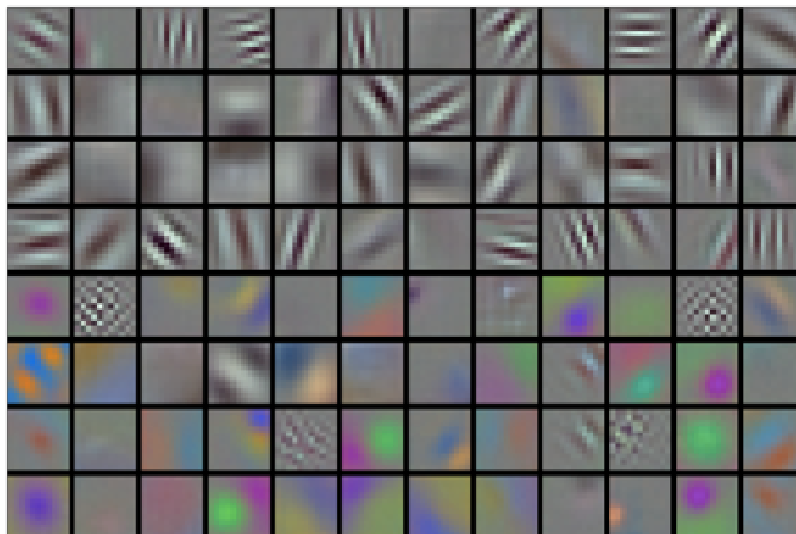


Fig. 7.1.1: Filtros de imagem aprendidos pela primeira camada do AlexNet.

As camadas superiores da rede podem se basear nessas representações para representar estruturas maiores, como olhos, narizes, folhas de grama e assim por diante. Mesmo camadas mais altas podem representar objetos inteiros como pessoas, aviões, cães ou frisbees. Em última análise, o estado oculto final aprende uma representação compacta da imagem que resume seu conteúdo de forma que os dados pertencentes a diferentes categorias possam ser facilmente separados.

Enquanto a inovação definitiva para CNNs de várias camadas veio em 2012, um grupo central de pesquisadores se dedicou a esta ideia, tentando aprender representações hierárquicas de dados visuais por muitos anos. O grande avanço em 2012 pode ser atribuído a dois fatores principais.

Ingrediente Faltante: Dados

Modelos profundos com muitas camadas requerem grandes quantidades de dados a fim de entrar no regime onde superam significativamente os métodos tradicionais com base em otimizações convexas (por exemplo, métodos lineares e de kernel). No entanto, dada a capacidade limitada de armazenamento dos computadores, a despesa relativa de sensores, e os orçamentos de pesquisa comparativamente mais apertados na década de 1990, a maioria das pesquisas baseou-se em pequenos conjuntos de dados. Numerosos artigos abordaram a coleção de conjuntos de dados da UCI, muitos dos quais continham apenas centenas ou (alguns) milhares de imagens capturado em configurações não naturais com baixa resolução.

Em 2009, o conjunto de dados ImageNet foi lançado, desafiando os pesquisadores a aprender modelos a partir de 1 milhão de exemplos, 1000 categorias distintas de objetos com 1000 dados em cada. Os pesquisadores, liderados por Fei-Fei Li, que apresentou este conjunto de dados aproveitou a

Pesquisa de imagens do Google para pré-filtrar grandes conjuntos de candidatos para cada categoria e empregado o pipeline de crowdsourcing Amazon Mechanical Turk para confirmar para cada imagem se pertencia à categoria associada. Essa escala não tinha precedentes. A competição associada, apelidada de Desafio ImageNet impulsionou a pesquisa sobre visão computacional e aprendizado de máquina, desafiando os pesquisadores a identificar quais modelos tiveram melhor desempenho em uma escala maior do que os acadêmicos haviam considerado anteriormente.

Ingrediente Faltante: *Hardware*

Modelos de aprendizado profundo são consumidores vorazes de ciclos de computação. O treinamento pode levar centenas de épocas, e cada iteração requer a passagem de dados por muitas camadas de operações de álgebra linear de alto custo computacional. Esta é uma das principais razões pelas quais, na década de 1990 e no início de 2000, algoritmos simples baseados em algoritmos otimizados de objetivas convexas mais eficientes foram preferidas.

Unidades de processamento gráfico (GPUs) provaram ser uma virada de jogo para tornar o aprendizado profundo viável. Esses chips há muito foram desenvolvidos para acelerar processamento gráfico para beneficiar os jogos de computador. Em particular, eles foram otimizados para produtos de vetor de matriz de alta capacidade 4×4, que são necessários para muitas tarefas de computação gráfica. Felizmente, essa matemática é muito semelhante ao necessário para calcular camadas convolucionais. Naquela época, a NVIDIA e a ATI começaram a otimizar GPUs para operações gerais de computação, indo tão longe a ponto de comercializá-los como *GPUs de uso geral* (GPGPU).

Para fornecer alguma intuição, considere os núcleos de um microprocessador moderno (CPU). Cada um dos núcleos é bastante poderoso rodando em uma alta frequência de clock e exibindo grandes caches (até vários megabytes de L3). Cada núcleo é adequado para executar uma ampla gama de instruções, com preditores de ramificação, um pipeline profundo e outras técnicas que permitem executar uma grande variedade de programas. Essa aparente força, no entanto, é também seu calcanhar de Aquiles: núcleos de uso geral são muito caros para construir. Eles exigem muita área de chip, uma estrutura de suporte sofisticada (interfaces de memória, lógica de cache entre os núcleos, interconexões de alta velocidade e assim por diante), e são comparativamente ruins em qualquer tarefa. Laptops modernos têm até 4 núcleos, e até mesmo servidores de última geração raramente excedem 64 núcleos, simplesmente porque não é rentável.

Em comparação, as GPUs consistem em $100 \sim 1000$ pequenos elementos de processamento (os detalhes diferem um pouco entre NVIDIA, ATI, ARM e outros fornecedores de chips), frequentemente agrupados em grupos maiores (a NVIDIA os chama de warps). Embora cada núcleo seja relativamente fraco, às vezes até rodando em frequência de clock abaixo de 1 GHz, é o número total de tais núcleos que torna as GPUs ordens de magnitude mais rápidas do que as CPUs. Por exemplo, a recente geração Volta da NVIDIA oferece até 120 TFlops por chip para instruções especializadas (e até 24 TFlops para aqueles de uso geral), enquanto o desempenho de ponto flutuante de CPUs não excedeu 1 TFlop até o momento. A razão pela qual isso é possível é bastante simples: primeiro, o consumo de energia tende a crescer *quadraticamente* com a frequência do clock. Portanto, para o orçamento de energia de um núcleo da CPU que funciona 4 vezes mais rápido (um número típico), você pode usar 16 núcleos de GPU por $1/4$ a velocidade, que rende $16 \times 1/4 = 4$ vezes o desempenho. Além disso, os núcleos da GPU são muito mais simples (na verdade, por muito tempo eles nem mesmo foram *capazes* para executar código de uso geral), o que os torna mais eficientes em termos de energia. Por último, muitas operações de aprendizado profundo exigem alta largura de banda de memória. Novamente, as GPUs brilham aqui com barramentos que são pelo menos 10 vezes mais largos que muitas CPUs.

De volta a 2012. Um grande avanço veio quando Alex Krizhevsky e Ilya Sutskever implementou uma CNN profunda que pode ser executado em hardware GPU. Eles perceberam que os gargalos computacionais nas CNNs, convoluções e multiplicações de matrizes, são todas as operações que podem ser paralelizadas no hardware. Usando dois NVIDIA GTX 580s com 3 GB de memória, eles implementaram convoluções rápidas. O código cuda-convnet[73] foi bom o suficiente por vários anos, era o padrão da indústria e alimentou os primeiros anos do boom do aprendizado profundo.

7.1.2 AlexNet

AlexNet, que empregava uma CNN de 8 camadas, venceu o Desafio de Reconhecimento Visual em Grande Escala ImageNet 2012 por uma margem fenomenalmente grande. Esta rede mostrou, pela primeira vez, que os recursos obtidos pelo aprendizado podem transcender recursos projetados manualmente, quebrando o paradigma anterior em visão computacional.

As arquiteturas de AlexNet e LeNet são muito semelhantes, como Fig. 7.1.2 ilustra. Observe que fornecemos uma versão ligeiramente simplificada do AlexNet removendo algumas das peculiaridades de design que eram necessárias em 2012 para fazer o modelo caber em duas pequenas GPUs.

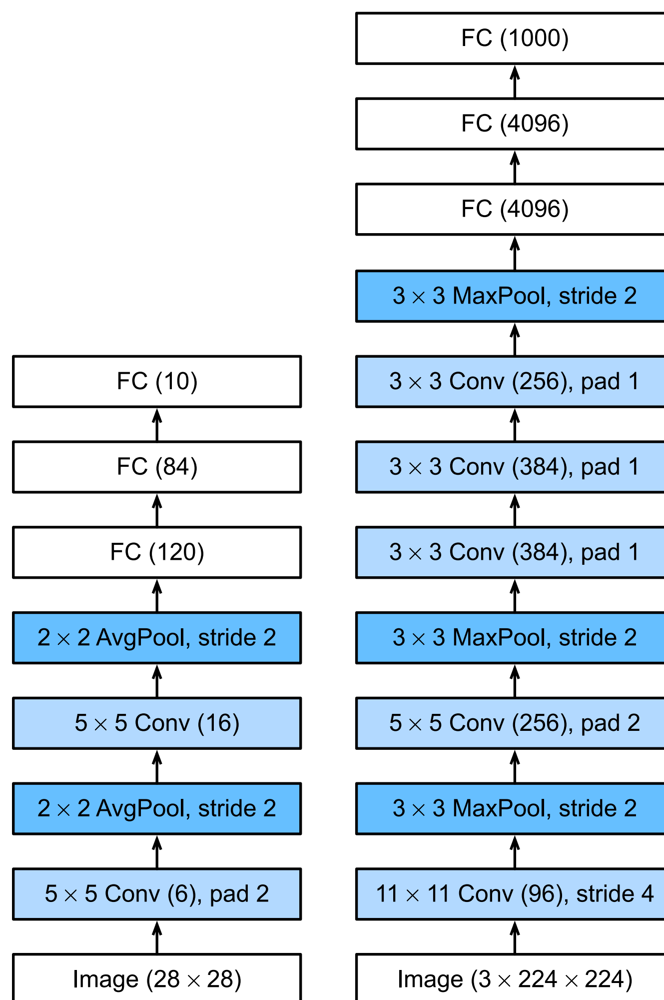


Fig. 7.1.2: De LeNet (esquerda) para AlexNet (direita).

As filosofias de design de AlexNet e LeNet são muito semelhantes, mas também existem diferenças

[73] https://code.google.com/archive/p/cuda-convnet/

significativas. Primeiro, o AlexNet é muito mais profundo do que o comparativamente pequeno LeNet5. AlexNet consiste em oito camadas: cinco camadas convolucionais, duas camadas ocultas totalmente conectadas e uma camada de saída totalmente conectada. Em segundo lugar, AlexNet usou o ReLU em vez do sigmóide como sua função de ativação. Vamos nos aprofundar nos detalhes abaixo.

Arquitetura

Na primeira camada do AlexNet, a forma da janela de convolução é 11×11. Uma vez que a maioria das imagens no ImageNet são mais de dez vezes maiores e mais largas do que as imagens MNIST, objetos em dados ImageNet tendem a ocupar mais pixels. Consequentemente, uma janela de convolução maior é necessária para capturar o objeto. A forma da janela de convolução na segunda camada é reduzido para 5×5, seguido por 3×3. Além disso, após a primeira, segunda e quinta camadas convolucionais, a rede adiciona camadas de pooling máximas com um formato de janela de 3×3 e uma distância de 2. Além disso, o AlexNet tem dez vezes mais canais de convolução do que o LeNet.

Após a última camada convolucional, existem duas camadas totalmente conectadas com 4096 saídas. Essas duas enormes camadas totalmente conectadas produzem parâmetros de modelo de quase 1 GB. Devido à memória limitada nas primeiras GPUs, o AlexNet original usava um design de fluxo de dados duplo, para que cada uma de suas duas GPUs pudesse ser responsável para armazenar e computar apenas sua metade do modelo. Felizmente, a memória da GPU é comparativamente abundante agora, então raramente precisamos separar os modelos das GPUs hoje em dia (nossa versão do modelo AlexNet se desvia do artigo original neste aspecto).

Função de Ativação

Além disso, AlexNet mudou a função de ativação sigmóide para uma função de ativação ReLU mais simples. Por um lado, o cálculo da função de ativação ReLU é mais simples. Por exemplo, ele não tem a operação de exponenciação encontrada na função de ativação sigmóide. Por outro lado, a função de ativação ReLU torna o treinamento do modelo mais fácil ao usar diferentes métodos de inicialização de parâmetro. Isso ocorre porque, quando a saída da função de ativação sigmóide está muito próxima de 0 ou 1, o gradiente dessas regiões é quase 0, de modo que a retropropagação não pode continuar a atualizar alguns dos parâmetros do modelo. Em contraste, o gradiente da função de ativação ReLU no intervalo positivo é sempre 1. Portanto, se os parâmetros do modelo não forem inicializados corretamente, a função sigmóide pode obter um gradiente de quase 0 no intervalo positivo, de modo que o modelo não pode ser efetivamente treinados.

Controle de Capacidade e Pré-processamento

AlexNet controla a complexidade do modelo da camada totalmente conectada por *dropout* (Section 4.6), enquanto o LeNet usa apenas redução de peso. Para aumentar ainda mais os dados, o loop de treinamento do AlexNet adicionou uma grande quantidade de aumento de imagem, como inversão, recorte e alterações de cor. Isso torna o modelo mais robusto e o tamanho de amostra maior reduz efetivamente o sobreajuste. Discutiremos o aumento de dados em maiores detalhes em Section 13.1.

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    # Aqui, usamos uma janela maior de 11 x 11 para capturar objetos. Ao mesmo tempo,
    # usamos um passo de 4 para reduzir significativamente a altura e a largura
    # da saída. Aqui, o número de canais de saída
    # é muito maior do que no LeNet
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Torne a janela de convolução menor, defina o preenchimento para 2
    # para altura e largura consistentes na entrada e saída,
    # e aumente o número de canais de saída
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Use três camadas convolucionais sucessivas e uma janela de convolução menor.
    # Exceto para a camada convolucional final, o número de canais de saída
    # é aumentado ainda mais. Camadas de pooling não são usadas para reduzir
    # a altura e largura de entrada após as duas primeiras camadas convolucionais
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Flatten(),
    # Aqui, o número de saídas da camada totalmente conectada é várias
    # vezes maior do que no LeNet.
    # Use a camada de eliminação para mitigar overfitting
    nn.Linear(6400, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    # Camada de saída. Como estamos usando o Fashion-MNIST,
    # o número de classes é 10, em vez de 1000 como no papel
    nn.Linear(4096, 10))
```

Construímos um exemplo de dados de canal único com altura e largura de 224 para observar a forma de saída de cada camada. Ele corresponde à arquitetura AlexNet em Fig. 7.1.2.

```
X = torch.randn(1, 1, 224, 224)
for layer in net:
    X=layer(X)
    print(layer.__class__.__name__,'output shape:\t',X.shape)
```

```
Conv2d output shape:         torch.Size([1, 96, 54, 54])
ReLU output shape:   torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Conv2d output shape:         torch.Size([1, 256, 26, 26])
ReLU output shape:   torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
```

(continues on next page)

(continued from previous page)

```
Conv2d output shape:         torch.Size([1, 256, 12, 12])
ReLU output shape:   torch.Size([1, 256, 12, 12])
MaxPool2d output shape:      torch.Size([1, 256, 5, 5])
Flatten output shape:        torch.Size([1, 6400])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

7.1.3 Lendo o Dataset

Embora AlexNet seja treinado em ImageNet no jornal, usamos Fashion-MNIST aqui já que treinar um modelo ImageNet para convergência pode levar horas ou dias mesmo em uma GPU moderna. Um dos problemas de aplicar AlexNet diretamente no Fashion-MNIST é que suas imagens têm resolução inferior (28×28 pixels) do que imagens ImageNet. Para fazer as coisas funcionarem, aumentamos a amostra para 224×224 (geralmente não é uma prática inteligente, mas fazemos isso aqui para sermos fiéis à arquitetura AlexNet). Realizamos esse redimensionamento com o argumento `resize` na função`d2l.load_data_fashion_mnist`.

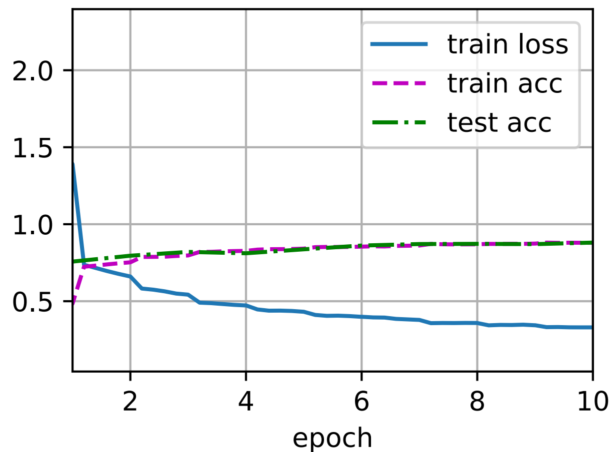
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

7.1.4 Treinamento

Agora, podemos começar a treinar AlexNet. Comparado com LeNet em Section 6.6, a principal mudança aqui é o uso de uma taxa de aprendizado menor e um treinamento muito mais lento devido à rede mais ampla e profunda, a resolução de imagem mais alta e as convoluções mais caras.

```
lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.330, train acc 0.879, test acc 0.881
4138.2 examples/sec on cuda:0
```



7.1.5 Sumário

- AlexNet tem uma estrutura semelhante à do LeNet, mas usa mais camadas convolucionais e um espaço de parâmetro maior para caber no conjunto de dados ImageNet em grande escala.
- Hoje o AlexNet foi superado por arquiteturas muito mais eficazes, mas é um passo importante das redes superficiais para as profundas que são usadas hoje em dia.
- Embora pareça que existam apenas mais algumas linhas na implementação do AlexNet do que no LeNet, a comunidade acadêmica levou muitos anos para abraçar essa mudança conceitual e tirar proveito de seus excelentes resultados experimentais. Isso também se deveu à falta de ferramentas computacionais eficientes.
- Desistência, ReLU e pré-processamento foram as outras etapas importantes para alcançar um desempenho excelente em tarefas de visão computacional.

7.1.6 Exercício

1. Experimente aumentar o número de épocas. Comparado com o LeNet, como os resultados são diferentes? Porque?
2. AlexNet pode ser muito complexo para o conjunto de dados Fashion-MNIST.
    1. Tente simplificar o modelo para tornar o treinamento mais rápido, garantindo que a precisão não caia significativamente.
    2. Projete um modelo melhor que funcione diretamente em imagens de 28×28.
3. Modifique o tamanho do lote e observe as mudanças na precisão e na memória da GPU.
4. Analise o desempenho computacional do AlexNet.
    1. Qual é a parte dominante para a pegada de memória do AlexNet?
    2. Qual é a parte dominante para computação no AlexNet?
    3. E quanto à largura de banda da memória ao calcular os resultados?
5. Aplique dropout e ReLU ao LeNet-5. Isso melhora? Que tal pré-processamento?

Discussão[74]

7.2 Redes Usando Blocos (VGG)

Enquanto AlexNet ofereceu evidências empíricas de que CNNs profundas pode alcançar bons resultados, não forneceu um modelo geral para orientar os pesquisadores subsequentes na concepção de novas redes. Nas seções a seguir, apresentaremos vários conceitos heurísticos comumente usado para projetar redes profundas.

O progresso neste campo reflete aquele no design de chips onde os engenheiros deixaram de colocar transistores para elementos lógicos para blocos lógicos. Da mesma forma, o projeto de arquiteturas de rede neural tornou-se progressivamente mais abstrato, com pesquisadores deixando de pensar em termos de neurônios individuais para camadas inteiras, e agora para blocos, repetindo padrões de camadas.

A ideia de usar blocos surgiu pela primeira vez a partir do Grupo de Geometria Visual[75] (VGG) na Universidade de Oxford, em sua rede de mesmo nome *VGG*. É fácil implementar essas estruturas repetidas no código com qualquer estrutura moderna de aprendizado profundo usando loops e sub-rotinas.

7.2.1 VGG Blocks

O bloco de construção básico das CNNs clássicas é uma sequência do seguinte: (i) uma camada convolucional com preenchimento para manter a resolução, (ii) uma não linearidade, como um ReLU, (iii) uma camada de pooling tal como uma camada de pooling máxima. Um bloco VGG consiste em uma sequência de camadas convolucionais, seguido por uma camada de *pooling* máxima para *downsampling* espacial. No artigo VGG original (Simonyan & Zisserman, 2014), Os autores convoluções empregadas com 3×3 kernels com preenchimento de 1 (mantendo a altura e largura) e 2×2 *pool* máximo com passo de 2 (reduzindo pela metade a resolução após cada bloco). No código abaixo, definimos uma função chamada `vgg_block` para implementar um bloco VGG.

: begin_tab: `mxnet`, `tensorflow` A função leva dois argumentos correspondendo ao número de camadas convolucionais `num_convs` e o número de canais de saída `num_channels`. : end_tab:

: begin_tab: `pytorch` A função leva três argumentos correspondentes ao número de camadas convolucionais `num_convs`, o número de canais de entrada `in_channels` e o número de canais de saída `out_channels`. : end_tab:

```
import torch
from torch import nn
from d2l import torch as d2l


def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(in_channels, out_channels,
                                kernel_size=3, padding=1))
        layers.append(nn.ReLU())
```

(continues on next page)

[74] https://discuss.d2l.ai/t/76
[75] http://www.robots.ox.ac.uk/~vgg/

(continued from previous page)

```
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

7.2.2 Camadas VGG

Como AlexNet e LeNet, a rede VGG pode ser dividida em duas partes: o primeiro consistindo principalmente de camadas convolucionais e de *pooling* e a segunda consistindo em camadas totalmente conectadas. Isso é descrito em Fig. 7.2.1.

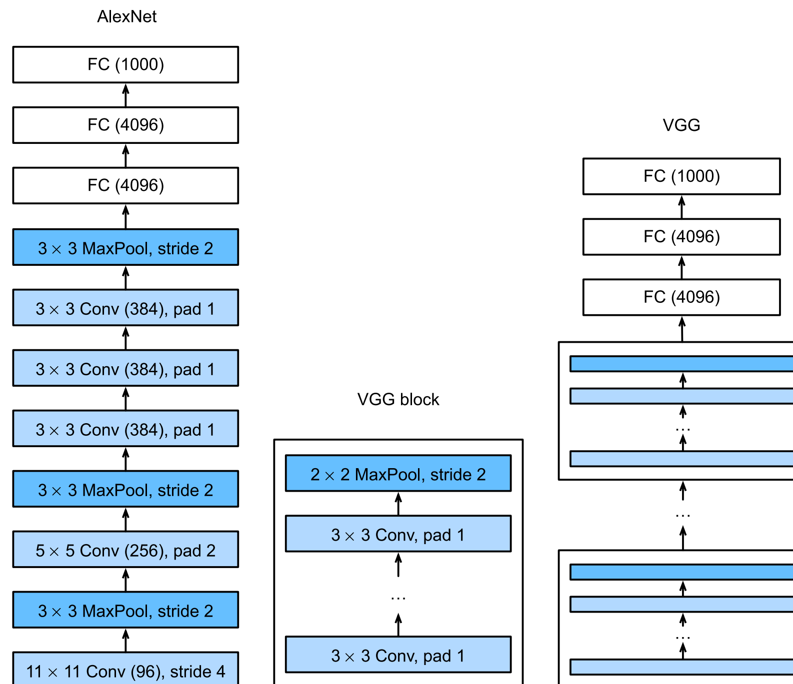


Fig. 7.2.1: De AlexNet a VGG que é projetado a partir de blocos de construção.

A parte convolucional da rede conecta vários blocos VGG de Fig. 7.2.1 (também definido na funçãovgg_block) em sucessão. A seguinte variável `conv_arch` consiste em uma lista de tuplas (uma por bloco), onde cada um contém dois valores: o número de camadas convolucionais e o número de canais de saída, quais são precisamente os argumentos necessários para chamar a função `vgg_block`. A parte totalmente conectada da rede VGG é idêntica à coberta no AlexNet.

A rede VGG original tinha 5 blocos convolucionais, entre os quais os dois primeiros têm uma camada convolucional cada e os três últimos contêm duas camadas convolucionais cada. O primeiro bloco tem 64 canais de saída e cada bloco subsequente dobra o número de canais de saída, até que esse número chegue a 512. Uma vez que esta rede usa 8 camadas convolucionais e 3 camadas totalmente conectadas, geralmente chamado de VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

O código a seguir implementa VGG-11. Esta é uma simples questão de executar um loop for sobre `conv_arch`.

```
def vgg(conv_arch):
    conv_blks = []
    in_channels = 1
    # A parte convolucional
    for (num_convs, out_channels) in conv_arch:
        conv_blks.append(vgg_block(num_convs, in_channels, out_channels))
        in_channels = out_channels

    return nn.Sequential(
        *conv_blks, nn.Flatten(),
        # A parte totalmente conectada
        nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 10))

net = vgg(conv_arch)
```

A seguir, construiremos um exemplo de dados de canal único com altura e largura de 224 para observar a forma de saída de cada camada.

```
X = torch.randn(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.__class__.__name__,'output shape:\t',X.shape)
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:        torch.Size([1, 25088])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

Como você pode ver, dividimos a altura e a largura em cada bloco, finalmente alcançando uma altura e largura de 7 antes de achatar as representações para processamento pela parte totalmente conectada da rede.

7.2.3 Treinamento

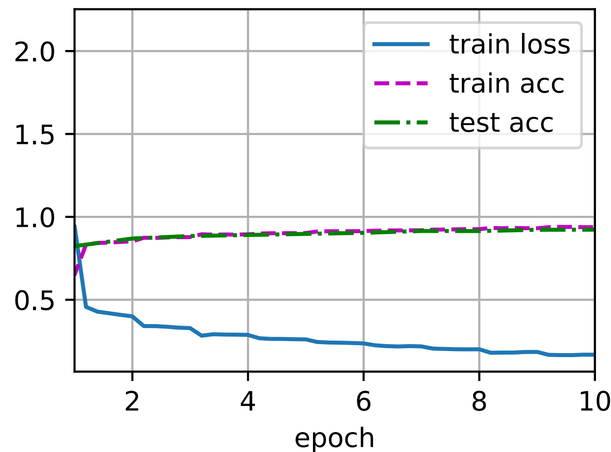
Como o VGG-11 é mais pesado em termos computacionais do que o AlexNet construímos uma rede com um número menor de canais. Isso é mais do que suficiente para o treinamento em Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Além de usar uma taxa de aprendizado um pouco maior, o processo de treinamento do modelo é semelhante ao do AlexNet em Section 7.1.

```
lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.169, train acc 0.938, test acc 0.923
2550.9 examples/sec on cuda:0
```



7.2.4 Sumário

- VGG-11 constrói uma rede usando blocos convolucionais reutilizáveis. Diferentes modelos de VGG podem ser definidos pelas diferenças no número de camadas convolucionais e canais de saída em cada bloco.
- O uso de blocos leva a representações muito compactas da definição da rede. Ele permite um projeto eficiente de redes complexas.
- Em seu artigo VGG, Simonyan e Ziserman experimentaram várias arquiteturas. Em particular, eles descobriram que várias camadas de convoluções profundas e estreitas (ou seja, 3×3) eram mais eficazes do que menos camadas de convoluções mais largas.

7.2.5 Exercícios

1. Ao imprimir as dimensões das camadas, vimos apenas 8 resultados, em vez de 11. Para onde foram as informações das 3 camadas restantes?
2. Comparado com o AlexNet, o VGG é muito mais lento em termos de computação e também precisa de mais memória GPU. Analise as razões disso.
3. Tente alterar a altura e a largura das imagens no Fashion-MNIST de 224 para 96. Que influência isso tem nos experimentos?
4. Consulte a Tabela 1 no artigo VGG (Simonyan & Zisserman, 2014) para construir outros modelos comuns, como VGG-16 ou VGG-19.

Discussions[76]

7.3 Network in Network (NiN)

LeNet, AlexNet e VGG compartilham um padrão de design comum: extrair recursos que exploram a estrutura *espacial* por meio de uma sequência de camadas de convolução e agrupamento e, em seguida, pós-processar as representações por meio de camadas totalmente conectadas. As melhorias no LeNet por AlexNet e VGG residem principalmente em como essas redes posteriores ampliam e aprofundam esses dois módulos. Alternativamente, pode-se imaginar o uso de camadas totalmente conectadas no início do processo. No entanto, um uso descuidado de camadas densas pode desistir da estrutura espacial da representação inteiramente, Os blocos *rede em rede* (*NiN*) oferecem uma alternativa. Eles foram propostos com base em uma visão muito simples: para usar um MLP nos canais para cada pixel separadamente (Lin et al., 2013).

7.3.1 Blocos NiN

Lembre-se de que as entradas e saídas das camadas convolucionais consistem em tensores quadridimensionais com eixos correspondendo ao exemplo, canal, altura e largura. Lembre-se também de que as entradas e saídas de camadas totalmente conectadas são tipicamente tensores bidimensionais correspondentes ao exemplo e ao recurso. A ideia por trás do NiN é aplicar uma camada totalmente conectada em cada localização de pixel (para cada altura e largura). Se amarrarmos os pesos em cada localização espacial, poderíamos pensar nisso como uma camada convolucional 1×1 (conforme descrito em Section 6.4) ou como uma camada totalmente conectada agindo de forma independente em cada localização de pixel. Outra maneira de ver isso é pensar em cada elemento na dimensão espacial (altura e largura) como equivalente a um exemplo e um canal como equivalente a um recurso.

Fig. 7.3.1 ilustra as principais diferenças estruturais entre VGG e NiN, e seus blocos. O bloco NiN consiste em uma camada convolucional seguido por duas camadas convolucionais 1×1 que atuam como camadas totalmente conectadas por pixel com ativações ReLU. A forma da janela de convolução da primeira camada é normalmente definida pelo usuário. As formas de janela subsequentes são fixadas em 1×1.

[76] https://discuss.d2l.ai/t/78

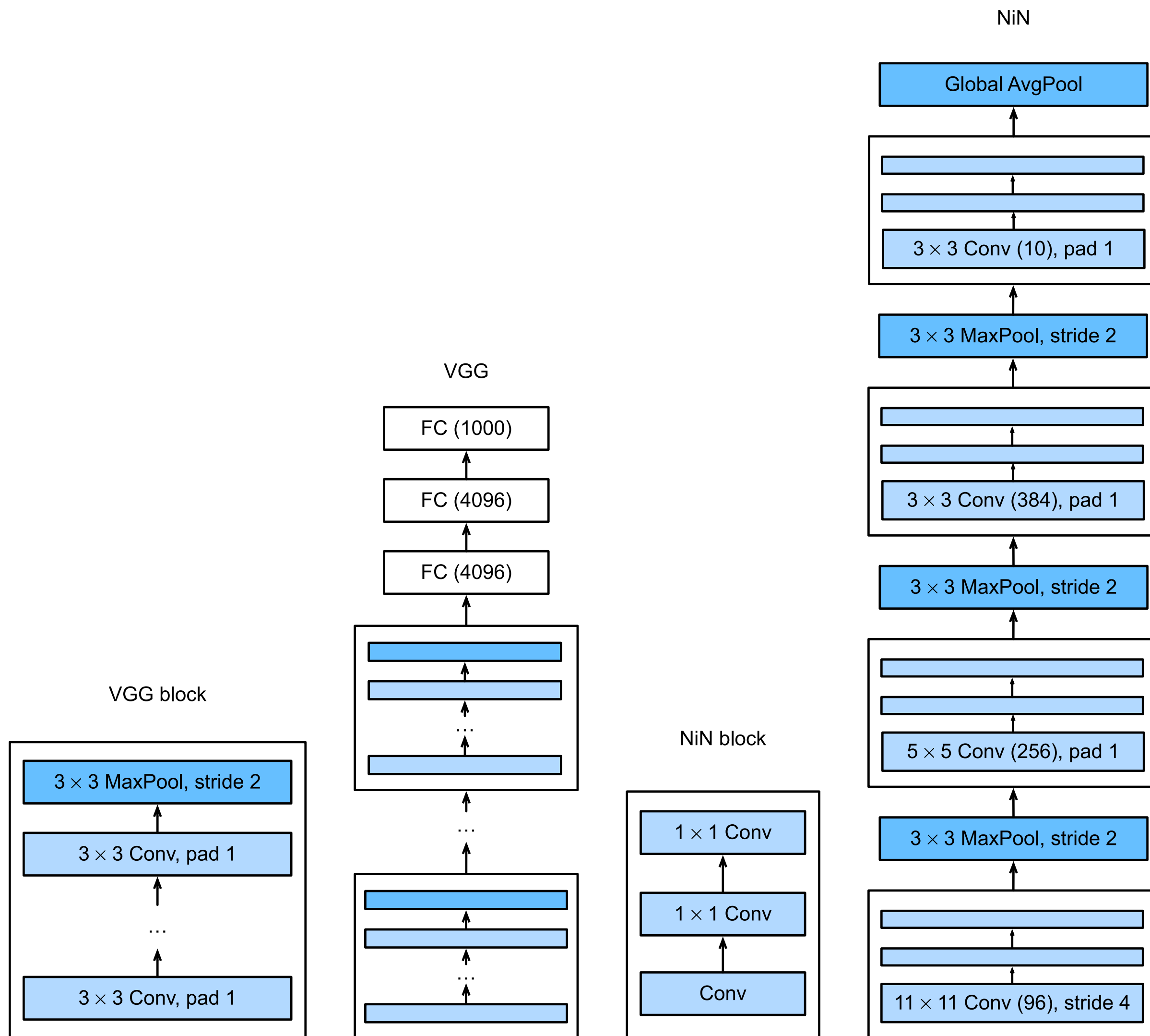


Fig. 7.3.1: Comparando arquiteturas de VGG e NiN, e seus blocos.

```
import torch
from torch import nn
from d2l import torch as d2l


def nin_block(in_channels, out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size, strides, padding),
        nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU())
```

7.3.2 Modelo NiN

A rede NiN original foi proposta logo após AlexNet e claramente tira alguma inspiração. NiN usa camadas convolucionais com formas de janela de 11×11, 5×5 e 3×3, e os números correspondentes de canais de saída são iguais aos do AlexNet. Cada bloco NiN é seguido por uma camada de *pooling* máxima com um passo de 2 e uma forma de janela de 3×3.

Uma diferença significativa entre NiN e AlexNet é que o NiN evita camadas totalmente conectadas. Em vez disso, NiN usa um bloco NiN com um número de canais de saída igual ao número de classes de rótulo, seguido por uma camada de *pooling* média global, produzindo um vetor de logits. Uma vantagem do design da NiN é que reduz o número de parâmetros de modelo necessários. No entanto, na prática, esse design às vezes requer aumento do tempo de treinamento do modelo.

```
net = nn.Sequential(
    nin_block(1, 96, kernel_size=11, strides=4, padding=0),
    nn.MaxPool2d(3, stride=2),
    nin_block(96, 256, kernel_size=5, strides=1, padding=2),
    nn.MaxPool2d(3, stride=2),
    nin_block(256, 384, kernel_size=3, strides=1, padding=1),
    nn.MaxPool2d(3, stride=2),
    nn.Dropout(0.5),
    # Há 3 classes de rótulos
    nin_block(384, 10, kernel_size=3, strides=1, padding=1),
    nn.AdaptiveAvgPool2d((1, 1)),
    # Transforme a saída quadridimensional em saída bidimensional com uma forma de (batch␣
↪size, 10)
    nn.Flatten())
```

Criamos um exemplo de dados para ver a forma de saída de cada bloco.

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

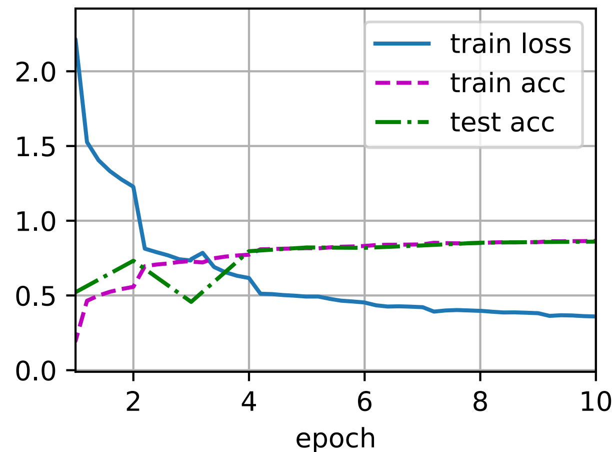
```
Sequential output shape:     torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Sequential output shape:     torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Sequential output shape:     torch.Size([1, 384, 12, 12])
MaxPool2d output shape:      torch.Size([1, 384, 5, 5])
Dropout output shape:        torch.Size([1, 384, 5, 5])
Sequential output shape:     torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d output shape:      torch.Size([1, 10, 1, 1])
Flatten output shape:        torch.Size([1, 10])
```

7.3.3 Treinamento

Como antes, usamos o Fashion-MNIST para treinar a modelo. O treinamento de NiN é semelhante ao de AlexNet e VGG.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.359, train acc 0.865, test acc 0.860
3201.6 examples/sec on cuda:0
```



7.3.4 Sumário

- NiN usa blocos que consistem em uma camada convolucional e várias camadas convolucionais 1×1. Isso pode ser usado dentro da pilha convolucional para permitir mais não linearidade por pixel.
- NiN remove as camadas totalmente conectadas e as substitui pelo agrupamento médio global (ou seja, somando todos os locais) depois de reduzir o número de canais para o número desejado de saídas (por exemplo, 10 para Fashion-MNIST).
- A remoção das camadas totalmente conectadas reduz o ajuste excessivo. NiN tem muito menos parâmetros.
- O design NiN influenciou muitos designs subsequentes da CNN.

7.3.5 Exercícios

1. Ajuste os hiperparâmetros para melhorar a precisão da classificação.
2. Por que existem duas camadas convolucionais 1×1 no bloco NiN? Remova um deles e então observe e analise os fenômenos experimentais.
3. Calcule o uso de recursos para NiN.
   1. Qual é o número de parâmetros?
   2. Qual é a quantidade de computação?
   3. Qual é a quantidade de memória necessária durante o treinamento?
   4. Qual é a quantidade de memória necessária durante a previsão?
4. Quais são os possíveis problemas com a redução da representação $384 \times 5 \times 5$ para uma representação $10 \times 5 \times 5$ em uma etapa?

Discussão[77]

7.4 Redes com Concatenações Paralelas (GoogLeNet)

Em 2014, *GoogLeNet* venceu o ImageNet Challenge, propondo uma estrutura que combinou as forças de NiN e paradigmas de blocos repetidos (Szegedy et al., 2015). Um dos focos do artigo foi abordar a questão dos quais tamanhos de núcleos de convolução são os melhores. Afinal, as redes populares anteriores empregavam escolhas tão pequenas quanto 1×1 e tão grande quanto 11×11. Uma ideia neste artigo foi que às vezes pode ser vantajoso empregar uma combinação de grãos de vários tamanhos. Nesta seção, apresentaremos GoogLeNet, apresentando uma versão ligeiramente simplificada do modelo original: nós omitir alguns recursos ad-hoc que foram adicionados para estabilizar o treinamento mas são desnecessários agora com melhores algoritmos de treinamento disponíveis.

7.4.1 Inception Blocks

O bloco convolucional básico no GoogLeNet é chamado de *bloco Inception*, provavelmente nomeado devido a uma citação do filme *Inception* ("Precisamos ir mais fundo"), que lançou um meme viral.

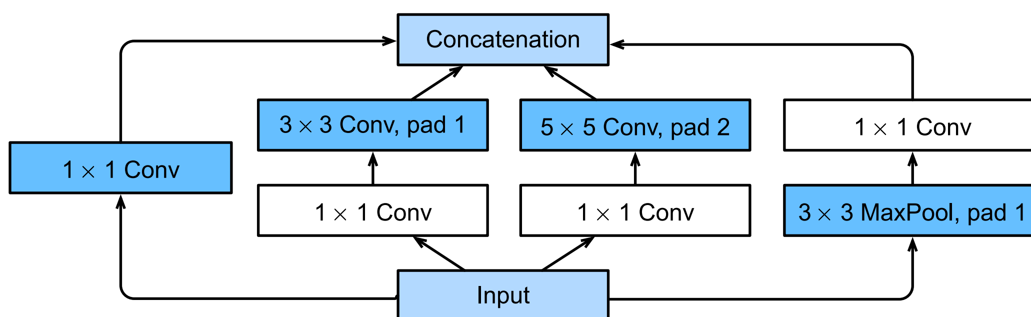


Fig. 7.4.1: Estrutura do bloco Inception.

[77] https://discuss.d2l.ai/t/80

Conforme descrito em Fig. 7.4.1, o bloco de iniciação consiste em quatro caminhos paralelos. Os primeiros três caminhos usam camadas convolucionais com tamanhos de janela de 1×1, 3×3, e 5×5 para extrair informações de diferentes tamanhos espaciais. Os dois caminhos intermediários realizam uma convolução 1×1 na entrada para reduzir o número de canais, diminuindo a complexidade do modelo. O quarto caminho usa uma camada de pooling máxima de 3×3, seguido por uma camada convolucional 1×1 para alterar o número de canais. Todos os quatro caminhos usam preenchimento apropriado para dar à entrada e saída a mesma altura e largura. Finalmente, as saídas ao longo de cada caminho são concatenadas ao longo da dimensão do canal e compreendem a saída do bloco. Os hiperparâmetros comumente ajustados do bloco de início são o número de canais de saída por camada.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


class Inception(nn.Module):
    # `c1`--`c4` are the number of output channels for each path
    def __init__(self, in_channels, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = nn.Conv2d(in_channels, c1, kernel_size=1)
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2d(in_channels, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = nn.Conv2d(in_channels, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_channels, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        # Concatenate the outputs on the channel dimension
        return torch.cat((p1, p2, p3, p4), dim=1)
```

Para ter alguma intuição de por que essa rede funciona tão bem, considere a combinação dos filtros. Eles exploram a imagem em uma variedade de tamanhos de filtro. Isso significa que os detalhes em diferentes extensões pode ser reconhecido de forma eficiente por filtros de diferentes tamanhos. Ao mesmo tempo, podemos alocar diferentes quantidades de parâmetros para filtros diferentes.

7.4.2 Modelo GoogLeNet

Conforme mostrado em Fig. 7.4.2, GoogLeNet usa uma pilha de um total de 9 blocos iniciais e pooling médio global para gerar suas estimativas. O agrupamento máximo entre os blocos de iniciação reduz a dimensionalidade. O primeiro módulo é semelhante ao AlexNet e LeNet. A pilha de blocos é herdada de VGG e o pool de média global evita uma pilha de camadas totalmente conectadas no final.

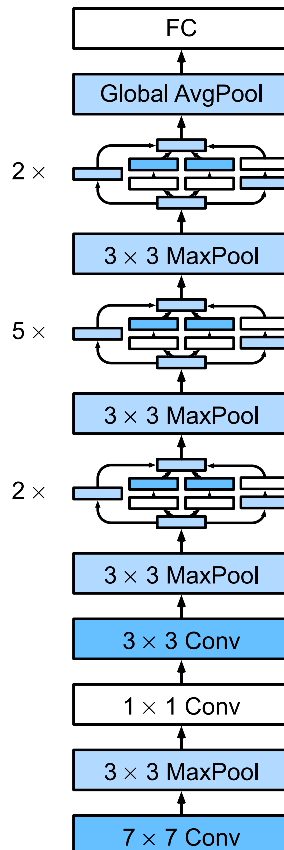


Fig. 7.4.2: A arquitetura GoogLeNet.

Agora podemos implementar o GoogLeNet peça por peça. O primeiro módulo usa uma camada convolucional 7×7 de 64 canais.

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

O segundo módulo usa duas camadas convolucionais: primeiro, uma camada convolucional 1×1 de 64 canais, em seguida, uma camada convolucional 3×3 que triplica o número de canais. Isso corresponde ao segundo caminho no bloco *Inception*.

```
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.ReLU(),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

O terceiro módulo conecta dois blocos de iniciação completos em série. O número de canais de saída do primeiro bloco de iniciação é $64 + 128 + 32 + 32 = 256$, e a relação número de canal de saída entre os quatro caminhos está $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. O segundo e o terceiro caminhos reduzem primeiro o número de canais de entrada para $96/192 = 1/2$ e $16/192 = 1/12$, respectivamente, e conecte a segunda camada convolucional. O número de canais de saída do segundo bloco de iniciação é aumentado para $128 + 192 + 96 + 64 = 480$, e a proporção do número de canal de saída entre os quatro caminhos está $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. O segundo e o terceiro caminhos reduzem primeiro o número de canais de entrada a $128/256 = 1/2$ e $32/256 = 1/8$, respectivamente.

```
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

O quarto módulo é mais complicado. Ele conecta cinco blocos de iniciação em série, e eles têm $192+208+48+64 = 512$, $160+224+64+64 = 512$, $128+256+64+64 = 512$, $112+288+64+64 = 528$, e $256 + 320 + 128 + 128 = 832$ canais de saída, respectivamente. O número de canais atribuídos a esses caminhos é semelhante para aquele no terceiro módulo: o segundo caminho com a camada convolucional 3×3 produz o maior número de canais, seguido pelo primeiro caminho com apenas a camada convolucional 1×1, o terceiro caminho com a camada convolucional 5×5, e o quarto caminho com a camada de pooling máxima 3×3. O segundo e terceiro caminhos irão primeiro reduzir o número de canais de acordo com a proporção. Essas proporções são ligeiramente diferentes em diferentes blocos *Inception*.

```
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
                   Inception(512, 128, (128, 256), (24, 64), 64),
                   Inception(512, 112, (144, 288), (32, 64), 64),
                   Inception(528, 256, (160, 320), (32, 128), 128),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

O quinto módulo tem dois blocos de iniciação com $256 + 320 + 128 + 128 = 832$ e$384 + 384 + 128 + 128 = 1024$ canais de saída. O número de canais atribuídos a cada caminho é o mesmo que no terceiro e quarto módulos, mas difere em valores específicos. Deve-se notar que o quinto bloco é seguido pela camada de saída. Este bloco usa a camada de pooling média global para alterar a altura e largura de cada canal para 1, assim como em NiN. Por fim, transformamos a saída em uma matriz bidimensional seguido por uma camada totalmente conectada cujo número de saídas é o número de classes de rótulo.

```
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                   Inception(832, 384, (192, 384), (48, 128), 128),
                   nn.AdaptiveAvgPool2d((1,1)),
                   nn.Flatten())

net = nn.Sequential(b1, b2, b3, b4, b5, nn.Linear(1024, 10))
```

O modelo GoogLeNet é computacionalmente complexo, portanto, não é tão fácil modificar o número de canais como no VGG. Para ter um tempo de treinamento razoável no Fashion-MNIST, reduzimos a altura e largura de entrada de 224 para 96. Isso simplifica o cálculo. As mudanças na forma da saída entre os vários módulos são demonstrados abaixo.

```
X = torch.rand(size=(1, 1, 96, 96))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

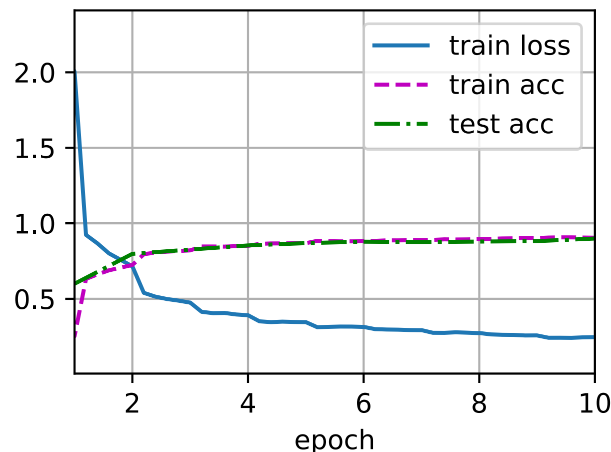
```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 192, 12, 12])
Sequential output shape:     torch.Size([1, 480, 6, 6])
Sequential output shape:     torch.Size([1, 832, 3, 3])
Sequential output shape:     torch.Size([1, 1024])
Linear output shape:         torch.Size([1, 10])
```

7.4.3 Treinamento

Como antes, treinamos nosso modelo usando o conjunto de dados Fashion-MNIST. Nós o transformamos em resolução de 96×96 pixels antes de invocar o procedimento de treinamento.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.246, train acc 0.905, test acc 0.899
3468.7 examples/sec on cuda:0
```



7.4.4 Sumário

- O bloco de iniciação é equivalente a uma sub-rede com quatro caminhos. Ele extrai informações em paralelo por meio de camadas convolucionais de diferentes formatos de janela e camadas de agrupamento máximo. As convoluções 1×1 reduzem a dimensionalidade do canal em um nível por pixel. O pool máximo reduz a resolução.
- GoogLeNet conecta vários blocos de iniciação bem projetados com outras camadas em série. A proporção do número de canais atribuídos no bloco de iniciação é obtida por meio de um grande número de experimentos no conjunto de dados ImageNet.

- GoogLeNet, assim como suas versões subsequentes, foi um dos modelos mais eficientes no ImageNet, fornecendo precisão de teste semelhante com menor complexidade computacional.

7.4.5 Exercícios

1. Existem várias iterações do GoogLeNet. Tente implementá-los e executá-los. Alguns deles incluem o seguinte:
    - Adicione uma camada de normalização em lote (Ioffe & Szegedy, 2015), conforme descrito mais tarde em Section 7.5.
    - Faça ajustes no bloco de iniciação (Szegedy et al., 2016).
    - Use suavização de rótulo para regularização de modelo (Szegedy et al., 2016).
    - Incluir na conexão residual (Szegedy et al., 2017), conforme descrito posteriormente em Section 7.6.
2. Qual é o tamanho mínimo de imagem para o GoogLeNet funcionar?
3. Compare os tamanhos dos parâmetros do modelo de AlexNet, VGG e NiN com GoogLeNet. Como as duas últimas arquiteturas de rede reduzem significativamente o tamanho do parâmetro do modelo?

Discussão[78]

7.5 Normalização de Lotes

Treinar redes neurais profundas é difícil. E fazer com que eles convirjam em um período de tempo razoável pode ser complicado. Nesta seção, descrevemos a *normalização em lote*, uma técnica popular e eficaz que acelera consistentemente a convergência de redes profundas (Ioffe & Szegedy, 2015). Juntamente com os blocos residuais — cobertos posteriormente em Section 7.6 — normalização em lote tornou possível para os praticantes para treinar rotineiramente redes com mais de 100 camadas.

7.5.1 Treinando Redes Profundas

Para motivar a normalização do lote, vamos revisar alguns desafios práticos que surgem ao treinar modelos de aprendizado de máquina e redes neurais em particular.

Em primeiro lugar, as escolhas relativas ao pré-processamento de dados costumam fazer uma enorme diferença nos resultados finais. Lembre-se de nossa aplicação de MLPs para prever preços de casas (Section 4.10). Nosso primeiro passo ao trabalhar com dados reais era padronizar nossos recursos de entrada para cada um tem uma média de zero e variância de um. Intuitivamente, essa padronização funciona bem com nossos otimizadores porque coloca os parâmetros *a priori* em uma escala semelhante.

Em segundo lugar, para um MLP ou CNN típico, enquanto treinamos, as variáveis (por exemplo, saídas de transformação afim em MLP) em camadas intermediárias podem assumir valores com magnitudes amplamente variáveis: tanto ao longo das camadas da entrada até a saída, através das

[78] https://discuss.d2l.ai/t/82

unidades na mesma camada, e ao longo do tempo devido às nossas atualizações nos parâmetros do modelo. Os inventores da normalização de lote postularam informalmente que esse desvio na distribuição de tais variáveis poderia dificultar a convergência da rede. Intuitivamente, podemos conjeturar que se um camada tem valores variáveis que são 100 vezes maiores que os de outra camada, isso pode exigir ajustes compensatórios nas taxas de aprendizagem.

Terceiro, redes mais profundas são complexas e facilmente capazes de overfitting. Isso significa que a regularização se torna mais crítica.

A normalização em lote é aplicada a camadas individuais (opcionalmente, para todos eles) e funciona da seguinte forma: Em cada iteração de treinamento, primeiro normalizamos as entradas (de normalização em lote) subtraindo sua média e dividindo por seu desvio padrão, onde ambos são estimados com base nas estatísticas do minibatch atual. Em seguida, aplicamos um coeficiente de escala e um deslocamento de escala. É precisamente devido a esta *normalização* baseada em estatísticas *batch* que *normalização de lote* deriva seu nome.

Observe que se tentamos aplicar a normalização de lote com minibatches de tamanho 1, não seríamos capazes de aprender nada. Isso porque depois de subtrair as médias, cada unidade oculta teria valor 0! Como você pode imaginar, uma vez que estamos dedicando uma seção inteira à normalização em lote, com minibatches grandes o suficiente, a abordagem se mostra eficaz e estável. Uma lição aqui é que, ao aplicar a normalização em lote, a escolha do tamanho do lote pode ser ainda mais significativo do que sem normalização em lote.

Formalmente, denotando por $\mathbf{x} \in \mathcal{B}$ uma entrada para normalização em lote (BN) que é de um minibatch $\mathcal{B}$, a normalização em lote transforma $\mathbf{x}$ de acordo com a seguinte expressão:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_\mathcal{B}}{\hat{\boldsymbol{\sigma}}_\mathcal{B}} + \boldsymbol{\beta}. \tag{7.5.1}$$

Em (7.5.1), $\hat{\boldsymbol{\mu}}_\mathcal{B}$ é a média da amostra e $\hat{\boldsymbol{\sigma}}_\mathcal{B}$ é o desvio padrão da amostra do minibatch $\mathcal{B}$. Depois de aplicar a padronização, o minibatch resultante tem média zero e variância unitária. Porque a escolha da variação da unidade (vs. algum outro número mágico) é uma escolha arbitrária, normalmente incluímos elemento a elemento *parâmetro de escala* $\boldsymbol{\gamma}$ e *parâmetro de deslocamento* $\boldsymbol{\beta}$ que tem a mesma forma de $\mathbf{x}$. Observe que $\boldsymbol{\gamma}$ e$\boldsymbol{\beta}$ são parâmetros que precisam ser aprendidos em conjunto com os outros parâmetros do modelo.

Consequentemente, as magnitudes variáveis para camadas intermediárias não pode divergir durante o treinamento porque a normalização em lote centra ativamente e os redimensiona de volta para uma dada média e tamanho (via $\hat{\boldsymbol{\mu}}_\mathcal{B}$ e $\hat{\boldsymbol{\sigma}}_\mathcal{B}$) Um pedaço da intuição ou sabedoria do praticante é que a normalização em lote parece permitir taxas de aprendizagem mais agressivas.

Formalmente, calculamos $\hat{\boldsymbol{\mu}}_\mathcal{B}$ e$\hat{\boldsymbol{\sigma}}_\mathcal{B}$ em (7.5.1) como a seguir:

$$\begin{aligned} \hat{\boldsymbol{\mu}}_\mathcal{B} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x},\\
\hat{\boldsymbol{\sigma}}_\mathcal{B}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.\end{aligned} \tag{7.5.2}$$

Observe que adicionamos uma pequena constante $\epsilon > 0$ para a estimativa de variância para garantir que nunca tentemos a divisão por zero, mesmo nos casos em que a estimativa de variância empírica pode desaparecer. As estimativas $\hat{\boldsymbol{\mu}}_\mathcal{B}$ e $\hat{\boldsymbol{\sigma}}_\mathcal{B}$ neutralizam o problema de escala usando estimativas ruidosas de média e variância. Você pode pensar que esse barulho deve ser um problema. Acontece que isso é realmente benéfico.

Esse é um tema recorrente no aprendizado profundo. Por razões que ainda não são bem caracterizadas teoricamente, várias fontes de ruído na otimização muitas vezes levam a um treina-

mento mais rápido e menor sobreajuste: essa variação parece atuar como uma forma de regularização. Em algumas pesquisas preliminares, (Teye et al., 2018) e (Luo et al., 2018) relacionam as propriedades de normalização de lote aos antecedentes e penalidades Bayesianas, respectivamente. Em particular, isso lança alguma luz sobre o quebra-cabeça de por que a normalização em lote funciona melhor para tamanhos moderados de minibatches na faixa de $50 \sim 100$.

Consertando um modelo treinado, você pode pensar que preferiríamos usar todo o conjunto de dados para estimar a média e a variância. Depois que o treinamento for concluído, por que desejaríamos a mesma imagem a ser classificada de forma diferente, dependendo do lote em que reside? Durante o treinamento, esse cálculo exato é inviável porque as variáveis intermediárias para todos os exemplos de dados mudar toda vez que atualizamos nosso modelo. No entanto, uma vez que o modelo é treinado, podemos calcular as médias e variações das variáveis de cada camada com base em todo o conjunto de dados. Na verdade, esta é uma prática padrão para modelos que empregam normalização em lote e, portanto, as camadas de normalização em lote funcionam de maneira diferente no *modo de treinamento* (normalizando por estatísticas de minibatch) e no *modo de previsão* (normalizando por estatísticas do conjunto de dados).

Agora estamos prontos para dar uma olhada em como a normalização em lote funciona na prática.

7.5.2 Camadas de Normalização de Lotes

Implementações de normalização em lote para camadas totalmente conectadas e as camadas convolucionais são ligeiramente diferentes. Discutimos ambos os casos abaixo. Lembre-se de que uma diferença fundamental entre a normalização em lote e outras camadas é que porque a normalização de lote opera em um minibatch completo por vez, não podemos simplesmente ignorar a dimensão do lote como fizemos antes ao introduzir outras camadas.

Camadas Completamente Conectadas

Ao aplicar a normalização de lote a camadas totalmente conectadas, o papel original insere a normalização do lote após a transformação afim e antes da função de ativação não linear (aplicativos posteriores podem inserir a normalização em lote logo após as funções de ativação) (Ioffe & Szegedy, 2015). Denotando a entrada para a camada totalmente conectada por $\mathbf{x}$, a transformação afim por $\mathbf{W}\mathbf{x} + \mathbf{b}$ (com o parâmetro de peso $\mathbf{W}$ e o parâmetro de polarização $\mathbf{b}$), e a função de ativação por ϕ, podemos expressar o cálculo de uma normalização de lote habilitada, saída de camada totalmente conectada $\mathbf{h}$ como segue:

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{7.5.3}$$

Lembre-se de que a média e a variância são calculadas no *mesmo* minibatch no qual a transformação é aplicada.

Convolutional Layers

Da mesma forma, com camadas convolucionais, podemos aplicar a normalização de lote após a convolução e antes da função de ativação não linear. Quando a convolução tem vários canais de saída, precisamos realizar a normalização do lote para *cada* uma das saídas desses canais, e cada canal tem sua própria escala e parâmetros de deslocamento, ambos são escalares. Suponha que nossos minibatches contenham m exemplos e isso para cada canal, a saída da convolução tem altura p e largura q. Para camadas convolucionais, realizamos a normalização de cada lote nos elementos $m \cdot p \cdot q$ por canal de saída simultaneamente. Assim, coletamos os valores de todas as localizações espaciais ao calcular a média e a variância e consequentemente aplique a mesma média e variância dentro de um determinado canal para normalizar o valor em cada localização espacial.

Normalização de Lotes Durante Predição

Como mencionamos anteriormente, a normalização em lote normalmente se comporta de maneira diferente no modo de treinamento e no modo de previsão. Primeiro, o ruído na média da amostra e a variância da amostra decorrentes da estimativa de cada um em minibatches não são mais desejáveis, uma vez que treinamos o modelo. Em segundo lugar, podemos não ter o luxo de computação de estatísticas de normalização por lote. Por exemplo, podemos precisar aplicar nosso modelo para fazer uma previsão de cada vez.

Normalmente, após o treinamento, usamos todo o conjunto de dados para calcular estimativas estáveis das estatísticas variáveis e, em seguida, corrigi-los no momento da previsão. Consequentemente, a normalização do lote se comporta de maneira diferente durante o treinamento e no momento do teste. Lembre-se de que o abandono também exibe essa característica.

7.5.3 Implementação do Zero

Abaixo, implementamos uma camada de normalização em lote com tensores do zero.

```
import torch
from torch import nn
from d2l import torch as d2l


def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use `is_grad_enabled` to determine whether the current mode is training
    # mode or prediction mode
    if not torch.is_grad_enabled():
        # If it is prediction mode, directly use the mean and variance
        # obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully-connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
```

(continues on next page)

(continued from previous page)

```
        # When using a two-dimensional convolutional layer, calculate the
        # mean and variance on the channel dimension (axis=1). Here we
        # need to maintain the shape of `X`, so that the broadcasting
        # operation can be carried out later
        mean = X.mean(dim=(0, 2, 3), keepdim=True)
        var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
    # In training mode, the current mean and variance are used for the
    # standardization
    X_hat = (X - mean) / torch.sqrt(var + eps)
    # Update the mean and variance using moving average
    moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
    moving_var = momentum * moving_var + (1.0 - momentum) * var
Y = gamma * X_hat + beta  # Scale and shift
return Y, moving_mean.data, moving_var.data
```

Agora podemos criar uma camada `BatchNorm` adequada. Nossa camada manterá os parâmetros adequados para escala `gama` e deslocamento`beta`, ambos serão atualizados no decorrer do treinamento. Além disso, nossa camada manterá médias móveis das médias e variâncias para uso subsequente durante a previsão do modelo.

Deixando de lado os detalhes algorítmicos, observe o padrão de design subjacente à nossa implementação da camada. Normalmente, definimos a matemática em uma função separada, digamos `batch_norm`. Em seguida, integramos essa funcionalidade em uma camada personalizada, cujo código aborda principalmente questões de contabilidade, como mover dados para o contexto certo do dispositivo, alocar e inicializar quaisquer variáveis necessárias, acompanhar as médias móveis (aqui para média e variância) e assim por diante. Esse padrão permite uma separação clara da matemática do código clichê. Observe também que por uma questão de conveniência não nos preocupamos em inferir automaticamente a forma de entrada aqui, portanto, precisamos especificar o número de recursos por toda parte. Não se preocupe, as APIs de normalização de lote de alto nível na estrutura de aprendizado profundo cuidarão disso para nós e iremos demonstrar isso mais tarde.

```
class BatchNorm(nn.Module):
    # `num_features`: the number of outputs for a fully-connected layer
    # or the number of output channels for a convolutional layer. `num_dims`:
    # 2 for a fully-connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # The variables that are not model parameters are initialized to 0 and 1
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.ones(shape)

    def forward(self, X):
        # If `X` is not on the main memory, copy `moving_mean` and
        # `moving_var` to the device where `X` is located
```

(continues on next page)

(continued from previous page)

```
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # Save the updated `moving_mean` and `moving_var`
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

7.5.4 Aplicando Normalização de Lotes em LeNet

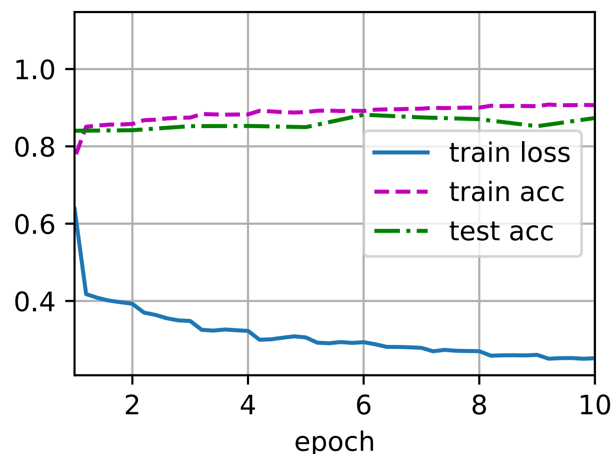
Para ver como aplicar `BatchNorm` no contexto, abaixo nós o aplicamos a um modelo LeNet tradicional (Section 6.6). Lembre-se de que a normalização de lote é aplicada após as camadas convolucionais ou camadas totalmente conectadas mas antes das funções de ativação correspondentes.

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), BatchNorm(6, num_dims=4), nn.Sigmoid(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), BatchNorm(16, num_dims=4), nn.Sigmoid(),
    nn.MaxPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(16*4*4, 120), BatchNorm(120, num_dims=2), nn.Sigmoid(),
    nn.Linear(120, 84), BatchNorm(84, num_dims=2), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Como antes, treinaremos nossa rede no conjunto de dados Fashion-MNIST. Este código é virtualmente idêntico àquele quando treinamos o LeNet pela primeira vez (Section 6.6). A principal diferença é a taxa de aprendizagem consideravelmente maior.

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.252, train acc 0.907, test acc 0.873
32992.0 examples/sec on cuda:0
```



Vamos dar uma olhada no parâmetro de escala gamma e o parâmetro *shift* beta aprendeu da primeira camada de normalização de lote.

```
net[1].gamma.reshape((-1,)), net[1].beta.reshape((-1,))
```

```
(tensor([1.2128, 1.1740, 2.2003, 2.2349, 2.1905, 1.4201], device='cuda:0',
        grad_fn=<ViewBackward>),
 tensor([-1.2550,  0.3122, -0.2924,  1.2467,  1.2471, -0.0674], device='cuda:0',
        grad_fn=<ViewBackward>))
```

7.5.5 Implementação Concisa

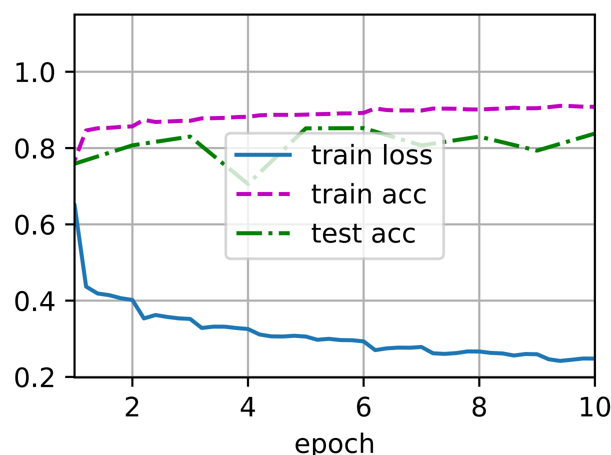
Comparado com a classe BatchNorm, que acabamos de definir a nós mesmos, podemos usar a classe BatchNorm definida em APIs de alto nível diretamente do framework de aprendizado profundo. O código parece virtualmente idêntico para a aplicação nossa implementação acima.

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), nn.BatchNorm2d(6), nn.Sigmoid(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16), nn.Sigmoid(),
    nn.MaxPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(256, 120), nn.BatchNorm1d(120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.BatchNorm1d(84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Abaixo, usamos os mesmos hiperparâmetros para treinar nosso modelo. Observe que, como de costume, a variante de API de alto nível é executada muito mais rápido porque seu código foi compilado para C ++ ou CUDA enquanto nossa implementação customizada deve ser interpretada por Python.

```
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.248, train acc 0.908, test acc 0.838
58305.7 examples/sec on cuda:0
```



7.5.6 Controvérsia

Intuitivamente, a normalização de lote é pensada para tornar o cenário de otimização mais suave. No entanto, devemos ter o cuidado de distinguir entre intuições especulativas e explicações verdadeiras para os fenômenos que observamos ao treinar modelos profundos. Lembre-se que não sabemos nem porque é mais simples redes neurais profundas (MLPs e CNNs convencionais) generalize bem em primeiro lugar. Mesmo com abandono e queda de peso, eles permanecem tão flexíveis que sua capacidade de generalizar para dados invisíveis não pode ser explicado por meio de garantias convencionais de generalização da teoria da aprendizagem.

No artigo original, propondo a normalização do lote, os autores, além de apresentar uma ferramenta poderosa e útil, ofereceram uma explicação de por que funciona: reduzindo *deslocamento interno da covariável*. Presumivelmente por *mudança interna de covariável* os autores significava algo como a intuição expressa acima — a noção de que a distribuição dos valores das variáveis muda ao longo do treinamento. No entanto, houve dois problemas com esta explicação: i) Esta deriva é muito diferente de *mudança de covariável*, tornando o nome um nome impróprio. ii) A explicação oferece uma intuição subespecificada mas deixa a questão de *por que exatamente essa técnica funciona* uma questão aberta que necessita de uma explicação rigorosa. Ao longo deste livro, pretendemos transmitir as intuições de que os praticantes usar para orientar o desenvolvimento de redes neurais profundas. No entanto, acreditamos que é importante para separar essas intuições orientadoras a partir de fato científico estabelecido. Eventualmente, quando você dominar este material e comece a escrever seus próprios artigos de pesquisa você vai querer ser claro para delinear entre afirmações técnicas e palpites.

Após o sucesso da normalização em lote, sua explicação em termos de *mudança interna de covariável* tem aparecido repetidamente em debates na literatura técnica e um discurso mais amplo sobre como apresentar a pesquisa de aprendizado de máquina. Em um discurso memorável ao aceitar o Prêmio Teste do Tempo na conferência NeurIPS de 2017, Ali Rahimi usou *mudança de covariável interna* como um ponto focal em um argumento comparando a prática moderna de aprendizado profundo a alquimia. Posteriormente, o exemplo foi revisitado em detalhes em um artigo opinativo delineando tendências preocupantes em aprendizado de máquina (Lipton & Steinhardt, 2018). Outros autores propuseram explicações alternativas para o sucesso da normalização em lote, alguns alegando que o sucesso da normalização em lote vem apesar de exibir comportamento ,de certa forma, oposto ao afirmado no artigo original (Santurkar et al., 2018).

Notamos que a *mudança interna da covariável* não é mais digna de crítica do que qualquer uma dos milhares de afirmações igualmente vagas feitas todos os anos na literatura técnica de aprendizado de máquina. Provavelmente, sua ressonância como ponto focal desses debates deve-se ao seu amplo reconhecimento pelo público-alvo. A normalização em lote provou ser um método indispensável, aplicado em quase todos os classificadores de imagem implantados, ganhando o papel que introduziu a técnica dezenas de milhares de citações.

7.5.7 Sumário

- Durante o treinamento do modelo, a normalização em lote ajusta continuamente a saída intermediária da rede neural, utilizando a média e o desvio padrão do minibatch, de modo que os valores da saída intermediária em cada camada em toda a rede neural sejam mais estáveis.
- Os métodos de normalização de lote para camadas totalmente conectadas e camadas convolucionais são ligeiramente diferentes.

- Como uma camada de eliminação, as camadas de normalização em lote têm resultados de computação diferentes no modo de treinamento e no modo de previsão.
- A normalização em lote tem muitos efeitos colaterais benéficos, principalmente o da regularização. Por outro lado, a motivação original de reduzir a mudança interna da covariável parece não ser uma explicação válida.

7.5.8 Exercícios

1. Podemos remover o parâmetro de polarização da camada totalmente conectada ou da camada convolucional antes da normalização do lote? Porque?
2. Compare as taxas de aprendizagem para LeNet com e sem normalização de lote.
    1. Plote o aumento na precisão do treinamento e do teste.
    2. Quão grande você pode aumentar a taxa de aprendizado?
3. Precisamos de normalização de lote em cada camada? Experimentar?
4. Você pode substituir o abandono pela normalização em lote? Como o comportamento muda?
5. Fixe os parâmetros `beta` e`gamma`, observe e analise os resultados.
6. Revise a documentação online para `BatchNorm` das APIs de alto nível para ver os outros aplicativos para normalização de lote.
7. Ideias de pesquisa: pense em outras transformações de normalização que você pode aplicar? Você pode aplicar a transformação integral de probabilidade? Que tal uma estimativa de covariância de classificação completa?

Discussions[79]

7.6 Redes Residuais (ResNet)

À medida que projetamos redes cada vez mais profundas, torna-se imperativo entender como a adição de camadas pode aumentar a complexidade e a expressividade da rede. Ainda mais importante é a capacidade de projetar redes onde adicionar camadas torna as redes estritamente mais expressivas, em vez de apenas diferentes. Para fazer algum progresso, precisamos de um pouco de matemática.

7.6.1 Classes Função

Considere $\mathcal{F}$, a classe de funções que uma arquitetura de rede específica (junto com as taxas de aprendizado e outras configurações de hiperparâmetros) pode alcançar. Ou seja, para todos os $f \in \mathcal{F}$ existe algum conjunto de parâmetros (por exemplo, pesos e vieses) que podem ser obtidos através do treinamento em um conjunto de dados adequado. Vamos supor que f^* seja a função "verdade" que realmente gostaríamos de encontrar. Se estiver em $\mathcal{F}$, estamos em boa forma, mas

[79] https://discuss.d2l.ai/t/84

normalmente não teremos tanta sorte. Em vez disso, tentaremos encontrar $f^*_\mathcal{F}$, que é nossa melhor aposta em $\mathcal{F}$. Por exemplo, dado um conjunto de dados com recursos $\mathbf{X}$ e rótulos $\mathbf{y}$, podemos tentar encontrá-lo resolvendo o seguinte problema de otimização:

$$f^*_\mathcal{F} \stackrel{\mathrm{def}}{=} \mathop{\mathrm{argmin}}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}. \tag{7.6.1}$$

É razoável supor que, se projetarmos uma arquitetura diferente e mais poderosa $\mathcal{F}'$, chegaremos a um resultado melhor. Em outras palavras, esperaríamos que $f^*_{\mathcal{F}'}$ seja "melhor" do que $f^*_{\mathcal{F}}$. No entanto, se $\mathcal{F} \not\subseteq \mathcal{F}'$ não há garantia de que isso acontecerá. Na verdade, $f^*_{\mathcal{F}'}$ pode muito bem ser pior. Conforme ilustrado por Fig. 7.6.1, para classes de função não aninhadas, uma classe de função maior nem sempre se aproxima da função "verdade" f^*. Por exemplo, à esquerda de: numref: `fig_functionclasses`, embora$\mathcal{F}_3$ esteja mais perto de f^* do que $\mathcal{F}_1$, $\mathcal{F}_6$ se afasta e não há garantia de que aumentar ainda mais a complexidade pode reduzir o distância de f^*. Com classes de função aninhadas onde $\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$ à direita de Fig. 7.6.1, podemos evitar o problema mencionado nas classes de função não aninhadas.

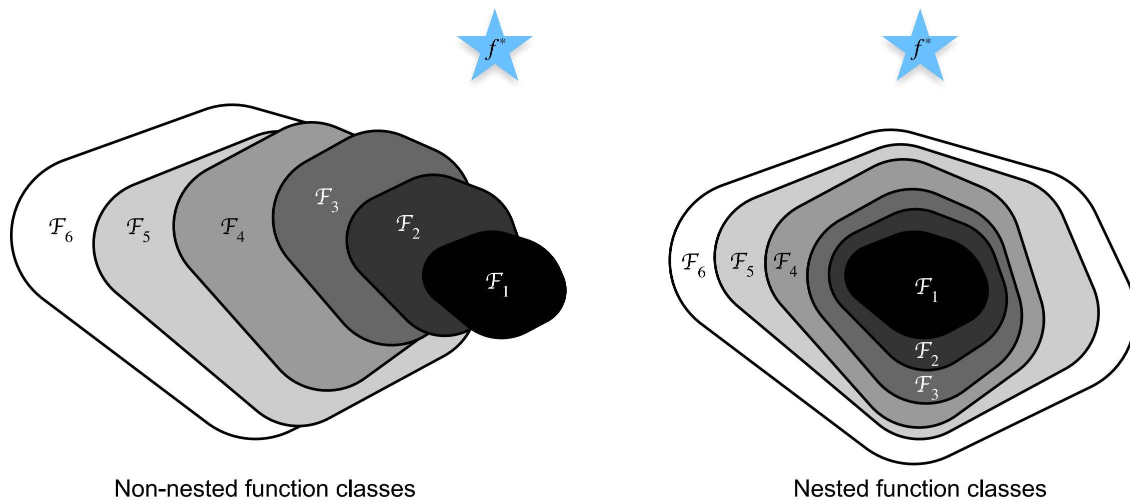


Fig. 7.6.1: Para classes de função não aninhadas, uma classe de função maior (indicada por área) não garante a aproximação da função "verdade" (f^*). Isso não acontece em classes de funções aninhadas.

Por isso, somente se as classes de função maiores contiverem as menores teremos a garantia de que aumentá-las aumenta estritamente o poder expressivo da rede. Para redes neurais profundas, se pudermos treinar a camada recém-adicionada em uma função de identidade $f(\mathbf{x}) = \mathbf{x}$, o novo modelo será tão eficaz quanto o modelo original. Como o novo modelo pode obter uma solução melhor para se ajustar ao conjunto de dados de treinamento, a camada adicionada pode facilitar a redução de erros de treinamento.

Essa é a pergunta que He et al. considerado quando se trabalha em modelos de visão computacional muito profundos (He et al., 2016). No cerne de sua proposta de *rede residual* (*ResNet*) está a ideia de que cada camada adicional deve mais facilmente conter a função de identidade como um de seus elementos. Essas considerações são bastante profundas, mas levaram a uma soluçao surpreendentemente simples, um *bloco residual*. Com ele, a ResNet venceu o Desafio de Reconhecimento Visual em Grande Escala da ImageNet em 2015. O design teve uma profunda influência em como construir redes neurais profundas.

7.6.2 Blocos Residuais

Vamos nos concentrar em uma parte local de uma rede neural, conforme descrito em Fig. 7.6.2. Denote a entrada por $\mathbf{x}$. Assumimos que o mapeamento subjacente desejado que queremos obter aprendendo é $f(\mathbf{x})$, a ser usado como entrada para a função de ativação no topo. À esquerda de Fig. 7.6.2, a parte dentro da caixa de linha pontilhada deve aprender diretamente o mapeamento $f(\mathbf{x})$. A direita, a parte de dentro da caixa de linha pontilhada precisa aprender o *mapeamento residual* $f(\mathbf{x}) - \mathbf{x}$, que é como o bloco residual deriva seu nome. Se o mapeamento de identidade $f(\mathbf{x}) = \mathbf{x}$ for o mapeamento subjacente desejado, o mapeamento residual é mais fácil de aprender: nós só precisamos empurrar os pesos e preconceitos da camada de peso superior (por exemplo, camada totalmente conectada e camada convolucional) dentro da caixa de linha pontilhada a zero. A figura certa em Fig. 7.6.2 ilustra o *bloco residual* do ResNet, onde a linha sólida carregando a entrada da camada $\mathbf{x}$ para o operador de adição é chamada de *conexão residual* (ou *conexão de atalho*). Com blocos residuais, as entradas podem para a frente se propagam mais rápido através das conexões residuais entre as camadas.

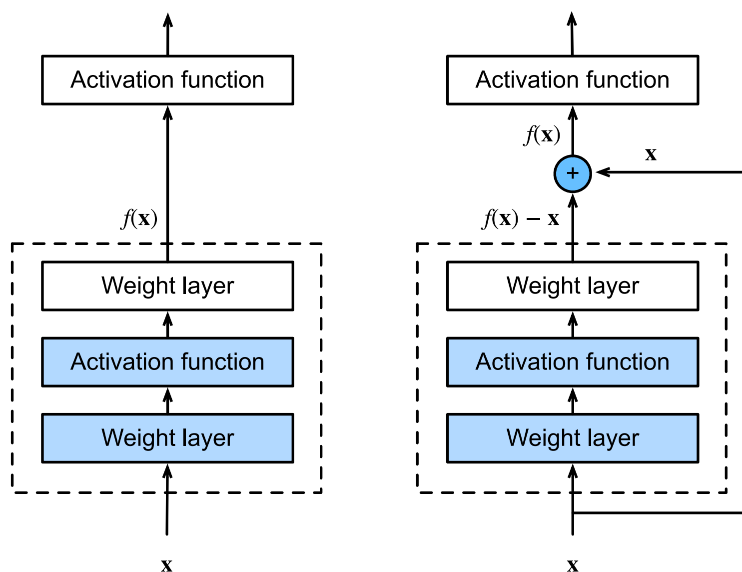


Fig. 7.6.2: Um bloco regular (esquerda) e um bloco residual (direita).

ResNet segue o design de camada convolucional 3×3 completo do VGG. O bloco residual tem duas camadas convolucionais 3×3 com o mesmo número de canais de saída. Cada camada convolucional é seguida por uma camada de normalização em lote e uma função de ativação ReLU. Em seguida, pulamos essas duas operações de convolução e adicionamos a entrada diretamente antes da função de ativação final do ReLU. Esse tipo de projeto requer que a saída das duas camadas convolucionais tenham o mesmo formato da entrada, para que possam ser somadas. Se quisermos mudar o número de canais, precisamos introduzir uma camada convolucional adicional 1×1 para transformar a entrada na forma desejada para a operação de adição. Vamos dar uma olhada no código abaixo.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

(continues on next page)

(continued from previous page)

```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet."""
    def __init__(self, input_channels, num_channels,
                 use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
                               kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
                               kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                   kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

Este código gera dois tipos de redes: uma onde adicionamos a entrada à saída antes de aplicar a não linearidade ReLU sempre que `use_1x1conv = False`, e outra onde ajustamos os canais e a resolução por meio de uma convolução 1×1 antes de adicionar. Fig. 7.6.3 ilustra isso:

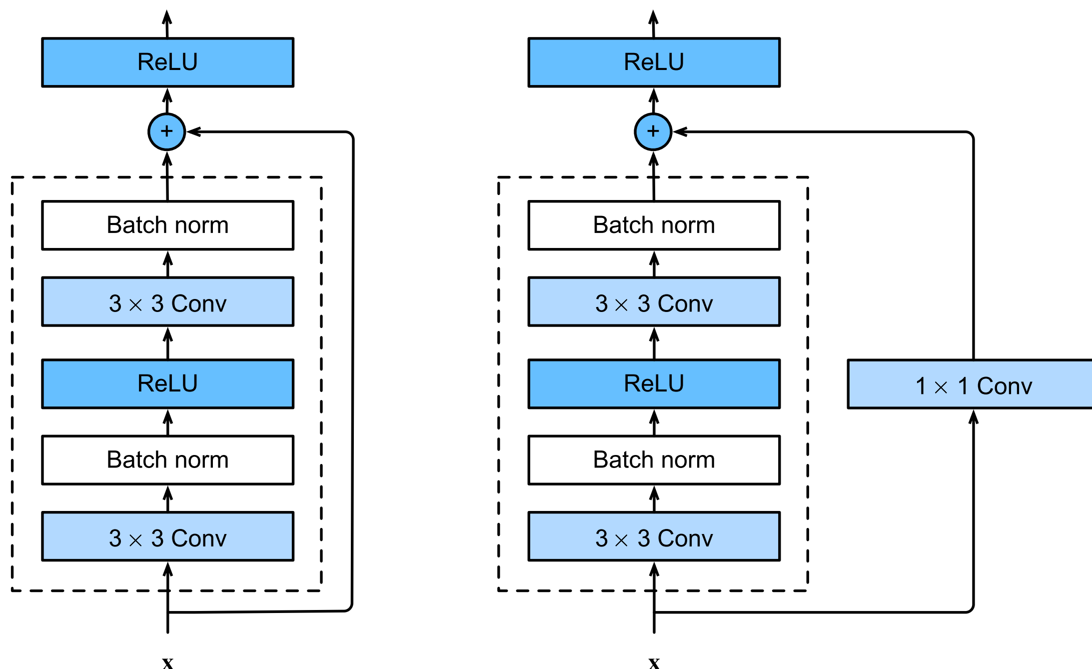


Fig. 7.6.3: Bloco ResNet com e sem convolução 1×1.

Agora, vejamos uma situação em que a entrada e a saída têm a mesma forma.

```
blk = Residual(3,3)
X = torch.rand(4, 3, 6, 6)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 3, 6, 6])
```

Também temos a opção de reduzir pela metade a altura e largura de saída, aumentando o número de canais de saída.

```
blk = Residual(3,6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

7.6.3 Modelo ResNet

As duas primeiras camadas do ResNet são iguais às do GoogLeNet que descrevemos antes: a camada convolucional 7×7 com 64 canais de saída e uma passada de 2 é seguida pela camada de pooling máxima 3×3 com uma passada de 2. A diferença é a camada de normalização de lote adicionada após cada camada convolucional no ResNet.

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet usa quatro módulos compostos de blocos de iniciação. No entanto, o ResNet usa quatro módulos compostos de blocos residuais, cada um dos quais usa vários blocos residuais com o mesmo número de canais de saída. O número de canais no primeiro módulo é igual ao número de canais de entrada. Como uma camada de pooling máxima com uma passada de 2 já foi usada, não é necessário reduzir a altura e a largura. No primeiro bloco residual para cada um dos módulos subsequentes, o número de canais é duplicado em comparação com o do módulo anterior e a altura e a largura são reduzidas à metade.

Agora, implementamos este módulo. Observe que o processamento especial foi executado no primeiro módulo.

```
def resnet_block(input_channels, num_channels, num_residuals,
                 first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels,
                                use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk
```

Em seguida, adicionamos todos os módulos ao ResNet. Aqui, dois blocos residuais são usados para cada módulo.

```
b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))
```

Finalmente, assim como GoogLeNet, adicionamos uma camada de pooling global média, seguida pela saída da camada totalmente conectada.

```
net = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear(512, 10))
```

Existem 4 camadas convolucionais em cada módulo (excluindo a camada convolucional 1×1). Junto com a primeira camada convolucional 7×7 e a camada final totalmente conectada, há 18 camadas no total. Portanto, esse modelo é comumente conhecido como ResNet-18. Configurando diferentes números de canais e blocos residuais no módulo, podemos criar diferentes modelos de ResNet, como o ResNet-152 de 152 camadas mais profundo. Embora a arquitetura principal do ResNet seja semelhante à do GoogLeNet, a estrutura do ResNet é mais simples e fácil de modificar. Todos esses fatores resultaram no uso rápido e generalizado da ResNet. Fig. 7.6.4 representa o ResNet-18 completo.

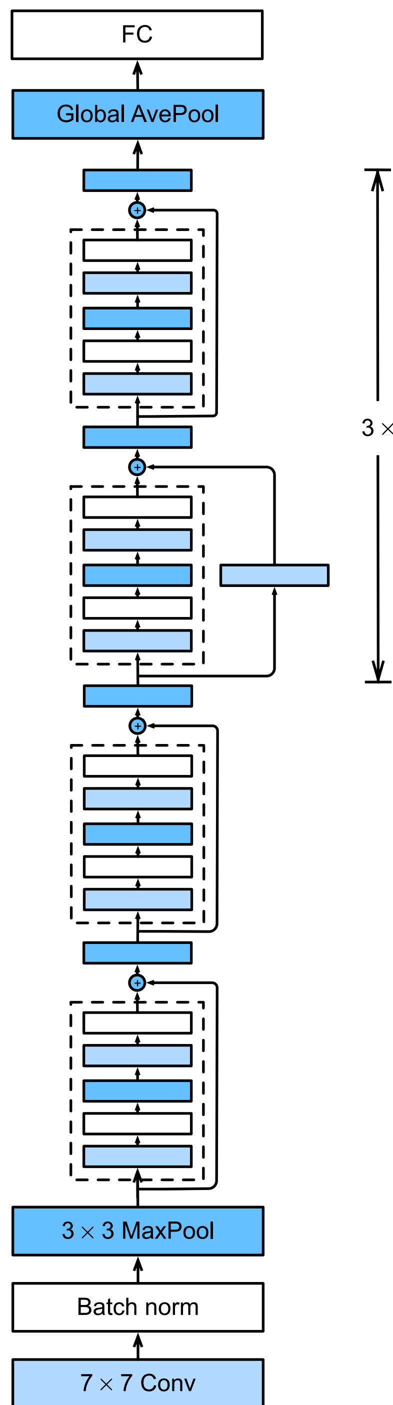


Fig. 7.6.4: A arquiteturaResNet-18.

Antes de treinar o ResNet, vamos observar como a forma da entrada muda nos diferentes módulos do ResNet. Como em todas as arquiteturas anteriores, a resolução diminui enquanto o número de canais aumenta até o ponto em que uma camada de pooling média global agrega todos os recursos.

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

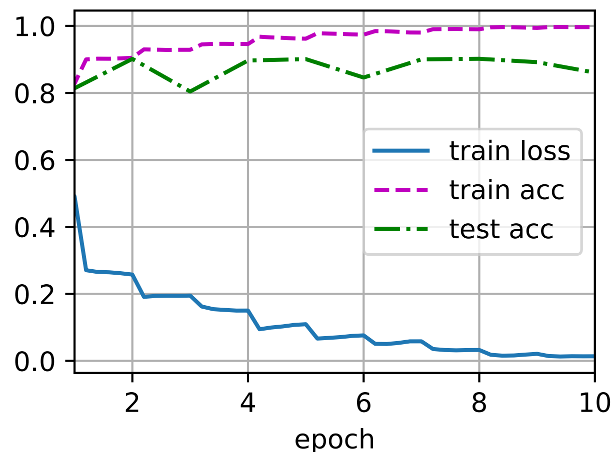
```
Sequential output shape:     torch.Size([1, 64, 56, 56])
Sequential output shape:     torch.Size([1, 64, 56, 56])
Sequential output shape:     torch.Size([1, 128, 28, 28])
Sequential output shape:     torch.Size([1, 256, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
AdaptiveAvgPool2d output shape:      torch.Size([1, 512, 1, 1])
Flatten output shape:        torch.Size([1, 512])
Linear output shape:         torch.Size([1, 10])
```

7.6.4 Treinamento

Treinamos ResNet no conjunto de dados Fashion-MNIST, assim como antes.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.014, train acc 0.997, test acc 0.861
4691.0 examples/sec on cuda:0
```



7.6.5 Sumário

- As classes de funções aninhadas são desejáveis. Aprender uma camada adicional em redes neurais profundas como uma função de identidade (embora este seja um caso extremo) deve ser facilitado.
- O mapeamento residual pode aprender a função de identidade mais facilmente, como empurrar parâmetros na camada de peso para zero.
- Podemos treinar uma rede neural profunda eficaz tendo blocos residuais. As entradas podem se propagar para frente mais rápido através das conexões residuais entre as camadas.
- O ResNet teve uma grande influência no projeto de redes neurais profundas subsequentes, tanto de natureza convolucional quanto sequencial.

7.6.6 Exercícios

1. Quais são as principais diferenças entre o bloco de iniciação em Fig. 7.4.1 e o bloco residual? Depois de remover alguns caminhos no bloco de *Inception*, como eles se relacionam?
2. Consulte a Tabela 1 no artigo ResNet (He et al., 2016) para implementar variantes diferentes.
3. Para redes mais profundas, a ResNet apresenta uma arquitetura de "gargalo" para reduzir complexidade do modelo. Tente implementá-lo.
4. Nas versões subsequentes do ResNet, os autores alteraram a configuração "convolução, lote normalização e ativação"estrutura para a" normalização em lote, estrutura de ativação e convolução ". Faça esta melhoria você mesmo. Veja a Figura 1 em [He.Zhang.Ren.ea.2016 * 1] para detalhes.
5. Por que não podemos simplesmente aumentar a complexidade das funções sem limites, mesmo se as classes de função estiverem aninhadas?

Discussions[80]

7.7 Redes Densamente Conectadas (DenseNet)

A ResNet mudou significativamente a visão de como parametrizar as funções em redes profundas. *DenseNet* (rede convolucional densa) é, até certo ponto, a extensão lógica disso (Huang et al., 2017). Para entender como chegar a isso, façamos um pequeno desvio para a matemática.

7.7.1 De ResNet para DenseNet

Lembre-se da expansão de Taylor para funções. Para o ponto $x = 0$, pode ser escrito como

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \ldots. \tag{7.7.1}$$

O ponto principal é que ele decompõe uma função em termos de ordem cada vez mais elevados. De maneira semelhante, o ResNet decompõe funções em

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{7.7.2}$$

Ou seja, o ResNet decompõe f em um termo linear simples e um termo mais complexo não linear. E se quisermos capturar (não necessariamente adicionar) informações além de dois termos? Uma solução foi DenseNet (Huang et al., 2017).

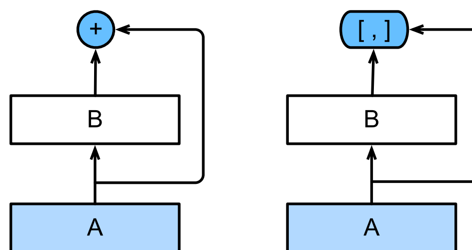


Fig. 7.7.1: A principal diferença entre ResNet (esquerda) e DenseNet (direita) em conexões de camada cruzada: uso de adição e uso de concatenação.

[80] https://discuss.d2l.ai/t/86

Conforme mostrado em Fig. 7.7.1, a principal diferença entre ResNet e DenseNet é que, no último caso, as saídas são *concatenadas* (denotadas por $[,]$) em vez de adicionadas. Como resultado, realizamos um mapeamento de $\mathbf{x}$ para seus valores após aplicar uma sequência cada vez mais complexa de funções:

$$\mathbf{x} \to \left[\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots\right]. \tag{7.7.3}$$

No final, todas essas funções são combinadas no MLP para reduzir o número de recursos novamente. Em termos de implementação, isso é bastante simples: em vez de adicionar termos, nós os concatenamos. O nome DenseNet surge do fato de o gráfico de dependência entre as variáveis se tornar bastante denso. A última camada de tal cadeia está densamente conectada a todas as camadas anteriores. As conexões densas são mostradas em Fig. 7.7.2.

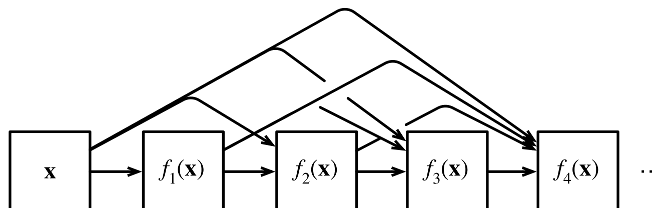


Fig. 7.7.2: Conexões densas na DenseNet.

Os principais componentes que compõem uma DenseNet são *blocos densos* e *camadas de transição*. O primeiro define como as entradas e saídas são concatenadas, enquanto o último controla o número de canais para que não seja muito grande.

7.7.2 Blocos Densos

A DenseNet usa a "normalização, ativação e convolução em lote" modificada estrutura do ResNet (veja o exercício em Section 7.6). Primeiro, implementamos essa estrutura de bloco de convolução.

```
import torch
from torch import nn
from d2l import torch as d2l


def conv_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=3, padding=1))
```

Um *bloco denso* consiste em vários blocos de convolução, cada um usando o mesmo número de canais de saída. Na propagação direta, entretanto, concatenamos a entrada e a saída de cada bloco de convolução na dimensão do canal.

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, input_channels, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
```

(continues on next page)

(continued from previous page)

```
        for i in range(num_convs):
            layer.append(conv_block(
                num_channels * i + input_channels, num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate the input and output of each block on the channel
            # dimension
            X = torch.cat((X, Y), dim=1)
        return X
```

No exemplo a seguir, definimos uma instância `DenseBlock` com 2 blocos de convolução de 10 canais de saída. Ao usar uma entrada com 3 canais, obteremos uma saída com $3 + 2 \times 10 = 23$ canais. O número de canais de bloco de convolução controla o crescimento do número de canais de saída em relação ao número de canais de entrada. Isso também é conhecido como *taxa de crescimento*.

```
blk = DenseBlock(2, 3, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 23, 8, 8])
```

7.7.3 Camadas de Transição

Uma vez que cada bloco denso aumentará o número de canais, adicionar muitos deles levará a um modelo excessivamente complexo. Uma *camada de transição* é usada para controlar a complexidade do modelo. Ele reduz o número de canais usando a camada convolucional 1×1 e divide pela metade a altura e a largura da camada de pooling média com uma distância de 2, reduzindo ainda mais a complexidade do modelo.

```
def transition_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

Aplique uma camada de transição com 10 canais à saída do bloco denso no exemplo anterior. Isso reduz o número de canais de saída para 10 e divide a altura e a largura pela metade.

```
blk = transition_block(23, 10)
blk(Y).shape
```

```
torch.Size([4, 10, 4, 4])
```

7.7.4 Modelo DenseNet

A seguir, construiremos um modelo DenseNet. A DenseNet usa primeiro a mesma camada convolucional única e camada máxima de pooling que no ResNet.

```
b1 = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Então, semelhante aos quatro módulos compostos de blocos residuais que o ResNet usa, A DenseNet usa quatro blocos densos. Semelhante ao ResNet, podemos definir o número de camadas convolucionais usadas em cada bloco denso. Aqui, nós o definimos como 4, consistente com o modelo ResNet-18 em Section 7.6. Além disso, definimos o número de canais (ou seja, taxa de crescimento) para as camadas convolucionais no bloco denso para 32, de modo que 128 canais serão adicionados a cada bloco denso.

No ResNet, a altura e a largura são reduzidas entre cada módulo por um bloco residual com uma distância de 2. Aqui, usamos a camada de transição para reduzir pela metade a altura e a largura e pela metade o número de canais.

```
# `num_channels`: the current number of channels
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]
blks = []
for i, num_convs in enumerate(num_convs_in_dense_blocks):
    blks.append(DenseBlock(num_convs, num_channels, growth_rate))
    # This is the number of output channels in the previous dense block
    num_channels += num_convs * growth_rate
    # A transition layer that halves the number of channels is added between
    # the dense blocks
    if i != len(num_convs_in_dense_blocks) - 1:
        blks.append(transition_block(num_channels, num_channels // 2))
        num_channels = num_channels // 2
```

Semelhante ao ResNet, uma camada de pooling global e uma camada totalmente conectada são conectadas na extremidade para produzir a saída.

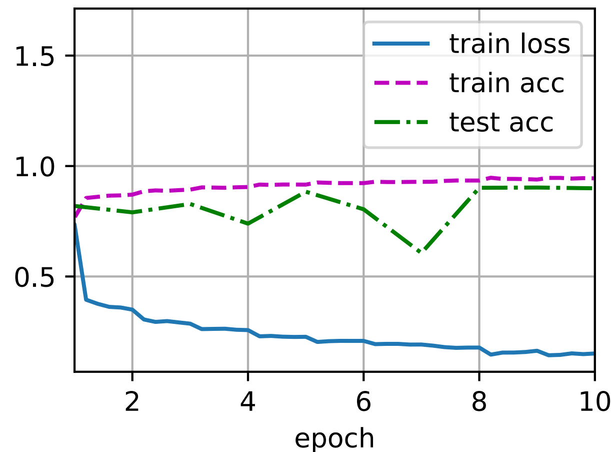
```
net = nn.Sequential(
    b1, *blks,
    nn.BatchNorm2d(num_channels), nn.ReLU(),
    nn.AdaptiveMaxPool2d((1, 1)),
    nn.Flatten(),
    nn.Linear(num_channels, 10))
```

7.7.5 Treinamento

Como estamos usando uma rede mais profunda aqui, nesta seção, reduziremos a altura e largura de entrada de 224 para 96 para simplificar o cálculo.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.151, train acc 0.944, test acc 0.899
5490.7 examples/sec on cuda:0
```



7.7.6 Sumário

- Em termos de conexões entre camadas, ao contrário do ResNet, onde entradas e saídas são adicionadas, o DenseNet concatena entradas e saídas na dimensão do canal.
- Os principais componentes que compõem o DenseNet são blocos densos e camadas de transição.
- Precisamos manter a dimensionalidade sob controle ao compor a rede, adicionando camadas de transição que reduzem o número de canais novamente.

7.7.7 Exercícios

1. Por que usamos pooling médio em vez de pooling máximo na camada de transição?
2. Uma das vantagens mencionadas no artigo da DenseNet é que seus parâmetros de modelo são menores que os do ResNet. Por que isso acontece?
3. Um problema pelo qual a DenseNet foi criticada é o alto consumo de memória.
    1. Este é realmente o caso? Tente alterar a forma de entrada para 224×224 para ver o consumo real de memória da GPU.
    2. Você consegue pensar em um meio alternativo de reduzir o consumo de memória? Como você precisa mudar a estrutura?

4. Implemente as várias versões da DenseNet apresentadas na Tabela 1 do artigo da DenseNet (Huang et al., 2017).
5. Projete um modelo baseado em MLP aplicando a ideia DenseNet. Aplique-o à tarefa de previsão do preço da habitação em Section 4.10.

Discussão[81]

[81] https://discuss.d2l.ai/t/88

8 | Redes Neurais Recorrentes

Até agora encontramos dois tipos de dados: dados tabulares e dados de imagem. Para o último, projetamos camadas especializadas para aproveitar a regularidade delas. Em outras palavras, se permutássemos os pixels em uma imagem, seria muito mais difícil raciocinar sobre seu conteúdo de algo que se pareceria muito com o fundo de um padrão de teste na época da TV analógica.

O mais importante é que, até agora, assumimos tacitamente que todos os nossos dados são retirados de alguma distribuição, e todos os exemplos são distribuídos de forma independente e idêntica (d.i.i.). Infelizmente, isso não é verdade para a maioria dos dados. Por exemplo, as palavras neste parágrafo são escritas em sequência e seria muito difícil decifrar seu significado se fossem permutadas aleatoriamente. Da mesma forma, os quadros de imagem em um vídeo, o sinal de áudio em uma conversa e o comportamento de navegação em um site seguem uma ordem sequencial. Portanto, é razoável supor que modelos especializados para esses dados terão um desempenho melhor em descrevê-los.

Outro problema surge do fato de que podemos não apenas receber uma sequência como uma entrada, mas esperar que continuemos a sequência. Por exemplo, a tarefa poderia ser continuar a série $2, 4, 6, 8, 10, \ldots$ Isso é bastante comum na análise de série temporal, para prever o mercado de ações, a curva de febre de um paciente ou a aceleração necessária para um carro de corrida. Novamente, queremos ter modelos que possam lidar com esses dados.

Em suma, embora as CNNs podem processar informações espaciais com eficiência, *as redes neurais recorrentes* (RNNs)[1] são projetadas para lidar melhor com as informações sequenciais. As RNNs introduzem variáveis de estado para armazenar informações anteriores, junto com as entradas atuais, para determinar as saídas atuais.

Muitos dos exemplos de uso de redes recorrentes são baseados em dados de texto. Portanto, enfatizaremos os modelos de linguagem neste capítulo. Após uma revisão mais formal dos dados sequenciais, apresentamos técnicas práticas para o pré-processamento desses. A seguir, discutimos os conceitos básicos de um modelo de linguagem e usamos essa discussão como inspiração para o design de RNNs. No final descrevemos o método de cálculo do gradiente para RNNs, uma vez que problemas podem ser encontrados durante o treinamento de tais redes.

[1] *recurrent neural networks*

8.1 Modelos Sequenciais

Imagine que você está assistindo a filmes na Netflix. Como um bom usuário do Netflix, você decide avaliar cada um dos filmes religiosamente. Afinal, um bom filme é um bom filme, e você quer assistir mais deles, certo? Acontece que as coisas não são tão simples. As opiniões das pessoas sobre os filmes podem mudar significativamente com o tempo. Na verdade, os psicólogos até têm nomes para alguns dos efeitos:

- Há *ancoragem*, com base na opinião de outra pessoa. Por exemplo, após a premiação do Oscar, as avaliações do filme correspondente aumentam, embora ainda seja o mesmo filme. Este efeito persiste por alguns meses até que o prêmio seja esquecido. Foi demonstrado que o efeito eleva a classificação em mais de meio ponto (Wu et al., 2017).
- Existe a *adaptação hedônica*, onde os humanos se adaptam rapidamente para aceitar uma situação melhorada ou piorada como o novo normal. Por exemplo, depois de assistir a muitos filmes bons, as expectativas de que o próximo filme seja igualmente bom ou melhor são altas. Consequentemente, mesmo um filme comum pode ser considerado ruim depois que muitos filmes incríveis são assistidos.
- Existe *sazonalidade*. Muito poucos espectadores gostam de assistir a um filme do Papai Noel em agosto.
- Em alguns casos, os filmes se tornam impopulares devido ao mau comportamento de diretores ou atores na produção.
- Alguns filmes se tornam filmes de culto, porque eram quase comicamente ruins. *Plan 9 from Outer Space* e *Troll 2* alcançaram um alto grau de notoriedade por este motivo.

Resumindo, as avaliações dos filmes são tudo menos estacionárias. Assim, usando a dinâmica temporal levou a recomendações de filmes mais precisas (Koren, 2009). É claro que os dados da sequência não se referem apenas às classificações dos filmes. O seguinte fornece mais ilustrações.

- Muitos usuários têm um comportamento altamente específico quando se trata do momento em que abrem aplicativos. Por exemplo, os aplicativos de mídia social são muito mais populares entre os alunos depois da escola. Os aplicativos de negociação do mercado de ações são mais comumente usados quando os mercados estão abertos.
- É muito mais difícil prever os preços das ações de amanhã do que preencher as lacunas de um preço das ações que perdemos ontem, embora ambos sejam apenas uma questão de estimar um número. Afinal, a previsão é muito mais difícil do que a retrospectiva. Em estatística, a primeira (previsão além das observações conhecidas) é chamada de *extrapolação*, enquanto a última (estimativa entre as observações existentes) é chamada de *interpolação*.
- Música, fala, texto e vídeos são todos sequenciais por natureza. Se fôssemos permutá-los, eles fariam pouco sentido. O título *cachorro morde homem* é muito menos surpreendente do que *homem morde cachorro*, embora as palavras sejam idênticas.
- Terremotos estão fortemente correlacionados, ou seja, após um grande terremoto, é muito provável que ocorram vários tremores menores, muito mais do que sem o forte terremoto. Na verdade, os terremotos são correlacionados espaço-temporalmente, ou seja, os tremores secundários ocorrem normalmente em um curto espaço de tempo e nas proximidades.
- Os humanos interagem uns com os outros em uma natureza sequencial, como pode ser visto nas lutas do Twitter, padrões de dança e debates.

8.1.1 Ferramentas Estatísticas

Precisamos de ferramentas estatísticas e novas arquiteturas de rede neural profunda para lidar com dados de sequência. Para manter as coisas simples, usamos o preço das ações (índice FTSE 100) ilustrado em :numref:`fig_ftse100 como exemplo.

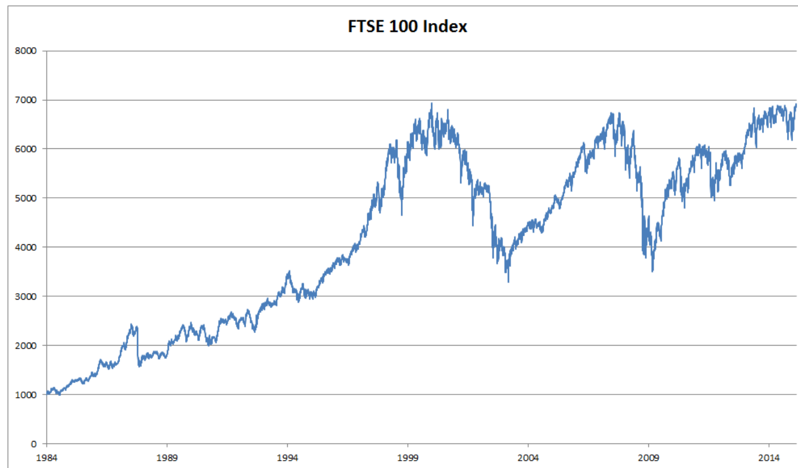


Fig. 8.1.1: Índice FTSE 100 ao longo de cerca de 30 anos.

Vamos denotar os preços por x_t, ou seja, no *passo de tempo*$t \in \mathbb{Z}^+$ observamos o preço x_t. Observe que para sequências neste texto, t normalmente será discreto e variará em números inteiros ou em seu subconjunto. Suponha que um trader que deseja ter um bom desempenho no mercado de ações no dia t prevê x_t via

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.1}$$

Modelos Autorregressivos

Para conseguir isso, nosso trader pode usar um modelo de regressão como o que treinamosem Section 3.3. Existe apenas um grande problema: o número de entradas, $x_{t-1}, \ldots, x_1$ varia, dependendo de t. Ou seja, o número aumenta com a quantidade de dados que encontramos e precisaremos de uma aproximação para tornar isso computacionalmente tratável. Muito do que se segue neste capítulo girará em torno de como estimar $P(x_t \mid x_{t-1}, \ldots, x_1)$ de forma eficiente. Em poucas palavras, ele se resume a duas estratégias como segue.

Primeiro, assuma que a sequência potencialmente bastante longa \$ $x_{t-1}, \ldots, x_1$ não é realmente necessária. Neste caso, podemos nos contentar com algum intervalo de tempo de comprimento τ e usar apenas observações $x_{t-1}, \ldots, x_{t-\tau}$. O benefício imediato é que agora o número de argumentos é sempre o mesmo, pelo menos para $t > \tau$. Isso nos permite treinar uma rede profunda, conforme indicado acima. Esses modelos serão chamados de *modelos autorregressivos*, pois eles literalmente realizam regressão em si mesmos.

A segunda estratégia, mostrada em Fig. 8.1.2, é manter algum resumo h_t das observações anteriores, e ao mesmo tempo atualizar h_t além da previsão $\hat{x}_t$. Isso leva a modelos que estimam x_t com $\hat{x}_t = P(x_t \mid h_t)$ e, além disso, atualizações da forma $h_t = g(h_{t-1}, x_{t-1})$. Como h_t nunca é observado, esses modelos também são chamados de *modelos autorregressivos latentes*.

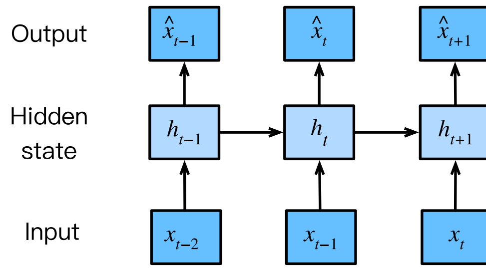


Fig. 8.1.2: Um modelo autorregressivo latente.

Ambos os casos levantam a questão óbvia de como gerar dados de treinamento. Normalmente, usa-se observações históricas para prever a próxima observação, dadas as até agora. Obviamente, não esperamos que o tempo pare. No entanto, uma suposição comum é que, embora os valores específicos de x_t possam mudar, pelo menos a dinâmica da própria sequência não mudará. Isso é razoável, uma vez que novas dinâmicas são apenas isso, novas e, portanto, não previsíveis usando os dados que temos até agora. Os estatísticos chamam a dinâmica que não muda de *estacionária*. Independentemente do que fizermos, obteremos uma estimativa de toda a sequência por meio de

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.2}$$

Observe que as considerações acima ainda valem se lidarmos com objetos discretos, como palavras, em vez de números contínuos. A única diferença é que, em tal situação, precisamos usar um classificador em vez de um modelo de regressão para estimar $P(x_t \mid x_{t-1}, \ldots, x_1)$.

Modelos de Markov

Lembre-se da aproximação de que em um modelo autoregressivo usamos apenas $x_{t-1}, \ldots, x_{t-\tau}$ em vez de $x_{t-1}, \ldots, x_1$ para estimar x_t . Sempre que essa aproximação for precisa, dizemos que a sequência satisfaz uma *condição de Markov*. Em particular, se $\tau = 1$, temos um *modelo de Markov de primeira ordem* e $P(x)$ é dado por

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ onde } P(x_1 \mid x_0) = P(x_1). \tag{8.1.3}$$

Esses modelos são particularmente bons sempre que x_t assume apenas um valor discreto, uma vez que, neste caso, a programação dinâmica pode ser usada para calcular valores exatamente ao longo da cadeia. Por exemplo, podemos calcular $P(x_{t+1} \mid x_{t-1})$ de forma eficiente:

$$\begin{aligned} P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\ &= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\ &= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1}) \end{aligned} \tag{8.1.4}$$

usando o fato de que só precisamos levar em consideração um histórico muito curto de observações anteriores: $P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$. Entraremos em detalhes da programação dinâmica está além do escopo desta seção. Algoritmos de aprendizagem de controle e reforço usam essas ferramentas extensivamente.

Causalidade

Em princípio, não há nada de errado em desdobrar $P(x_1, \ldots, x_T)$ na ordem inversa. Afinal, por condicionamento, podemos sempre escrevê-la via

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T). \tag{8.1.5}$$

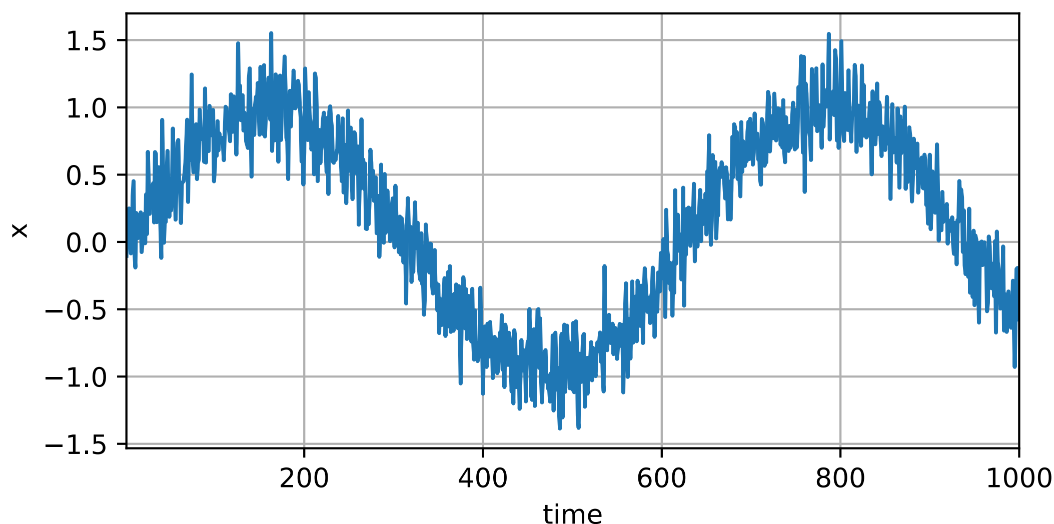
Na verdade, se tivermos um modelo de Markov, também podemos obter uma distribuição de probabilidade condicional reversa. Em muitos casos, no entanto, existe uma direção natural para os dados, a saber, avançar no tempo. É claro que eventos futuros não podem influenciar o passado. Portanto, se mudarmos x_t, podemos influenciar o que acontece com x_{t+1} daqui para frente, mas não o contrário. Ou seja, se mudarmos x_t, a distribuição sobre os eventos anteriores não mudará. Consequentemente, deveria ser mais fácil explicar $P(x_{t+1} \mid x_t)$ em vez de $P(x_t \mid x_{t+1})$. Por exemplo, foi mostrado que em alguns casos podemos encontrar $x_{t+1} = f(x_t) + \epsilon$ para algum ruído aditivo ϵ, enquanto o inverso não é verdadeiro (Hoyer et al., 2009). Esta é uma ótima notícia, pois normalmente é a direção para a frente que estamos interessados em estimar. O livro de Peters et al. explicou mais sobre este tópico (Peters et al., 2017a). Estamos apenas arranhando a superfície disso.

8.1.2 Trainamento

Depois de revisar tantas ferramentas estatísticas, vamos tentar isso na prática. Começamos gerando alguns dados. Para manter as coisas simples, geramos nossos dados de sequência usando uma função seno com algum ruído aditivo para etapas de tempo $1, 2, \ldots, 1000$.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

```
T = 1000  # Generate a total of 1000 points
time = torch.arange(1, T + 1, dtype=torch.float32)
x = torch.sin(0.01 * time) + torch.normal(0, 0.2, (T,))
d2l.plot(time, [x], 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



Em seguida, precisamos transformar essa sequência em recursos e rótulos nos quais nosso modelo pode treinar. Com base na dimensão de incorporação τ, mapeamos os dados em pares $y_t = x_t$ e $\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$. O leitor astuto deve ter notado que isso nos dá τ menos exemplos de dados, uma vez que não temos histórico suficiente para os primeiros τ deles. Uma solução simples, em particular se a sequência for longa, é descartar esses poucos termos. Como alternativa, podemos preencher a sequência com zeros. Aqui, usamos apenas os primeiros 600 pares de rótulos de recursos para treinamento.

```
tau = 4
features = torch.zeros((T - tau, tau))
for i in range(tau):
    features[:, i] = x[i: T - tau + i]
labels = x[tau:].reshape((-1, 1))
```

```
batch_size, n_train = 16, 600
# Only the first `n_train` examples are used for training
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

Aqui, mantemos a arquitetura bastante simples: apenas um MLP com duas camadas totalmente conectadas, ativação ReLU e perda quadrática.

```
# Function for initializing the weights of the network
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)

# A simple MLP
def get_net():
    net = nn.Sequential(nn.Linear(4, 10),
                        nn.ReLU(),
                        nn.Linear(10, 1))
    net.apply(init_weights)
    return net

# Square loss
loss = nn.MSELoss()
```

Agora estamos prontos para treinar o modelo. O código abaixo é essencialmente idêntico ao loop de treinamento nas seções anteriores, como Section 3.3. Portanto, não iremos nos aprofundar em muitos detalhes.

```
def train(net, train_iter, loss, epochs, lr):
    trainer = torch.optim.Adam(net.parameters(), lr)
    for epoch in range(epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.backward()
            trainer.step()
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')
```

(continues on next page)

(continued from previous page)

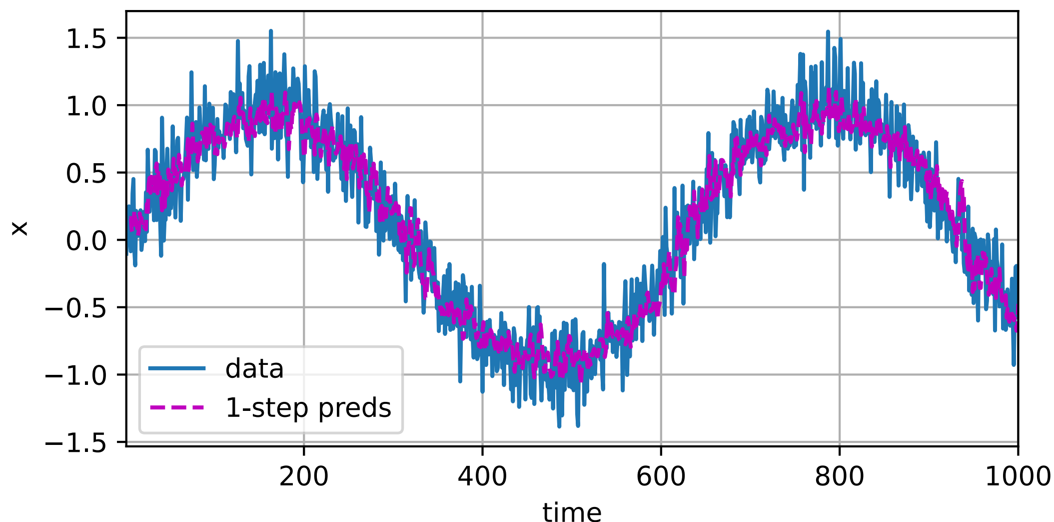
```
net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
epoch 1, loss: 0.059361
epoch 2, loss: 0.055103
epoch 3, loss: 0.052086
epoch 4, loss: 0.054857
epoch 5, loss: 0.050476
```

8.1.3 Predição

Como a perda de treinamento é pequena, esperamos que nosso modelo funcione bem. Vamos ver o que isso significa na prática. A primeira coisa a verificar é o quão bem o modelo é capaz de prever o que acontece na próxima etapa, ou seja, a *previsão um passo à frente*.

```
onestep_preds = net(features)
d2l.plot([time, time[tau:]], [x.detach().numpy(), onestep_preds.detach().numpy()], 'time',
         'x', legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



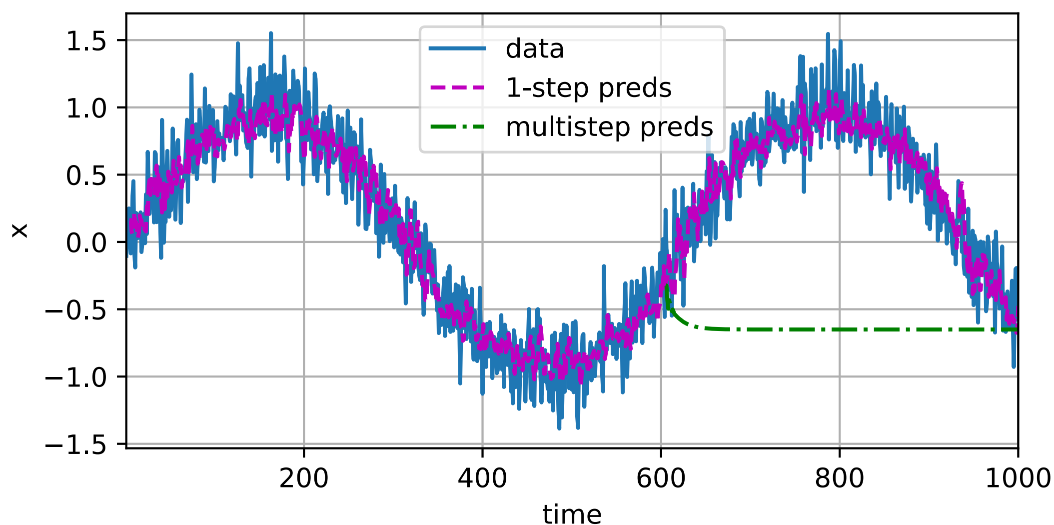
As previsões de um passo à frente parecem boas, exatamente como esperávamos. Mesmo além de 604 (`n_train + tau`) observações, as previsões ainda parecem confiáveis. No entanto, há apenas um pequeno problema para isso: se observarmos os dados de sequência apenas até a etapa de tempo 604, não podemos esperar receber as entradas para todas as previsões futuras de um passo à frente. Em vez disso, precisamos trabalhar nosso caminho adiante, um passo de cada vez:

$$
\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}), \\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}), \\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}), \\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}), \\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}), \\
&\ldots
\end{aligned}
\tag{8.1.6}
$$

Geralmente, para uma sequência observada até x_t, sua saída prevista $\hat{x}_{t+k}$ no passo de tempo $t+k$ é chamada de k - *previsão passo à frente*. Como observamos até x_{604}, sua previsão de k-step-forward é $\hat{x}_{604+k}$. Em outras palavras, teremos que usar nossas próprias previsões para fazer previsões em várias etapas. Vamos ver como isso vai bem.

```
multistep_preds = torch.zeros(T)
multistep_preds[: n_train + tau] = x[: n_train + tau]
for i in range(n_train + tau, T):
    multistep_preds[i] = net(
        multistep_preds[i - tau:i].reshape((1, -1)))
```

```
d2l.plot([time, time[tau:], time[n_train + tau:]],
         [x.detach().numpy(), onestep_preds.detach().numpy(),
          multistep_preds[n_train + tau:].detach().numpy()], 'time',
         'x', legend=['data', '1-step preds', 'multistep preds'],
         xlim=[1, 1000], figsize=(6, 3))
```



Como mostra o exemplo acima, este é um fracasso espetacular. As previsões decaem para uma constante muito rapidamente após algumas etapas de previsão. Por que o algoritmo funcionou tão mal? Em última análise, isso se deve ao fato de que os erros se acumulam. Digamos que após o passo 1 tenhamos algum erro $\epsilon_1 = \bar\epsilon$. Agora a *entrada* para a etapa 2 é perturbada por ϵ_1, portanto, sofremos algum erro na ordem de $\epsilon_2 = \bar\epsilon + c\epsilon_1$ para alguma constante c, e assim por diante. O erro pode divergir rapidamente das observações verdadeiras. Este é um fenômeno comum. Por exemplo, as previsões do tempo para as próximas 24 horas tendem a ser bastante precisas, mas, além disso, a precisão diminui rapidamente. Discutiremos métodos para melhorar isso ao longo deste capítulo e além.

Vamos dar uma olhada mais de perto nas dificuldades nas previsões de k-step-ahead calculando previsões em toda a sequência para $k = 1, 4, 16, 64$.

```
max_steps = 64
```

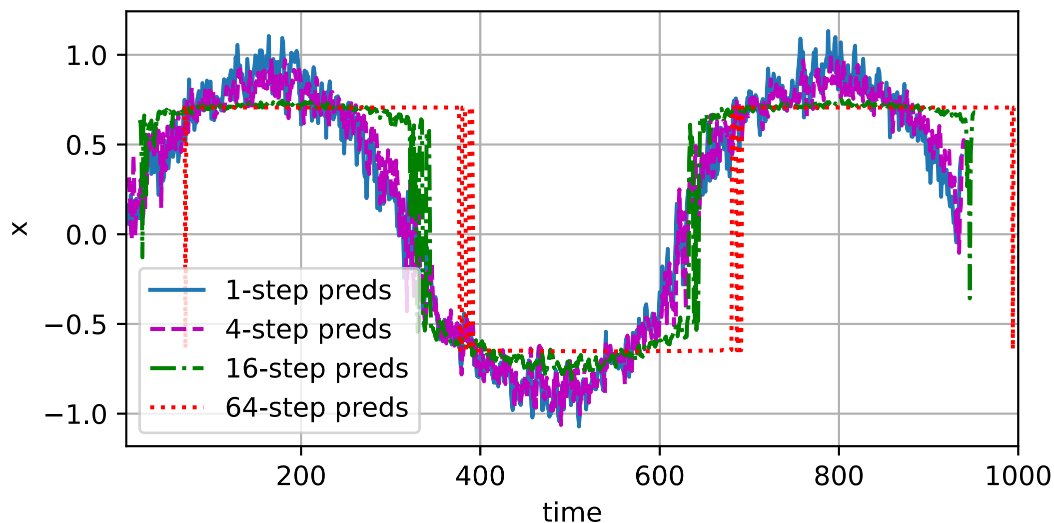
```
features = torch.zeros((T - tau - max_steps + 1, tau + max_steps))
# Column `i` (`i` < `tau`) are observations from `x` for time steps from
```

(continues on next page)

(continued from previous page)

```
# `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau):
    features[:, i] = x[i: i + T - tau - max_steps + 1]

# Column `i` (`i` >= `tau`) are the (`i - tau + 1`)-step-ahead predictions for
# time steps from `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau, tau + max_steps):
    features[:, i] = net(features[:, i - tau:i]).reshape(-1)
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1: T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].detach().numpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds' for i in steps], xlim=[5, 1000],
         figsize=(6, 3))
```



This clearly illustrates how the quality of the prediction changes as we try to predict further into the future. While the 4-step-ahead predictions still look good, anything beyond that is almost useless.

8.1.4 Resumo

- Existe uma grande diferença de dificuldade entre interpolação e extrapolação. Consequentemente, se você tiver uma sequência, sempre respeite a ordem temporal dos dados ao treinar, ou seja, nunca treine com dados futuros.
- Os modelos de sequência requerem ferramentas estatísticas especializadas para estimativa. Duas escolhas populares são os modelos autorregressivos e os modelos autorregressivos de variáveis latentes.
- Para modelos causais (por exemplo, o tempo indo para frente), estimar a direção para frente é normalmente muito mais fácil do que a direção reversa.
- Para uma sequência observada até o passo de tempo t, sua saída prevista no passo de tempo $t + k$ is the k*-previsão de avanço*. Como prevemos mais no tempo aumentando k, os erros se

acumulam e a qualidade da previsão se degrada, muitas vezes de forma dramática.

8.1.5 Exercícios

1. Melhore o modelo no experimento desta seção.
    1. Incorpora mais do que as últimas 4 observações? De quantas você realmente precisa?
    2. Quantas observações anteriores você precisaria se não houvesse ruído? Dica: você pode escrever sin e cos como uma equação diferencial.
    3. Você pode incorporar observações mais antigas enquanto mantém constante o número total de recursos? Isso melhora a precisão? Por quê?
    4. Altere a arquitetura da rede neural e avalie o desempenho.
2. Um investidor deseja encontrar um bom título para comprar. Ele olha para os retornos anteriores para decidir qual deles provavelmente terá um bom desempenho. O que poderia dar errado com essa estratégia?
3. A causalidade também se aplica ao texto? Até que ponto?
4. Dê um exemplo de quando um modelo autoregressivo latente pode ser necessário para capturar a dinâmica dos dados.

Discussions[82]

8.2 Preprocessamento de Texto

Nós revisamos e avaliamos ferramentas estatísticas e desafios de previsão para dados de sequência. Esses dados podem assumir várias formas. Especificamente, como vamos nos concentrar em em muitos capítulos do livro, text é um dos exemplos mais populares de dados de sequência. Por exemplo, um artigo pode ser visto simplesmente como uma sequência de palavras ou mesmo uma sequência de caracteres. Para facilitar nossos experimentos futuros com dados de sequência, vamos dedicar esta seção para explicar as etapas comuns de pré-processamento para texto. Normalmente, essas etapas são:

1. Carregar o texto como strings na memória.
2. Dividir as strings em tokens (por exemplo, palavras e caracteres).
3. Construir uma tabela de vocabulário para mapear os tokens divididos em índices numéricos.
4. Converter o texto em sequências de índices numéricos para que possam ser facilmente manipulados por modelos.

```
import collections
import re
from d2l import torch as d2l
```

[82] https://discuss.d2l.ai/t/114

8.2.1 Lendo o Dataset

Para começar, carregamos o texto de H. G. Wells '[*The Time Machine*] (http://www.gutenberg.org/ebooks/35). Este é um corpus bastante pequeno de pouco mais de 30000 palavras, mas para o propósito do que queremos ilustrar, está tudo bem. Coleções de documentos mais realistas contêm muitos bilhões de palavras. A função a seguir lê o conjunto de dados em uma lista de linhas de texto, onde cada linha é uma *string*. Para simplificar, aqui ignoramos a pontuação e a capitalização.

```
#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.amazonaws.com/
↪timemachine.txt...
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

8.2.2 Tokenização

A seguinte função `tokenize` recebe uma lista (`lines`) como entrada, onde cada lista é uma sequência de texto (por exemplo, uma linha de texto). Cada sequência de texto é dividida em uma lista de tokens. Um *token* é a unidade básica no texto. No fim, uma lista de listas de tokens é retornada, onde cada token é uma string.

```
def tokenize(lines, token='word'):  #@save
    """Split text lines into word or character tokens."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)

tokens = tokenize(lines)
for i in range(11):
    print(tokens[i])
```

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to', 'speak',
↪'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey', 'eyes', 'shone',
↪ 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and', 'animated',
↪'the']
```

8.2.3 Vocabulário

O tipo de string do token é inconveniente para ser usado por modelos, que usam entradas numéricas. Agora, vamos construir um dicionário, também chamado de *vocabulário*, para mapear tokens de string em índices numéricos começando em 0. Para fazer isso, primeiro contamos os tokens exclusivos em todos os documentos do conjunto de treinamento, ou seja, um *corpus*, e, em seguida, atribua um índice numérico a cada token exclusivo de acordo com sua frequência. Os tokens raramente exibidos são frequentemente removidos para reduzir a complexidade. Qualquer token que não exista no corpus ou que tenha sido removido é mapeado em um token especial desconhecido “<unk>”. Opcionalmente, adicionamos uma lista de tokens reservados, como “<pad>” para preenchimento, “<bos>” para apresentar o início de uma sequência, e “<eos>”” para o final de uma sequência.

```
class Vocab:  #@save
    """Vocabulary for text."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # The index for the unknown token is 0
        self.unk, uniq_tokens = 0, ['<unk>'] + reserved_tokens
        uniq_tokens += [token for token, freq in self.token_freqs
                        if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
```

(continues on next page)

(continued from previous page)

```
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # Here `tokens` is a 1D list or 2D list
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Flatten a list of token lists into a list of tokens
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

Construímos um vocabulário usando o conjunto de dados da máquina do tempo como corpus. Em seguida, imprimimos os primeiros tokens frequentes com seus índices.

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to', 6), ('was', 7),␣
↪('in', 8), ('that', 9)]
```

Agora podemos converter cada linha de texto em uma lista de índices numéricos.

```
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 2183, 2184, 400]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and',
↪'animated', 'the']
indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]
```

8.2.4 Juntando Todas as Coisas

Usando as funções acima, empacotamos tudo na função `load_corpus_time_machine`, que retornacorpus, uma lista de índices de token, e `vocabulário`, o vocabulário do corpus da máquina do tempo. As modificações que fizemos aqui são: i) simbolizamos o texto em caracteres, não em palavras, para simplificar o treinamento em seções posteriores; ii) `corpus` é uma lista única, não uma lista de listas de tokens, uma vez que cada linha de texto no conjunto de dados da máquina do tempo não é necessariamente uma frase ou um parágrafo.

```
def load_corpus_time_machine(max_tokens=-1):  #@save
    """Return token indices and the vocabulary of the time machine dataset."""
```

(continues on next page)

(continued from previous page)

```
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Since each text line in the time machine dataset is not necessarily a
    # sentence or a paragraph, flatten all the text lines into a single list
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(170580, 28)
```

8.2.5 Resumo

- O texto é uma forma importante de dados de sequência.
- Para pré-processar o texto, geralmente dividimos o texto em tokens, construímos um vocabulário para mapear strings de token em índices numéricos e convertemos dados de texto em índices de token para os modelos manipularem.

8.2.6 Exercícios

1. A tokenização é uma etapa chave de pré-processamento. Isso varia para diferentes idiomas. Tente encontrar outros três métodos comumente usados para tokenizar texto.
2. No experimento desta seção, tokenize o texto em palavras e varie os argumentos `min_freq` da instância`Vocab`. Como isso afeta o tamanho do vocabulário?

Discussions[83]

8.3 Modelos de Linguagem e o *Dataset*

Em Section 8.2, vemos como mapear dados de texto em tokens, onde esses tokens podem ser vistos como uma sequência de observações discretas, como palavras ou caracteres. Suponha que os tokens em uma sequência de texto de comprimento T sejam, por sua vez, $x_1, x_2, \ldots, x_T$. Então, na sequência de texto, $x_t(1 \leq t \leq T)$ pode ser considerado como a observação ou rótulo no intervalo de tempo t. Dada essa sequência de texto, o objetivo de um *modelo de linguagem* é estimar a probabilidade conjunta da sequência

$$P(x_1, x_2, \ldots, x_T). \tag{8.3.1}$$

Os modelos de linguagem são incrivelmente úteis. Por exemplo, um modelo de linguagem ideal seria capaz de gerar texto natural sozinho, simplesmente desenhando um token por vez $x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$. Bem ao contrário do macaco usando uma máquina de escrever, todo

[83] https://discuss.d2l.ai/t/115

texto emergente de tal modelo passaria como linguagem natural, por exemplo, texto em inglês. Além disso, seria suficiente para gerar um diálogo significativo, simplesmente condicionando o texto a fragmentos de diálogo anteriores. Claramente, ainda estamos muito longe de projetar tal sistema, uma vez que seria necessário *compreender* o texto em vez de apenas gerar conteúdo gramaticalmente sensato.

No entanto, os modelos de linguagem são de grande utilidade, mesmo em sua forma limitada. Por exemplo, as frases "reconhecer a fala" e "destruir uma bela praia"[1] soam muito semelhantes. Isso pode causar ambiguidade no reconhecimento de fala, o que é facilmente resolvido por meio de um modelo de linguagem que rejeita a segunda tradução como bizarra. Da mesma forma, em um algoritmo de sumarização de documentos vale a pena saber que "cachorro morde homem" é muito mais frequente do que "homem morde cachorro", ou que "quero comer vovó" é uma afirmação um tanto perturbadora, enquanto "quero comer, vovó" é muito mais benigna.

8.3.1 Aprendendo um Modelo de Linguagem

A questão óbvia é como devemos modelar um documento, ou mesmo uma sequência de tokens. Suponha que tokenizemos dados de texto no nível da palavra. Podemos recorrer à análise que aplicamos aos modelos de sequência em Section 8.1. Vamos começar aplicando regras básicas de probabilidade:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{8.3.2}$$

Por exemplo, a probabilidade de uma sequência de texto contendo quatro palavras seria dada como:

$$P(\text{deep}, \text{learning}, \text{é}, \text{divertido}) = P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{é} \mid \text{deep}, \text{learning})P(\text{divertido} \mid \text{deep}, \text{learning}, \text{é}) \tag{8.3.3}$$

Para calcular o modelo de linguagem, precisamos calcular a probabilidade de palavras e a probabilidade condicional de uma palavra dada as poucas palavras anteriores. Essas probabilidades são essencialmente parâmetros do modelo de linguagem.

Aqui nós supomos que o conjunto de dados de treinamento é um grande corpus de texto, como todas as entradas da Wikipedia, Project Gutenberg[84], e todo o texto postado no wede. A probabilidade das palavras pode ser calculada a partir da palavra relativa frequência de uma determinada palavra no conjunto de dados de treinamento. Por exemplo, a estimativa $\hat{P}(\text{deep})$ pode ser calculada como o probabilidade de qualquer frase que comece com a palavra "deep". Uma abordagem ligeiramente menos precisa seria contar todas as ocorrências de a palavra "deep" e dividi-la pelo número total de palavras em o corpus. Isso funciona muito bem, especialmente para palavras. Continuando, podemos tentar estimar

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})}, \tag{8.3.4}$$

onde $n(x)$ e $n(x, x')$ são o número de ocorrências de *singletons* e pares de palavras consecutivas, respectivamente. Infelizmente, estimando a probabilidade de um par de palavras é um pouco mais difícil, uma vez que as ocorrências de "deep learning" são muito menos frequentes. No em particular, para algumas combinações incomuns de palavras, pode ser complicado encontrar

[1] Traduzido de *to recognize speech* e *to wreck a nice beach*
[84] https://en.wikipedia.org/wiki/Project_Gutenberg

ocorrências suficientes para obter estimativas precisas. As coisas pioram com as combinações de três palavras e além. Haverá muitas combinações plausíveis de três palavras que provavelmente não veremos em nosso conjunto de dados. A menos que forneçamos alguma solução para atribuir tais combinações de palavras contagens diferentes de zero, não poderemos usá-las em um modelo de linguagem. Se o conjunto de dados for pequeno ou se as palavras forem muito raras, podemos não encontrar nem mesmo uma delas.

Uma estratégia comum é realizar alguma forma de *suavização de Laplace*. A solução é adicionar uma pequena constante a todas as contagens. Denote por n o número total de palavras em o conjunto de treinamento e m o número de palavras únicas. Esta solução ajuda com singletons, por exemplo, via

$$\begin{aligned}
\hat{P}(x) & = \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{P}(x' \mid x) & = \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\
\hat{P}(x'' \mid x,x') & = \frac{n(x, x',x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.
\end{aligned} \tag{8.3.5}$$

Aqui ϵ_1,ϵ_2, and ϵ_3 são hiperparâmetros. Tome ϵ_1 como exemplo: quando $\epsilon_1 = 0$, nenhuma suavização é aplicada; quando ϵ_1 se aproxima do infinito positivo, $\hat{P}(x)$ se aproxima da probabilidade uniforme $1/m$. O acima é uma variante bastante primitiva do que outras técnicas podem realizar: cite: `Wood.Gasthaus.Archambeau.ea.2011`.

Infelizmente, modelos como este tornam-se difíceis de manejar rapidamente pelos seguintes motivos. Primeiro, precisamos armazenar todas as contagens. Em segundo lugar, isso ignora inteiramente o significado das palavras. Para Por exemplo, "gato" e "felino" devem ocorrer em contextos relacionados. É muito difícil ajustar esses modelos a contextos adicionais, Considerando que, modelos de linguagem baseados em aprendizagem profunda são bem adequados para levar em consideração. Por último, sequências de palavras longas são quase certas de serem novas, portanto, um modelo que simplesmente conta que a frequência de sequências de palavras vistas anteriormente tem um desempenho insatisfatório.

8.3.2 Modelos de Markov e n-gramas

Antes de discutirmos as soluções que envolvem o *deep learning*, precisamos de mais terminologia e conceitos. Lembre-se de nossa discussão sobre os Modelos de Markov em Section 8.1. Vamos aplicar isso à modelagem de linguagem. Uma distribuição sobre sequências satisfaz a propriedade Markov de primeira ordem se $P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$. Ordens mais altas correspondem a dependências mais longas. Isso leva a uma série de aproximações que podemos aplicar para modelar uma sequência:

$$\begin{aligned}
P(x_1, x_2, x_3, x_4) &= P(x_1) P(x_2) P(x_3) P(x_4),\\
P(x_1, x_2, x_3, x_4) &= P(x_1) P(x_2 \mid x_1) P(x_3 \mid x_2) P(x_4 \mid x_3),\\
P(x_1, x_2, x_3, x_4) &= P(x_1) P(x_2 \mid x_1) P(x_3 \mid x_1, x_2) P(x_4 \mid x_2, x_3).
\end{aligned} \tag{8.3.6}$$

As fórmulas de probabilidade que envolvem uma, duas e três variáveis são normalmente referidas como modelos *unigrama*, *bigrama* e *trigrama*, respectivamente. A seguir, aprenderemos como projetar modelos melhores.

8.3.3 Estatísticas de Linguagem Natural

Vamos ver como isso funciona com dados reais. Construímos um vocabulário baseado no conjunto de dados da máquina do tempo, conforme apresentado em Section 8.2 e imprima as 10 palavras mais frequentes.

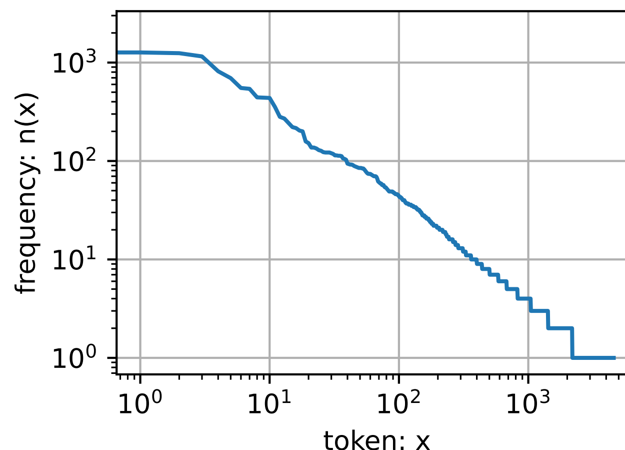
```
import random
import torch
from d2l import torch as d2l
```

```
tokens = d2l.tokenize(d2l.read_time_machine())
# Since each text line is not necessarily a sentence or a paragraph, we
# concatenate all text lines
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

Como podemos ver, as palavras mais populares são realmente muito chatas de se olhar. Freqüentemente, são chamadas de *palavras de interrupção* e, portanto, são filtradas. No entanto, eles ainda carregam significado e ainda os usaremos. Além disso, é bastante claro que a palavra frequência decai rapidamente. A 10ª palavra mais frequente é menos de $1/5$ tão comum quanto a mais popular. Para se ter uma ideia melhor, traçamos a figura da frequência da palavra.

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
```



Chegamos a algo bastante fundamental aqui: a frequência da palavra decai rapidamente de uma forma bem definida. Depois de lidar com as primeiras palavras como exceções, todas as palavras restantes seguem aproximadamente uma linha reta em um gráfico log-log. Isso significa que as palavras satisfazem a *lei de Zipf*, que afirma que a frequência n_i da i^a palavra mais frequente é:

$$n_i \propto \frac{1}{i^\alpha}, \tag{8.3.7}$$

que é equivalente a

$$\log n_i = -\alpha \log i + c, \tag{8.3.8}$$

onde α é o expoente que caracteriza a distribuição e c é uma constante. Isso já deve nos dar uma pausa se quisermos modelar palavras por estatísticas de contagem e suavização. Afinal, superestimaremos significativamente a frequência da cauda, também conhecida como palavras infrequentes. Mas e quanto às outras combinações de palavras, como bigramas, trigramas e além? Vamos ver se a frequência do bigrama se comporta da mesma maneira que a frequência do unigrama.

```
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[(('of', 'the'), 309),
 (('in', 'the'), 169),
 (('i', 'had'), 130),
 (('i', 'was'), 112),
 (('and', 'the'), 109),
 (('the', 'time'), 102),
 (('it', 'was'), 99),
 (('to', 'the'), 85),
 (('as', 'i'), 78),
 (('of', 'a'), 73)]
```

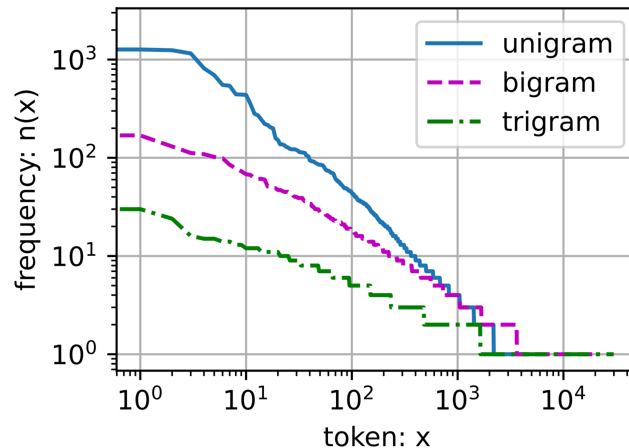
Uma coisa é notável aqui. Dos dez pares de palavras mais frequentes, nove são compostos de palavras irrelevantes e apenas um é relevante para o livro real — “o tempo”. Além disso, vejamos se a frequência do trigrama se comporta da mesma maneira.

```
trigram_tokens = [triple for triple in zip(
    corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

```
[(('the', 'time', 'traveller'), 59),
 (('the', 'time', 'machine'), 30),
 (('the', 'medical', 'man'), 24),
 (('it', 'seemed', 'to'), 16),
 (('it', 'was', 'a'), 15),
 (('here', 'and', 'there'), 15),
 (('seemed', 'to', 'me'), 14),
 (('i', 'did', 'not'), 14),
 (('i', 'saw', 'the'), 13),
 (('i', 'began', 'to'), 13)]
```

Por último, vamos visualizar a frequência do token entre esses três modelos: unigramas, bigramas e trigramas.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



Esse número é bastante empolgante por uma série de razões. Primeiro, além das palavras unigrama, as sequências de palavras também parecem seguir a lei de Zipf, embora com um expoente menor α em (8.3.7), dependendo do comprimento da sequência. Em segundo lugar, o número de n-gramas distintos não é tão grande. Isso nos dá esperança de que haja uma grande estrutura na linguagem. Terceiro, muitos n-gramas ocorrem muito raramente, o que torna a suavização de Laplace bastante inadequada para modelagem de linguagem. Em vez disso, usaremos modelos baseados em *deep learning*.

8.3.4 Leitura de Dados de Longa Sequência

Uma vez que os dados da sequência são, por sua própria natureza, sequenciais, precisamos abordar a questão de processá-la. Fizemos isso de uma maneira bastante ad-hoc em Section 8.1. Quando as sequências ficam muito longas para serem processadas por modelos tudas de uma vez, podemos desejar dividir essas sequências para leitura. Agora vamos descrever as estratégias gerais. Antes de apresentar o modelo, vamos supor que usaremos uma rede neural para treinar um modelo de linguagem, onde a rede processa um minibatch de sequências com comprimento predefinido, digamos n etapas de tempo, de cada vez. Agora, a questão é como ler minibatches de recursos e rótulos aleatoriamente.

Começando, uma vez que uma sequência de texto pode ser arbitrariamente longa, como todo o livro *The Time Machine*, podemos particionar uma sequência tão longa em subsequências com o mesmo número de etapas de tempo. Ao treinar nossa rede neural, um minibatch de tais subsequências será alimentado no modelo. Suponha que a rede processe uma subsequência de n passos de tempo de uma vez. Fig. 8.3.1 mostra todas as diferentes maneiras de obter subsequências de uma sequência de texto original, onde $n = 5$ e um token em cada etapa corresponde a um caractere. Observe que temos bastante liberdade, pois podemos escolher um deslocamento arbitrário que indica a posição inicial.

the time machine by h g wells

the time machine by h g wells

the time machine by h g wells

the time machine by h g wells

the time machine by h g wells

the time machine by h g wells

Fig. 8.3.1: Different offsets lead to different subsequences when splitting up text.

Portanto, qual devemos escolher de Fig. 8.3.1? Na verdade, todos eles são igualmente bons. No entanto, se escolhermos apenas um deslocamento, há cobertura limitada de todas as possíveis subsequências para treinar nossa rede. Portanto, podemos começar com um deslocamento aleatório para particionar uma sequência para obter *cobertura* e *aleatoriedade*. Na sequência, descrevemos como fazer isso para ambos *estratégias de amostragem aleatória* e *particionamento sequencial*.

Amostragem Aleatória

Na amostragem aleatória, cada exemplo é uma subsequência capturada arbitrariamente na longa sequência original. As subsequências de dois minibatches aleatórios adjacentes durante a iteração não são necessariamente adjacentes na sequência original. Para modelagem de linguagem, o objetivo é prever o próximo token com base nos tokens que vimos até agora, portanto, os rótulos são a sequência original, deslocada por um token.

O código a seguir gera aleatoriamente um minibatch dos dados a cada vez. Aqui, o argumento `batch_size` especifica o número de exemplos de subsequência em cada minibatch e `num_steps` é o número predefinido de etapas de tempo em cada subsequência.

```
def seq_data_iter_random(corpus, batch_size, num_steps):  #@save
    """Generate a minibatch of subsequences using random sampling."""
    # Start with a random offset (inclusive of `num_steps - 1`) to partition a
    # sequence
    corpus = corpus[random.randint(0, num_steps - 1):]
    # Subtract 1 since we need to account for labels
    num_subseqs = (len(corpus) - 1) // num_steps
    # The starting indices for subsequences of length `num_steps`
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))
    # In random sampling, the subsequences from two adjacent random
    # minibatches during iteration are not necessarily adjacent on the
    # original sequence
    random.shuffle(initial_indices)

    def data(pos):
        # Return a sequence of length `num_steps` starting from `pos`
        return corpus[pos: pos + num_steps]

    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
```

(continues on next page)

(continued from previous page)

```
        # Here, `initial_indices` contains randomized starting indices for
        # subsequences
        initial_indices_per_batch = initial_indices[i: i + batch_size]
        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield torch.tensor(X), torch.tensor(Y)
```

Vamos gerar manualmente uma sequência de 0 a 34. Nós assumimos que o tamanho do lote e o número de etapas de tempo são 2 e 5, respectivamente. Isso significa que podemos gerar $\lfloor (35 - 1)/5 \rfloor = 6$ pares de subsequências de rótulo de recurso. Com um tamanho de minibatch de 2, obtemos apenas 3 minibatches.

```
my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[23, 24, 25, 26, 27],
        [28, 29, 30, 31, 32]])
Y: tensor([[24, 25, 26, 27, 28],
        [29, 30, 31, 32, 33]])
X:  tensor([[18, 19, 20, 21, 22],
        [13, 14, 15, 16, 17]])
Y: tensor([[19, 20, 21, 22, 23],
        [14, 15, 16, 17, 18]])
X:  tensor([[ 3,  4,  5,  6,  7],
        [ 8,  9, 10, 11, 12]])
Y: tensor([[ 4,  5,  6,  7,  8],
        [ 9, 10, 11, 12, 13]])
```

Particionamento Sequencial

Além da amostragem aleatória da sequência original, também podemos garantir que as subsequências de dois minibatches adjacentes durante a iteração são adjacentes na sequência original. Essa estratégia preserva a ordem das subsequências divididas ao iterar em minibatches; portanto, é chamada de particionamento sequencial.

```
def seq_data_iter_sequential(corpus, batch_size, num_steps):  #@save
    """Generate a minibatch of subsequences using sequential partitioning."""
    # Start with a random offset to partition a sequence
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = torch.tensor(corpus[offset: offset + num_tokens])
    Ys = torch.tensor(corpus[offset + 1: offset + 1 + num_tokens])
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_steps * num_batches, num_steps):
        X = Xs[:, i: i + num_steps]
        Y = Ys[:, i: i + num_steps]
        yield X, Y
```

Usando as mesmas configurações, vamos imprimir as *features* `X` e os rótulos `Y` para cada minibatch

de subsequências lidas por particionamento sequencial. Observe que as subsequências de dois minibatches adjacentes durante a iteração são de fato adjacentes na sequência original.

```
for X, Y in seq_data_iter_sequential(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[ 5,  6,  7,  8,  9],
        [19, 20, 21, 22, 23]])
Y: tensor([[ 6,  7,  8,  9, 10],
        [20, 21, 22, 23, 24]])
X:  tensor([[10, 11, 12, 13, 14],
        [24, 25, 26, 27, 28]])
Y: tensor([[11, 12, 13, 14, 15],
        [25, 26, 27, 28, 29]])
```

Agora, agrupamos as duas funções de amostragem acima em uma classe para que possamos usá-la como um iterador de dados posteriormente.

```
class SeqDataLoader:  #@save
    """An iterator to load sequence data."""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            self.data_iter_fn = d2l.seq_data_iter_random
        else:
            self.data_iter_fn = d2l.seq_data_iter_sequential
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

Por último, definimos uma função `load_data_time_machine` que retorna o iterador de dados e o vocabulário, para que possamos usá-lo da mesma forma que outras funções com o prefixo `load_data`, como `d2l.load_data_fashion_mnist` definido em Section 3.5.

```
def load_data_time_machine(batch_size, num_steps,  #@save
                           use_random_iter=False, max_tokens=10000):
    """Return the iterator and the vocabulary of the time machine dataset."""
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

8.3.5 Resumo

- Modelos de linguagem são fundamentais para o processamento de linguagem natural.
- n-grams fornecem um modelo conveniente para lidar com sequências longas truncando a dependência.
- Sequências longas sofrem com o problema de ocorrerem muito raramente ou nunca.
- A lei de Zipf rege a distribuição de palavras não apenas para unigramas, mas também para os outros n-gramas.

- Há muita estrutura, mas não frequência suficiente para lidar com combinações de palavras infrequentes de forma eficiente por meio da suavização de Laplace.
- As principais opções para ler sequências longas são a amostragem aleatória e o particionamento sequencial. O último pode garantir que as subsequências de dois minibatches adjacentes durante a iteração sejam adjacentes à sequência original.

8.3.6 Exercícios

1. Suponha que haja 100.000 palavras no conjunto de dados de treinamento. Quanta frequência de palavra e frequência adjacente de várias palavras um quadrigrama precisa armazenar?
2. Como você modelaria um diálogo?
3. Estime o expoente da lei de Zipf para unigramas, bigramas e trigramas.
4. Em que outros métodos você pode pensar para ler dados de sequência longa?
5. Considere o deslocamento aleatório que usamos para ler sequências longas.
    1. Por que é uma boa ideia ter um deslocamento aleatório?
    2. Isso realmente leva a uma distribuição perfeitamente uniforme nas sequências do documento?
    3. O que você teria que fazer para tornar as coisas ainda mais uniformes?
6. Se quisermos que um exemplo de sequência seja uma frase completa, que tipo de problema isso apresenta na amostragem de minibatch? Como podemos resolver o problema?

Discussions[85]

8.4 Redes Neurais Recorrentes (RNNs)

Em Section 8.3 introduzimos modelos de n-gramas, onde a probabilidade condicional da palavra x_t no passo de tempo t depende apenas das $n-1$ palavras anteriores. Se quisermos incorporar o possível efeito de palavras anteriores ao passo de tempo $t-(n-1)$ em x_t, precisamos aumentar n. No entanto, o número de parâmetros do modelo também aumentaria exponencialmente com ele, pois precisamos armazenar $|\mathcal{V}|^n$ números para um conjunto de vocabulário $\mathcal{V}$. Portanto, em vez de modelar $P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$, é preferível usar um modelo de variável latente:

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}), \tag{8.4.1}$$

onde h_{t-1} é um *estado oculto* (também conhecido como uma variável oculta) que armazena as informações da sequência até o passo de tempo $t-1$. Em geral, o estado oculto em qualquer etapa t pode ser calculado com base na entrada atual x_t e no estado oculto anterior h_{t-1}:

$$h_t = f(x_t, h_{t-1}). \tag{8.4.2}$$

Para uma função suficientemente poderosa f em (8.4.2), o modelo de variável latente não é uma aproximação. Afinal, h_t pode simplesmente armazenar todos os dados que observou até agora. No entanto, isso pode tornar a computação e o armazenamento caros.

[85] https://discuss.d2l.ai/t/118

Lembre-se de que discutimos camadas ocultas com unidades ocultas em Chapter 4. É digno de nota que camadas ocultas e estados ocultos referem-se a dois conceitos muito diferentes. Camadas ocultas são, conforme explicado, camadas que ficam ocultas da visualização no caminho da entrada à saída. Estados ocultos são tecnicamente falando *entradas* para tudo o que fazemos em uma determinada etapa, e elas só podem ser calculadas observando os dados em etapas de tempo anteriores.

Redes neurais recorrentes (RNNs) são redes neurais com estados ocultos. Antes de introduzir o modelo RNN, primeiro revisitamos o modelo MLP introduzido em Section 4.1.

8.4.1 Redes Neurais sem Estados Ocultos

Vamos dar uma olhada em um MLP com uma única camada oculta. Deixe a função de ativação da camada oculta ser ϕ. Dado um minibatch de exemplos $\mathbf{X} \in \mathbb{R}^{n \times d}$ com tamanho de lote n e d entradas, a saída da camada oculta $\mathbf{H} \in \mathbb{R}^{n \times h}$ é calculada como

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{8.4.3}$$

Em (8.4.3), temos o parâmetro de peso $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, o parâmetro de polarização $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, e o número de unidades ocultas h, para a camada oculta. Assim, a transmissão (ver Section 2.1.3) é aplicada durante a soma. Em seguida, a variável oculta $\mathbf{H}$ é usada como entrada da camada de saída. A camada de saída é fornecida por

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{8.4.4}$$

onde $\mathbf{O} \in \mathbb{R}^{n \times q}$ é a variável de saída, $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ é o parâmetro de peso, e $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ é o parâmetro de polarização da camada de saída. Se for um problema de classificação, podemos usar $\text{softmax}(\mathbf{O})$ para calcular a distribuição de probabilidade das categorias de saída.

Isso é inteiramente análogo ao problema de regressão que resolvemos anteriormente em Section 8.1, portanto omitimos detalhes. Basta dizer que podemos escolher pares de rótulo de recurso aleatoriamente e aprender os parâmetros de nossa rede por meio de diferenciação automática e gradiente descendente estocástico.

8.4.2 Redes Neurais Recorrentes com Estados Ocultos

As coisas são totalmente diferentes quando temos estados ocultos. Vejamos a estrutura com mais detalhes.

Suponha que temos um minibatch de entradas $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ no passo de tempo t. Em outras palavras, para um minibatch de exemplos de sequência n, cada linha de $\mathbf{X}_t$ corresponde a um exemplo no passo de tempo t da sequência. Em seguida, denote por $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ a variável oculta do passo de tempo t. Ao contrário do MLP, aqui salvamos a variável oculta $\mathbf{H}_{t-1}$ da etapa de tempo anterior e introduzimos um novo parâmetro de peso $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ para descrever como usar a variável oculta da etapa de tempo anterior na etapa de tempo atual. Especificamente, o cálculo da variável oculta da etapa de tempo atual é determinado pela entrada da etapa de tempo atual junto com a variável oculta da etapa de tempo anterior:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{8.4.5}$$

Comparado com (8.4.3), (8.4.5) adiciona mais um termo $\mathbf{H}_{t-1} \mathbf{W}_{hh}$ e assim instancia (8.4.2). A partir da relação entre as variáveis ocultas $\mathbf{H}_t$ e $\mathbf{H}_{t-1}$ de etapas de tempo adjacentes, sabemos que

essas variáveis capturaram e retiveram as informações históricas da sequência até sua etapa de tempo atual, assim como o estado ou a memória da etapa de tempo atual da rede neural. Portanto, essa variável oculta é chamada de *estado oculto*. Visto que o estado oculto usa a mesma definição da etapa de tempo anterior na etapa de tempo atual, o cálculo de (8.4.5) é *recorrente*. Consequentemente, redes neurais com estados ocultos com base em cálculos recorrentes são nomeados *redes neurais recorrentes*. Camadas que fazem o cálculo de (8.4.5) em RNNs são chamadas de *camadas recorrentes*.

Existem muitas maneiras diferentes de construir RNNs. RNNs com um estado oculto definido por (8.4.5) são muito comuns. Para a etapa de tempo t, a saída da camada de saída é semelhante à computação no MLP:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{8.4.6}$$

Parâmetros do RNN incluem os pesos $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}, \mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, e o *bias* $\mathbf{b}_h \in \mathbb{R}^{1\times h}$ da camada oculta, junto com os pesos $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ e o *bias* $\mathbf{b}_q \in \mathbb{R}^{1\times q}$ da camada de saída. Vale a pena mencionar que mesmo em diferentes etapas de tempo, Os RNNs sempre usam esses parâmetros do modelo. Portanto, o custo de parametrização de um RNN não cresce à medida que o número de etapas de tempo aumenta.

Fig. 8.4.1 ilustra a lógica computacional de uma RNN em três etapas de tempo adjacentes. A qualquer momento, passo t, o cálculo do estado oculto pode ser tratado como: i) concatenar a entrada $\mathbf{X}_t$ na etapa de tempo atual t e o estado oculto $\mathbf{H}_{t-1}$ na etapa de tempo anterior $t-1$; ii) alimentar o resultado da concatenação em uma camada totalmente conectada com a função de ativação ϕ. A saída dessa camada totalmente conectada é o estado oculto $\mathbf{H}_t$ do intervalo de tempo atual t. Nesse caso, os parâmetros do modelo são a concatenação de $\mathbf{W}_{xh}$ e $\mathbf{W}_{hh}$, e um *bias* de $\mathbf{b}_h$, tudo de (8.4.5). O estado oculto do passo de tempo atual t, $\mathbf{H}_t$, participará do cálculo do estado oculto $\mathbf{H}_{t+1}$ do próximo passo de tempo $t+1$. Além disso, $\mathbf{H}_t$ também será alimentado na camada de saída totalmente conectada para calcular a saída $\mathbf{O}_t$ do passo de tempo atual t.

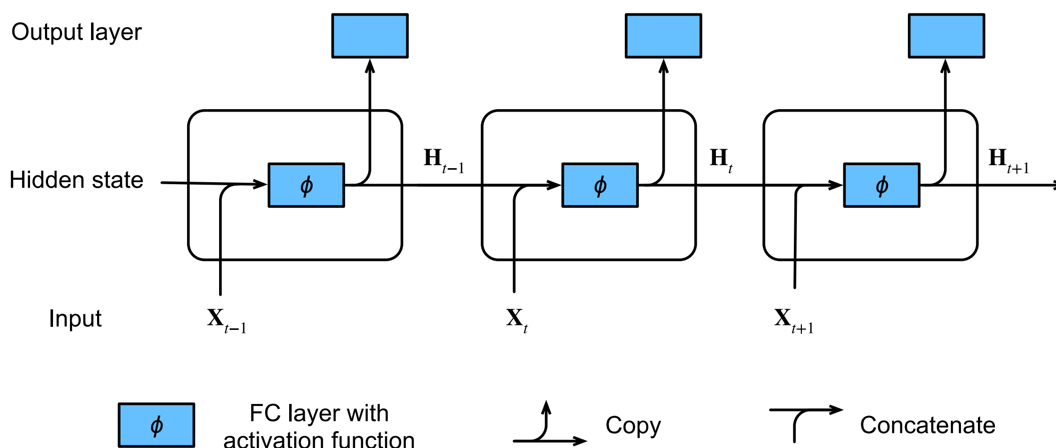


Fig. 8.4.1: Uma RNN com um estado oculto.

Acabamos de mencionar que o cálculo de $\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$ para o estado oculto é equivalente a multiplicação de matriz de concatenação de $\mathbf{X}_t$ and $\mathbf{H}_{t-1}$ e concatenação de $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$. Embora isso possa ser comprovado pela matemática, a seguir, apenas usamos um trecho de código simples para mostrar isso. Começando por, definir as matrizes `X`,`W_xh`, `H` e`W_hh`, cujas formas são (3, 1), (1, 4), (3, 4) e (4, 4), respectivamente. Multiplicando `X` por`W_xh`, e `H` por`W_hh`, respectivamente e, em seguida, adicionando essas duas multiplicações, obtemos uma matriz de forma (3, 4).

```
import torch
from d2l import torch as d2l
```

```
X, W_xh = torch.normal(0, 1, (3, 1)), torch.normal(0, 1, (1, 4))
H, W_hh = torch.normal(0, 1, (3, 4)), torch.normal(0, 1, (4, 4))
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

```
tensor([[ 0.9656, -0.2889,  1.2013,  0.3412],
        [ 2.0865, -2.5401,  1.2020,  3.0863],
        [-1.5326,  0.3890, -4.6161, -0.2931]])
```

Agora vamos concatenar as matrizes `X` e `H` ao longo das colunas (eixo 1), e as matrizes `W_xh` e `W_hh` ao longo das linhas (eixo 0). Essas duas concatenações resulta em matrizes de forma (3, 5) e da forma (5, 4), respectivamente. Multiplicando essas duas matrizes concatenadas, obtemos a mesma matriz de saída de forma (3, 4) como acima.

```
torch.matmul(torch.cat((X, H), 1), torch.cat((W_xh, W_hh), 0))
```

```
tensor([[ 0.9656, -0.2889,  1.2013,  0.3412],
        [ 2.0865, -2.5401,  1.2020,  3.0863],
        [-1.5326,  0.3890, -4.6161, -0.2931]])
```

8.4.3 Modelos de Linguagem em Nível de Caracteres Baseados em RNN

Lembre-se que para a modelagem de linguagem em Section 8.3, pretendemos prever o próximo token com base em os tokens atuais e passados, assim, mudamos a sequência original em um token como os rótulos. Bengio et al. propuseram primeiro usar uma rede neural para modelagem de linguagem (Bengio et al., 2003). A seguir, ilustramos como os RNNs podem ser usadas para construir um modelo de linguagem. Deixe o tamanho do minibatch ser um e a sequência do texto ser "máquina". Para simplificar o treinamento nas seções subsequentes, nós tokenizamos o texto em caracteres em vez de palavras e considere um *modelo de linguagem em nível de caractere*. Fig. 8.4.2 demonstra como prever o próximo caractere com base nos caracteres atuais e anteriores através de um RNN para modelagem de linguagem em nível de caractere.

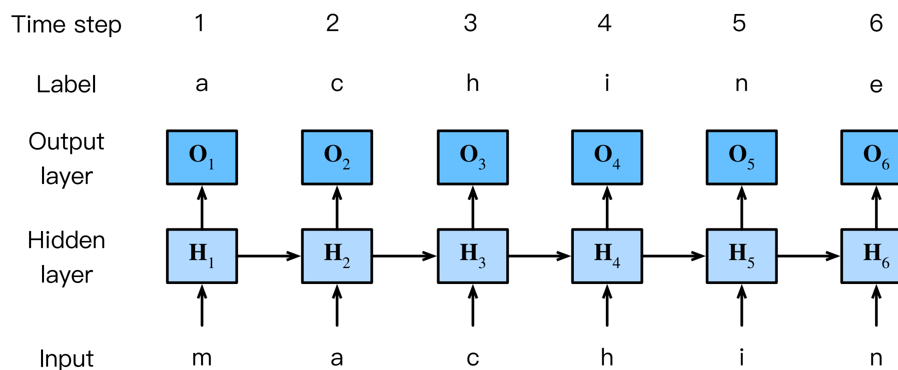


Fig. 8.4.2: Um modelo de linguagem de nível de caractere baseado no RNN. As sequências de entrada e rótulo são "machin" e "achine", respectivamente.

Durante o processo de treinamento, executamos uma operação softmax na saída da camada de saída para cada etapa de tempo e, em seguida, usamos a perda de entropia cruzada para calcular o erro entre a saída do modelo e o rótulo. Devido ao cálculo recorrente do estado oculto na camada oculta, a saída da etapa de tempo 3 em Fig. 8.4.2, $\mathbf{O}_3$, é determinada pela sequência de texto "m", "a" e "c". Como o próximo caractere da sequência nos dados de treinamento é "h", a perda de tempo da etapa 3 dependerá da distribuição de probabilidade do próximo caractere gerado com base na sequência de características "m", "a", "c" e o rótulo "h" desta etapa de tempo.

Na prática, cada token é representado por um vetor d-dimensional e usamos um tamanho de batch $n > 1$. Portanto, a entrada $ mathbf X_t $ no passo de tempo $ t $ será uma matriz $\mathbf{X}_t$, que é idêntica ao que discutimos em Section 8.4.2.

8.4.4 Perplexidade

Por último, vamos discutir sobre como medir a qualidade do modelo de linguagem, que será usado para avaliar nossos modelos baseados em RNN nas seções subsequentes. Uma maneira é verificar o quão surpreendente é o texto. Um bom modelo de linguagem é capaz de prever com tokens de alta precisão que veremos a seguir. Considere as seguintes continuações da frase "Está chovendo", conforme proposto por diferentes modelos de linguagem:

1. "Está chovendo lá fora"
2. "Está chovendo bananeira"
3. "Está chovendo piouw; kcj pwepoiut"

Em termos de qualidade, o exemplo 1 é claramente o melhor. As palavras são sensatas e logicamente coerentes. Embora possa não refletir com muita precisão qual palavra segue semanticamente ("em São Francisco" e "no inverno" seriam extensões perfeitamente razoáveis), o modelo é capaz de capturar qual tipo de palavra se segue. O exemplo 2 é consideravelmente pior ao produzir uma extensão sem sentido. No entanto, pelo menos o modelo aprendeu como soletrar palavras e algum grau de correlação entre as palavras. Por último, o exemplo 3 indica um modelo mal treinado que não ajusta os dados adequadamente.

Podemos medir a qualidade do modelo calculando a probabilidade da sequência. Infelizmente, esse é um número difícil de entender e difícil de comparar. Afinal, as sequências mais curtas têm muito mais probabilidade de ocorrer do que as mais longas, portanto, avaliando o modelo na magnum opus de Tolstoy *Guerra e paz* produzirá inevitavelmente uma probabilidade muito menor do que, digamos, na novela de Saint-Exupéry *O Pequeno Príncipe*. O que falta equivale a uma média.

A teoria da informação é útil aqui. Definimos entropia, surpresa e entropia cruzada quando introduzimos a regressão softmax (Section 3.4.7) e mais sobre a teoria da informação é discutido no apêndice online sobre teoria da informação[86]. Se quisermos compactar o texto, podemos perguntar sobre prever o próximo token dado o conjunto atual de tokens. Um modelo de linguagem melhor deve nos permitir prever o próximo token com mais precisão. Assim, deve permitir-nos gastar menos bits na compressão da sequência. Então, podemos medi-lo pela perda de entropia cruzada média sobre todos os n tokens de uma sequência:

$$\frac{1}{n} \sum_{t=1}^n -\log P(x_t \mid x_{t-1}, \ldots, x_1), \tag{8.4.7}$$

onde P é dado por um modelo de linguagem e x_t é o token real observado no passo de tempo t da sequência. Isso torna o desempenho em documentos de comprimentos diferentes comparáveis.

[86] https://d2l.ai/chapter_apencha-mathematics-for-deep-learning/information-theory.html

Por razões históricas, os cientistas do processamento de linguagem natural preferem usar uma quantidade chamada *perplexidade*. Em poucas palavras, é a exponencial de (8.4.7):

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t \mid x_{t-1},\ldots,x_1)\right). \tag{8.4.8}$$

A perplexidade pode ser melhor entendida como a média harmônica do número de escolhas reais que temos ao decidir qual ficha escolher a seguir. Vejamos alguns casos:

- No melhor cenário, o modelo sempre estima perfeitamente a probabilidade do token de rótulo como 1. Nesse caso, a perplexidade do modelo é 1.
- No pior cenário, o modelo sempre prevê a probabilidade do token de rótulo como 0. Nessa situação, a perplexidade é infinita positiva.
- Na linha de base, o modelo prevê uma distribuição uniforme de todos os tokens disponíveis do vocabulário. Nesse caso, a perplexidade é igual ao número de tokens exclusivos do vocabulário. Na verdade, se armazenássemos a sequência sem nenhuma compressão, seria o melhor que poderíamos fazer para codificá-la. Conseqüentemente, isso fornece um limite superior não trivial que qualquer modelo útil deve superar.

Nas seções a seguir, implementaremos RNNs para modelos de linguagem em nível de personagem e usaremos perplexidade para avaliar tais modelos.

8.4.5 Resumo

- Uma rede neural que usa computação recorrente para estados ocultos é chamada de rede neural recorrente (RNN).
- O estado oculto de uma RNN pode capturar informações históricas da sequência até o intervalo de tempo atual.
- O número de parâmetros do modelo RNN não aumenta com o aumento do número de etapas de tempo.
- Podemos criar modelos de linguagem em nível de caractere usando um RNN.
- Podemos usar a perplexidade para avaliar a qualidade dos modelos de linguagem.

8.4.6 Exercícios

1. Se usarmos uma RNN para prever o próximo caractere em uma sequência de texto, qual é a dimensão necessária para qualquer saída?
2. Por que as RNNs podem expressar a probabilidade condicional de um token em algum intervalo de tempo com base em todos os tokens anteriores na sequência de texto?
3. O que acontece com o gradiente se você retropropaga através de uma longa sequência?
4. Quais são alguns dos problemas associados ao modelo de linguagem descrito nesta seção?

Discussions[87]

[87] https://discuss.d2l.ai/t/1050

8.5 Implementação de Redes Neurais Recorrentes do Zero

Nesta seção, implementaremos uma RNN do zero para um modelo de linguagem de nível de personagem, de acordo com nossas descrições em Section 8.4. Tal modelo será treinado em H. G. Wells '* The Time Machine *. Como antes, começamos lendo o conjunto de dados primeiro, que é apresentado em Section 8.3.

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

8.5.1 Codificação One-Hot

Lembre-se de que cada token é representado como um índice numérico em `train_iter`. Alimentar esses índices diretamente para uma rede neural pode tornar difícil aprender. Frequentemente, representamos cada token como um vetor de *features* mais expressivo. A representação mais fácil é chamada de *codificação one-hot*, que é introduzida em Section 3.4.1.

Em resumo, mapeamos cada índice para um vetor de unidade diferente: suponha que o número de tokens diferentes no vocabulário seja N (`len (vocab)`) e os índices de token variam de 0 a $N-1$. Se o índice de um token é o inteiro i, então criamos um vetor de 0s com um comprimento de N e definimos o elemento na posição i como 1. Este vetor é o vetor one-hot do token original. Os vetores one-hot com índices 0 e 2 são mostrados abaixo.

```
F.one_hot(torch.tensor([0, 2]), len(vocab))
```

```
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0]])
```

A forma do minibatch que amostramos a cada vez é (tamanho do lote, número de etapas de tempo). A função `one_hot` transforma tal minibatch em um tensor tridimensional com a última dimensão igual ao tamanho do vocabulário (`len (vocab)`). Freqüentemente, transpomos a entrada para que possamos obter um saída de forma (número de etapas de tempo, tamanho do lote, tamanho do vocabulário). Isso nos permitirá mais convenientemente fazer um loop pela dimensão mais externa para atualizar os estados ocultos de um minibatch, passo a passo do tempo.

```
X = torch.arange(10).reshape((2, 5))
F.one_hot(X.T, 28).shape
```

```
torch.Size([5, 2, 28])
```

8.5.2 Inicializando os Parâmetros do Modelo

Em seguida, inicializamos os parâmetros do modelo para o modelo RNN. O número de unidades ocultas `num_hiddens` é um hiperparâmetro ajustável. Ao treinar modelos de linguagem, as entradas e saídas são do mesmo vocabulário. Portanto, eles têm a mesma dimensão, que é igual ao tamanho do vocabulário.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    # Hidden layer parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

8.5.3 Modelo RNN

Para definir um modelo RNN, primeiro precisamos de uma função `init_rnn_state` para retornar ao estado oculto na inicialização. Ele retorna um tensor preenchido com 0 e com uma forma de (tamanho do lote, número de unidades ocultas). O uso de tuplas torna mais fácil lidar com situações em que o estado oculto contém várias variáveis, que encontraremos em seções posteriores.

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

A seguinte função `rnn` define como calcular o estado oculto e a saída em uma etapa de tempo. Observe que o modelo RNN percorre a dimensão mais externa de `entradas` para que ela atualize os estados ocultos `H` de um minibatch, passo a passo do tempo. Além do mais, a função de ativação aqui usa a função tanh. Como descrito em Section 4.1, o o valor médio da função tanh é 0, quando os elementos são uniformemente distribuídos sobre os números reais.

```
def rnn(inputs, state, params):
    # Here `inputs` shape: (`num_steps`, `batch_size`, `vocab_size`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # Shape of `X`: (`batch_size`, `vocab_size`)
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

Com todas as funções necessárias sendo definidas, em seguida, criamos uma classe para envolver essas funções e armazenar parâmetros para um modelo RNN implementado do zero.

```
class RNNModelScratch: #@save
    """A RNN Model implemented from scratch."""
    def __init__(self, vocab_size, num_hiddens, device,
                 get_params, init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return self.init_state(batch_size, self.num_hiddens, device)
```

Vamos verificar se as saídas têm as formas corretas, por exemplo, para garantir que a dimensionalidade do estado oculto permaneça inalterada.

```
num_hiddens = 512
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0], d2l.try_gpu())
Y, new_state = net(X.to(d2l.try_gpu()), state)
Y.shape, len(new_state), new_state[0].shape
```

```
(torch.Size([10, 28]), 1, torch.Size([2, 512]))
```

Podemos ver que a forma de saída é (número de etapas de tempo $\times$ tamanho do lote, tamanho do vocabulário), enquanto a forma do estado oculto permanece a mesma, ou seja, (tamanho do lote, número de unidades ocultas).

8.5.4 Predição

Vamos primeiro definir a função de predição para gerar novos personagens seguindo o `prefixo` fornecido pelo usuário, que é uma string contendo vários caracteres. Ao percorrer esses caracteres iniciais em `prefixo`, continuamos passando pelo estado escondido para a próxima etapa sem gerando qualquer saída. Isso é chamado de período de *aquecimento*, durante o qual o modelo se atualiza (por exemplo, atualizar o estado oculto) mas não faz previsões. Após o período de aquecimento, o estado oculto é geralmente melhor do que seu valor inicializado no início. Assim, geramos os caracteres previstos e os emitimos.

```
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """Generate new characters following the `prefix`."""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape((1, 1))
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
```

(continues on next page)

(continued from previous page)

```
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

Agora podemos testar a função `predict_ch8`. Especificamos o prefixo como `viajante do tempo` e fazemos com que ele gere 10 caracteres adicionais. Visto que não treinamos a rede, isso vai gerar previsões sem sentido.

```
predict_ch8('time traveller ', 10, net, vocab, d2l.try_gpu())
```

```
'time traveller ufcr ufcr '
```

8.5.5 Recorte de Gradiente

Para uma sequência de comprimento T, calculamos os gradientes ao longo desses T passos de tempo em uma iteração, que resulta em uma cadeia de produtos-matriz com comprimento $\mathcal{O}(T)$ durante a retropropagação. Conforme mencionado em Section 4.8, pode resultar em instabilidade numérica, por exemplo, os gradientes podem explodir ou desaparecer, quando T é grande. Portanto, os modelos RNN geralmente precisam de ajuda extra para estabilizar o treinamento.

De um modo geral, ao resolver um problema de otimização, executamos etapas de atualização para o parâmetro do modelo, diga na forma vetorial $\mathbf{x}$, na direção do gradiente negativo $\mathbf{g}$ em um minibatch. Por exemplo, com $\eta > 0$ como a taxa de aprendizagem, em uma iteração nós atualizamos $\mathbf{x}$ como $\mathbf{x} - \eta \mathbf{g}$. Vamos supor ainda que a função objetivo f é bem comportada, digamos, *Lipschitz contínuo* com L constante. Quer dizer, para qualquer $\mathbf{x}$ e $\mathbf{y}$, temos

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L \|\mathbf{x} - \mathbf{y}\|. \tag{8.5.1}$$

Neste caso, podemos assumir com segurança que, se atualizarmos o vetor de parâmetro por $\eta \mathbf{g}$, então

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L \eta\|\mathbf{g}\|, \tag{8.5.2}$$

o que significa que não observaremos uma mudança de mais de $L \eta\|\mathbf{g}\|$. Isso é uma maldição e uma bênção. Do lado da maldição, limita a velocidade de progresso; enquanto do lado da bênção, limita até que ponto as coisas podem dar errado se seguirmos na direção errada.

Às vezes, os gradientes podem ser muito grandes e o algoritmo de otimização pode falhar em convergir. Poderíamos resolver isso reduzindo a taxa de aprendizado η. Mas e se nós *raramente* obtivermos gradientes grandes? Nesse caso, essa abordagem pode parecer totalmente injustificada. Uma alternativa popular é cortar o gradiente $\mathbf{g}$ projetando-o de volta para uma bola de um determinado raio, digamos θ via

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right) \mathbf{g}. \tag{8.5.3}$$

Fazendo isso, sabemos que a norma do gradiente nunca excede θ e que o gradiente atualizado é totalmente alinhado com a direção original de $\mathbf{g}$. Também tem o efeito colateral desejável de limitar a influência de qualquer minibatch (e dentro dele qualquer amostra dada) pode exercer

sobre o vetor de parâmetro. Isto confere certo grau de robustez ao modelo. O recorte de gradiente fornece uma solução rápida para a explosão do gradiente. Embora não resolva totalmente o problema, é uma das muitas técnicas para aliviá-lo.

Abaixo, definimos uma função para cortar os gradientes de um modelo que é implementado do zero ou um modelo construído pelas APIs de alto nível. Observe também que calculamos a norma do gradiente em todos os parâmetros do modelo.

```
def grad_clipping(net, theta):  #@save
    """Clip the gradient."""
    if isinstance(net, nn.Module):
        params = [p for p in net.parameters() if p.requires_grad]
    else:
        params = net.params
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

8.5.6 Treinamento

Antes de treinar o modelo, vamos definir uma função para treinar o modelo em uma época. É diferente de como treinamos o modelo de: Section 3.6 em três lugares:

1. Diferentes métodos de amostragem para dados sequenciais (amostragem aleatória e particionamento sequencial) resultarão em diferenças na inicialização de estados ocultos.
2. Cortamos os gradientes antes de atualizar os parâmetros do modelo. Isso garante que o modelo não diverge, mesmo quando os gradientes explodem em algum ponto durante o processo de treinamento.
3. Usamos perplexidade para avaliar o modelo. Conforme discutido em Section 8.4.4, isso garante que as sequências de comprimentos diferentes sejam comparáveis.

Especificamente, quando o particionamento sequencial é usado, inicializamos o estado oculto apenas no início de cada época. Uma vez que o exemplo de subsequência i^th no próximo minibatch é adjacente ao exemplo de subsequência i^th atual, o estado oculto no final do minibatch atual será usado para inicializar o estado oculto no início do próximo minibatch. Nesse caminho, informação histórica da sequência armazenado no estado oculto pode fluir em subsequências adjacentes dentro de uma época. No entanto, o cálculo do estado oculto em qualquer ponto depende de todos os minibatches anteriores na mesma época, o que complica o cálculo do gradiente. Para reduzir o custo computacional, destacamos o gradiente antes de processar qualquer minibatch de modo que o cálculo do gradiente do estado oculto é sempre limitado aos passos de tempo em um minibatch.

Ao usar a amostragem aleatória, precisamos reinicializar o estado oculto para cada iteração, uma vez que cada exemplo é amostrado com uma posição aleatória. Igual à função `train_epoch_ch3` em Section 3.6, `atualizador` é uma função geral para atualizar os parâmetros do modelo. Pode ser a função `d2l.sgd` implementada do zero ou a função de otimização integrada em uma estrutura de aprendizagem profunda.

```
#@save
def train_epoch_ch8(net, train_iter, loss, updater, device, use_random_iter):
```

(continues on next page)

(continued from previous page)

```
    """Train a net within one epoch (defined in Chapter 8)."""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # Sum of training loss, no. of tokens
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # Initialize `state` when either it is the first iteration or
            # using random sampling
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                # `state` is a tensor for `nn.GRU`
                state.detach_()
            else:
                # `state` is a tuple of tensors for `nn.LSTM` and
                # for our custom scratch implementation
                for s in state:
                    s.detach_()
        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)
        l = loss(y_hat, y.long()).mean()
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            updater.step()
        else:
            l.backward()
            grad_clipping(net, 1)
            # Since the `mean` function has been invoked
            updater(batch_size=1)
        metric.add(l * y.numel(), y.numel())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

A função de treinamento suporta um modelo RNN implementado ou do zero ou usando APIs de alto nível.

```
#@save
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """Train a model (defined in Chapter 8)."""
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    # Initialize
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
    # Train and predict
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(
            net, train_iter, loss, updater, device, use_random_iter)
```

(continues on next page)

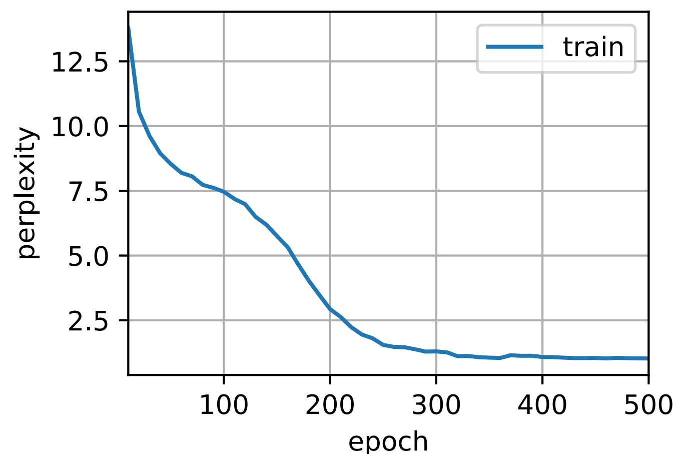
(continued from previous page)

```
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch + 1, [ppl])
    print(f'perplexity {ppl:.1f}, {speed:.1f} tokens/sec on {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

Agora podemos treinar o modelo RNN. Como usamos apenas 10.000 tokens no conjunto de dados, o modelo precisa de mais épocas para convergir melhor.

```
num_epochs, lr = 500, 1
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu())
```

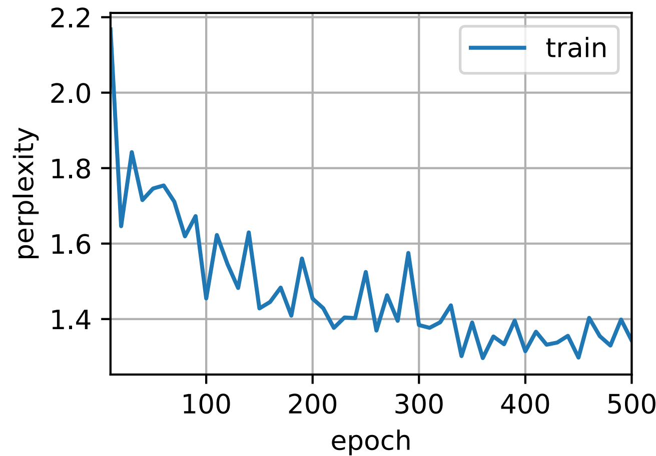
```
perplexity 1.0, 62511.3 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller with a slight accession ofcheerfulness really thi
```



Finalmente, vamos verificar os resultados do uso do método de amostragem aleatória.

```
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu(),
          use_random_iter=True)
```

```
perplexity 1.3, 59672.6 tokens/sec on cuda:0
time travellerit s against reason said filbywhat os have been at
travellerit s against reason said filbywhat os have been at
```



Embora a implementação do modelo RNN acima do zero seja instrutiva, não é conveniente. Na próxima seção, veremos como melhorar o modelo RNN, por exemplo, como torná-lo mais fácil de implementar e fazê-lo funcionar mais rápido.

8.5.7 Resumo

- Podemos treinar um modelo de linguagem de nível de caractere baseado em RNN para gerar texto seguindo o prefixo de texto fornecido pelo usuário.
- Um modelo de linguagem RNN simples consiste em codificação de entrada, modelagem RNN e geração de saída.
- Os modelos RNN precisam de inicialização de estado para treinamento, embora a amostragem aleatória e o particionamento sequencial usem maneiras diferentes.
- Ao usar o particionamento sequencial, precisamos separar o gradiente para reduzir o custo computacional.
- Um período de aquecimento permite que um modelo se atualize (por exemplo, obtenha um estado oculto melhor do que seu valor inicializado) antes de fazer qualquer previsão.
- O recorte de gradiente evita a explosão do gradiente, mas não pode corrigir gradientes que desaparecem.

8.5.8 Exercícios

1. Mostre que a codificação one-hot é equivalente a escolher uma incorporação diferente para cada objeto.
2. Ajuste os hiperparâmetros (por exemplo, número de épocas, número de unidades ocultas, número de etapas de tempo em um minibatch e taxa de aprendizado) para melhorar a perplexidade.
    - Quão baixo você pode ir?
    - Substitua a codificação one-hot por *embeddings* que podem ser aprendidos. Isso leva a um melhor desempenho?

- Será que funcionará bem em outros livros de H. G. Wells, por exemplo, The War of the Worlds[88]?

3. Modifique a função de previsão para usar amostragem em vez de escolher o próximo caractere mais provável.
   - O que acontece?
   - Desvie o modelo para resultados mais prováveis, por exemplo, amostrando de $q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha$ para $\alpha > 1$.
4. Execute o código nesta seção sem cortar o gradiente. O que acontece?
5. Altere o particionamento sequencial para que não separe os estados ocultos do gráfico computacional. O tempo de execução muda? Que tal a perplexidade?
6. Substitua a função de ativação usada nesta seção por ReLU e repita os experimentos nesta seção. Ainda precisamos de recorte de gradiente? Por quê?

Discussions[89]

8.6 Implementação Concisa de Redes Neurais Recorrentes

Embora Section 8.5 tenha sido instrutivo para ver como RNNs são implementados, isso não é conveniente ou rápido. Esta seção mostrará como implementar o mesmo modelo de linguagem de forma mais eficiente usando funções fornecidas por APIs de alto nível de uma estrutura de aprendizado profundo. Começamos como antes, lendo o conjunto de dados da máquina do tempo.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

8.6.1 Definindo o Modelo

APIs de alto nível fornecem implementações de redes neurais recorrentes. Construímos a camada de rede neural recorrente `rnn_layer` com uma única camada oculta e 256 unidades ocultas. Na verdade, ainda não discutimos o que significa ter várias camadas — isso vai acontecer em Section 9.3. Por enquanto, basta dizer que várias camadas simplesmente equivalem à saída de uma camada de RNN sendo usada como entrada para a próxima camada de RNN.

```
num_hiddens = 256
rnn_layer = nn.RNN(len(vocab), num_hiddens)
```

Usamos um tensor para inicializar o estado oculto, cuja forma é (número de camadas ocultas, tamanho do lote, número de unidades ocultas).

[88] http://www.gutenberg.org/ebooks/36
[89] https://discuss.d2l.ai/t/486

```
state = torch.zeros((1, batch_size, num_hiddens))
state.shape
```

```
torch.Size([1, 32, 256])
```

Com um estado oculto e uma entrada, podemos calcular a saída com o estado oculto atualizado. Deve ser enfatizado que a "saída" (Y) dernn_layer *não* envolve computação de camadas de saída: isso se refere a o estado oculto em *cada* passo de tempo, e eles podem ser usados como entrada para a camada de saída subsequente.

```
X = torch.rand(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, state_new.shape
```

```
(torch.Size([35, 32, 256]), torch.Size([1, 32, 256]))
```

Semelhante a Section 8.5, nós definimos uma classe RNNModel para um modelo RNN completo. Observe que rnn_layer contém apenas as camadas recorrentes ocultas, precisamos criar uma camada de saída separada.

```
#@save
class RNNModel(nn.Module):
    """The RNN model."""
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.num_hiddens = self.rnn.hidden_size
        # If the RNN is bidirectional (to be introduced later),
        # `num_directions` should be 2, else it should be 1.
        if not self.rnn.bidirectional:
            self.num_directions = 1
            self.linear = nn.Linear(self.num_hiddens, self.vocab_size)
        else:
            self.num_directions = 2
            self.linear = nn.Linear(self.num_hiddens * 2, self.vocab_size)

    def forward(self, inputs, state):
        X = F.one_hot(inputs.T.long(), self.vocab_size)
        X = X.to(torch.float32)
        Y, state = self.rnn(X, state)
        # The fully connected layer will first change the shape of `Y` to
        # (`num_steps` * `batch_size`, `num_hiddens`). Its output shape is
        # (`num_steps` * `batch_size`, `vocab_size`).
        output = self.linear(Y.reshape((-1, Y.shape[-1])))
        return output, state

    def begin_state(self, device, batch_size=1):
        if not isinstance(self.rnn, nn.LSTM):
            # `nn.GRU` takes a tensor as hidden state
            return  torch.zeros((self.num_directions * self.rnn.num_layers,
                                 batch_size, self.num_hiddens),
                                device=device)
```

(continues on next page)

(continued from previous page)

```
        else:
            # `nn.LSTM` takes a tuple of hidden states
            return (torch.zeros((
                self.num_directions * self.rnn.num_layers,
                batch_size, self.num_hiddens), device=device),
                    torch.zeros((
                        self.num_directions * self.rnn.num_layers,
                        batch_size, self.num_hiddens), device=device))
```

8.6.2 Treinamento e Previsão

Antes de treinar o modelo, façamos uma previsão com um modelo que possui pesos aleatórios.

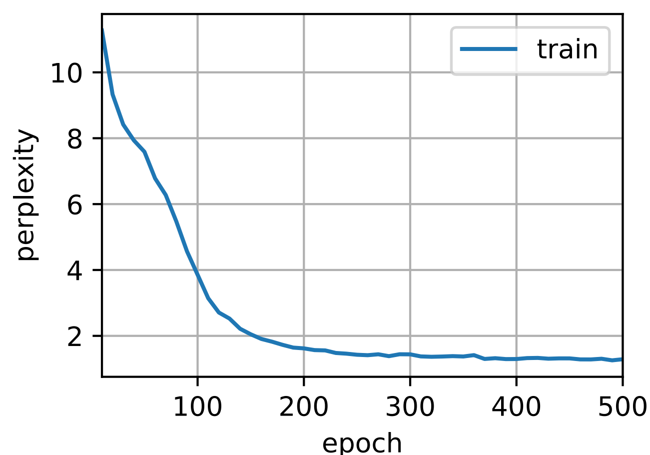
```
device = d2l.try_gpu()
net = RNNModel(rnn_layer, vocab_size=len(vocab))
net = net.to(device)
d2l.predict_ch8('time traveller', 10, net, vocab, device)
```

```
'time traveller<unk>j<unk>j<unk>j<unk>j<unk>j'
```

Como é bastante óbvio, este modelo não funciona. Em seguida, chamamos `train_ch8` com os mesmos hiperparâmetros definidos em Section 8.5 e treinamos nosso modelo com APIs de alto nível.

```
num_epochs, lr = 500, 1
d2l.train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.3, 296232.7 tokens/sec on cuda:0
time travellerif f uplyond ur very ctispsoubstorne filburmy the
travelleryou can spon whis asmered the provincill sorkent m
```



Comparado com a última seção, este modelo atinge perplexidade comparável, embora dentro de um período de tempo mais curto, devido ao código ser mais otimizado por APIs de alto nível da estrutura de aprendizado profundo.

8.6.3 Resumo

- APIs de alto nível da estrutura de *deep learning* fornecem uma implementação da camada RNN.
- A camada RNN de APIs de alto nível retorna uma saída e um estado oculto atualizado, em que a saída não envolve computação da camada de saída.
- Usar APIs de alto nível leva a um treinamento RNN mais rápido do que usar sua implementação do zero.

8.6.4 Exercícios

1. Você pode fazer o modelo RNN sobreajustar usando as APIs de alto nível?
2. O que acontece se você aumentar o número de camadas ocultas no modelo RNN? Você pode fazer o modelo funcionar?
3. Implemente o modelo autoregressivo de Section 8.1 usando um RNN.

Discussions[90]

8.7 Retropropagação ao Longo do Tempo

Até agora, temos repetidamente aludido a coisas como *gradientes explosivos*, *gradientes de desaparecimento*, e a necessidade de *destacar o gradiente* para RNNs. Por exemplo, em Section 8.5 invocamos a função `detach` na sequência. Nada disso foi realmente completamente explicado, no interesse de ser capaz de construir um modelo rapidamente e para ver como funciona. Nesta secção, vamos nos aprofundar um pouco mais nos detalhes de retropropagação para modelos de sequência e por que (e como) a matemática funciona.

Encontramos alguns dos efeitos da explosão de gradiente quando primeiro RNNs implementados (Section 8.5). No especial, se você resolveu os exercícios, você poderia ter visto que o corte de gradiente é vital para garantir convergência. Para fornecer uma melhor compreensão deste problema, esta seção irá rever como os gradientes são calculados para modelos de sequência. Observe que não há nada conceitualmente novo em como funciona. Afinal, ainda estamos apenas aplicando a regra da cadeia para calcular gradientes. No entanto, vale a pena revisar a retropropagação (Section 4.7) novamente.

Descrevemos propagações para frente e para trás e gráficos computacionais em MLPs em Section 4.7. A propagação direta em uma RNN é relativamente para a frente. *Retropropagação através do tempo* é, na verdade, uma aplicação específica de retropropagação em RNNs (Werbos, 1990). Isto exige que expandamos o gráfico computacional de uma RNN um passo de cada vez para obter as dependências entre variáveis e parâmetros do modelo. Então, com base na regra da cadeia, aplicamos retropropagação para calcular e gradientes de loja. Uma vez que as sequências podem ser bastante longas, a dependência pode ser bastante longa. Por exemplo, para uma sequência de 1000 caracteres, o primeiro token pode ter uma influência significativa sobre o token na posição final. Isso não é realmente viável computacionalmente (leva muito tempo e requer muita memória) e requer mais de 1000 produtos de matriz antes de chegarmos a esse gradiente muito indescritível. Este é um processo repleto de incertezas computacionais e estatísticas. A seguir iremos elucidar o que acontece e como resolver isso na prática.

[90] https://discuss.d2l.ai/t/1053

8.7.1 Análise de Gradientes em RNNs

Começamos com um modelo simplificado de como funciona uma RNN. Este modelo ignora detalhes sobre as especificações do estado oculto e como ele é atualizado. A notação matemática aqui não distingue explicitamente escalares, vetores e matrizes como costumava fazer. Esses detalhes são irrelevantes para a análise e serviriam apenas para bagunçar a notação nesta subseção.

Neste modelo simplificado, denotamos h_t como o estado oculto, x_t como a entrada e o_t como a saída no passo de tempo t. Lembre-se de nossas discussões em Section 8.4.2 que a entrada e o estado oculto podem ser concatenados ao serem multiplicados por uma variável de peso na camada oculta. Assim, usamos w_h e w_o para indicar os pesos da camada oculta e da camada de saída, respectivamente. Como resultado, os estados ocultos e saídas em cada etapa de tempo podem ser explicados como

$$\begin{aligned}h_t &= f(x_t, h_{t-1}, w_h),\\o_t &= g(h_t, w_o),\end{aligned} \tag{8.7.1}$$

onde f e g são transformações da camada oculta e da camada de saída, respectivamente. Portanto, temos uma cadeia de valores $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ que dependem uns dos outros por meio de computação recorrente. A propagação direta é bastante direta. Tudo o que precisamos é percorrer as triplas (x_t, h_t, o_t) um passo de tempo de cada vez. A discrepância entre a saída o_t e o rótulo desejado y_t é então avaliada por uma função objetivo em todas as etapas de tempo T como

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^T l(y_t, o_t). \tag{8.7.2}$$

Para retropropagação, as coisas são um pouco mais complicadas, especialmente quando calculamos os gradientes em relação aos parâmetros w_h da função objetivo L. Para ser específico, pela regra da cadeia,

$$\begin{aligned}\frac{\partial L}{\partial w_h} &= \frac{1}{T}\sum_{t=1}^T \frac{\partial l(y_t, o_t)}{\partial w_h}\\&= \frac{1}{T}\sum_{t=1}^T \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial g(h_t, w_o)}{\partial h_t} \frac{\partial h_t}{\partial w_h}.\end{aligned} \tag{8.7.3}$$

O primeiro e o segundo fatores do produto em (8.7.3) são fáceis de calcular. O terceiro fator $\partial h_t/\partial w_h$ é onde as coisas ficam complicadas, já que precisamos calcular recorrentemente o efeito do parâmetro w_h em h_t. De acordo com o cálculo recorrente em (8.7.1), h_t depende de h_{t-1} e w_h, onde cálculo de h_{t-1} também depende de w_h. Assim, usando a regra da cadeia temos

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{8.7.4}$$

Para derivar o gradiente acima, suponha que temos três sequências $\{a_t\}, \{b_t\}, \{c_t\}$ satisfatória $a_0 = 0$ and $a_t = b_t + c_t a_{t-1}$ for $t = 1, 2, \ldots$. Então, para $t \geq 1$, é fácil mostrar

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^t c_j\right) b_i. \tag{8.7.5}$$

Substituindo $a_t, b_t, e c_t de acordo com

$$\begin{aligned}a_t &= \frac{\partial h_t}{\partial w_h},\\ b_t &= \frac{\partial f(x_{t},h_{t-1},w_h)}{\partial w_h}, \\ c_t &= \frac{\partial f(x_{t},h_{t-1},w_h)}{\partial h_{t-1}},\end{aligned} \tag{8.7.6}$$

o cálculo do gradiente em: eqref: `eq_bptt_partial_ht_wh_recur` satisfaz $a_{t}=b_{t}+c_{t}a_{t-1}$. Assim, por (8.7.5), podemos remover o cálculo recorrente em (8.7.4) com

$$\frac{\partial h_t}{\partial w_h}=\frac{\partial f(x_{t},h_{t-1},w_h)}{\partial w_h}+\sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} \frac{\partial f(x_{j},h_{j-1},w_h)}{\partial h_{j-1}} \right) \frac{\partial f(x_{i},h_{i-1},w_h)}{\partial w_h}. \tag{8.7.7}$$

Embora possamos usar a regra da cadeia para calcular **:raw-latex:`\partial `h_t/:raw-latex:`**partial *w_h$ recursivamente, esta cadeia pode ficar muito longa sempre que :math:`*t* for grande. Vamos discutir uma série de estratégias para lidar com esse problema.

Computação Completa

Obviamente, podemos apenas calcular a soma total em (8.7.7). Porém, isso é muito lento e os gradientes podem explodir, uma vez que mudanças sutis nas condições iniciais podem afetar muito o resultado. Ou seja, poderíamos ver coisas semelhantes ao efeito borboleta, em que mudanças mínimas nas condições iniciais levam a mudanças desproporcionais no resultado. Na verdade, isso é bastante indesejável em termos do modelo que queremos estimar. Afinal, estamos procurando estimadores robustos que generalizem bem. Portanto, essa estratégia quase nunca é usada na prática.

Truncamento de Etapas de Tempo

Alternativamente, podemos truncar a soma em (8.7.7) após τ passos. Isso é o que estivemos discutindo até agora, como quando separamos os gradientes em Section 8.5. Isso leva a uma *aproximação* do gradiente verdadeiro, simplesmente terminando a soma em $\partial h_{t-\tau}/\partial w_h$. Na prática, isso funciona muito bem. É o que é comumente referido como retropropagação truncada ao longo do tempo (Jaeger, 2002). Uma das consequências disso é que o modelo se concentra principalmente na influência de curto prazo, e não nas consequências de longo prazo. Na verdade, isso é *desejável*, pois inclina a estimativa para modelos mais simples e estáveis.

Truncamento Randomizado

Por último, podemos substituir $\partial h_t/\partial w_h$ por uma variável aleatória que está correta na expectativa, mas trunca a sequência. Isso é conseguido usando uma sequência de ξ_t com $0 \leq \pi_t \leq 1$ predefinido, onde $P(\xi_t = 0) = 1-\pi_t$ e $P(\xi_t = \pi_t^{-1}) = \pi_t$, portanto $E[\xi_t] = 1$. Usamos isso para substituir o gradiente $\partial h_t/\partial w_h$ em (8.7.4) com

$$z_t= \frac{\partial f(x_{t},h_{t-1},w_h)}{\partial w_h} +\xi_t \frac{\partial f(x_{t},h_{t-1},w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{8.7.8}$$

Segue da definição de ξ_t that $E[z_t] = \partial h_t/\partial w_h$. Sempre que $\xi_t = 0$ o cálculo recorrente termina nesse momento no passo t. Isso leva a uma soma ponderada de sequências de comprimentos variados, em que sequências longas são raras, mas apropriadamente sobrecarregadas. Esta ideia foi proposta por Tallec e Ollivier (Tallec & Ollivier, 2017).

Comparando Estratégias

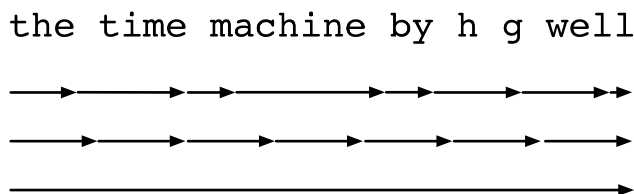


Fig. 8.7.1: Comparando estratégias para computar gradientes em RNNs. De cima para baixo: truncamento aleatório, truncamento regular e computação completa.

Fig. 8.7.1 ilustra as três estratégias ao analisar os primeiros caracteres do livro *The Time Machine* usando retropropagação através do tempo para RNNs:

- A primeira linha é o truncamento aleatório que divide o texto em segmentos de comprimentos variados.
- A segunda linha é o truncamento regular que divide o texto em subsequências do mesmo comprimento. Isso é o que temos feito em experimentos RNN.
- A terceira linha é a retropropagação completa ao longo do tempo que leva a uma expressão computacionalmente inviável.

Infelizmente, embora seja atraente em teoria, o truncamento aleatório não funciona muito melhor do que o truncamento regular, provavelmente devido a uma série de fatores. Primeiro, o efeito de uma observação após várias etapas de retropropagação no passado é suficiente para capturar dependências na prática. Segundo, o aumento da variância neutraliza o fato de que o gradiente é mais preciso com mais etapas. Terceiro, nós realmente *queremos* modelos que tenham apenas um curto intervalo de interações. Conseqüentemente, a retropropagação regularmente truncada ao longo do tempo tem um leve efeito de regularização que pode ser desejável.

8.7.2 Retropropagação ao Longo do Tempo em Detalhes

Depois de discutir o princípio geral, vamos discutir a retropropagação ao longo do tempo em detalhes. Diferente da análise em Section 8.7.1, na sequência vamos mostrar como calcular os gradientes da função objetivo com respeito a todos os parâmetros do modelo decomposto. Para manter as coisas simples, consideramos uma RNN sem parâmetros de polarização, cuja função de ativação na camada oculta usa o mapeamento de identidade ($\phi(x) = x$). Para a etapa de tempo t, deixe a entrada de exemplo único e o rótulo ser $\$\mathbf{x}_t \in \mathbb{R}^d$ and y_t, respectivamente. O estado oculto $\mathbf{h}_t \in \mathbb{R}^h$ e a saída $\mathbf{o}_t \in \mathbb{R}^q$ são computados como

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},\\ \mathbf{o}_t &= \mathbf{W}_{qh}\mathbf{h}_t,\end{aligned} \tag{8.7.9}$$

onde $\mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, e $\mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$ são os parâmetros de peso. Denotar por $l(\mathbf{o}_t, y_t)$ a perda na etapa de tempo t. Nossa função objetivo, a perda em etapas de tempo de T desde o início da sequência é assim

$$L = \frac{1}{T} \sum_{t=1}^{T} l(\mathbf{o}_t, y_t). \tag{8.7.10}$$

A fim de visualizar as dependências entre variáveis e parâmetros do modelo durante o cálculo do RNN, podemos desenhar um gráfico computacional para o modelo, como mostrado em Fig. 8.7.2. Por exemplo, o cálculo dos estados ocultos do passo de tempo 3, $\mathbf{h}_3$, depende dos parâmetros do modelo $\mathbf{W}_{hx}$ e $\mathbf{W}_{hh}$, o estado oculto da última etapa de tempo $\mathbf{h}_2$, e a entrada do intervalo de tempo atual $\mathbf{x}_3$.

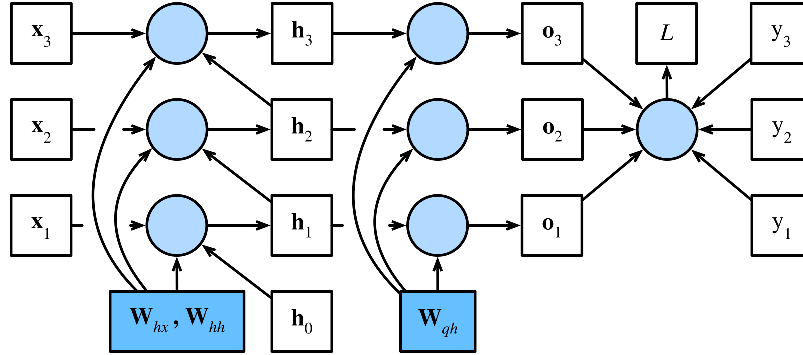


Fig. 8.7.2: Gráfico computacional mostrando dependências para um modelo RNN com três intervalos de tempo. Caixas representam variáveis (não sombreadas) ou parâmetros (sombreados) e círculos representam operadores.

Como acabamos de mencionar, os parâmetros do modelo em Fig. 8.7.2 são $\mathbf{W}_{hx}$, $\mathbf{W}_{hh}$, e $\mathbf{W}_{qh}$. Geralmente, treinar este modelo requer cálculo de gradiente em relação a esses parâmetros $\partial L/\partial \mathbf{W}_{hx}$, $\partial L/\partial \mathbf{W}_{hh}$, e $\partial L/\partial \mathbf{W}_{qh}$. De acordo com as dependências em Fig. 8.7.2, nós podemos atravessar na direção oposta das setas para calcular e armazenar os gradientes por sua vez. Para expressar de forma flexível a multiplicação de matrizes, vetores e escalares de diferentes formas na regra da cadeia, nós continuamos a usar o operador prod conforme descrito em Section 4.7.

Em primeiro lugar, diferenciando a função objetivo com relação à saída do modelo a qualquer momento, etapa t é bastante simples:

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial l(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t} \in \mathbb{R}^q. \tag{8.7.11}$$

Agora, podemos calcular o gradiente da função objetivo em relação ao parâmetro $\mathbf{W}_{qh}$ na camada de saída: $\partial L/\partial \mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$. Com base em Fig. 8.7.2, a função objetivo L depende de $\mathbf{W}_{qh}$ via $\mathbf{o}_1, \ldots, \mathbf{o}_T$. Usar a regra da cadeia produz

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top, \tag{8.7.12}$$

onde $\partial L/\partial \mathbf{o}_t$ é fornecido por (8.7.11).

A seguir, conforme mostrado em Fig. 8.7.2, no tempo final, passo T a função objetivo L depende do estado oculto $\mathbf{h}_T$ apenas via $\mathbf{o}_T$. Portanto, podemos facilmente encontrar o gradiente $\partial L/\partial \mathbf{h}_T \in$

$\mathbb{R}^h$ usando a regra da cadeia:

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}. \tag{8.7.13}$$

Fica mais complicado para qualquer passo de tempo $t < T$, onde a função objetivo L depende de $\mathbf{h}_t$ via $\mathbf{h}_{t+1}$ e $\mathbf{o}_t$. De acordo com a regra da cadeia, o gradiente do estado oculto $\partial L/\partial \mathbf{h}_t \in \mathbb{R}^h$ a qualquer momento, o passo $t < T$ pode ser calculado recorrentemente como:

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}. \tag{8.7.14}$$

Para análise, expandindo a computação recorrente para qualquer etapa de tempo $1 \leq t \leq T$ dá

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^{T} \left(\mathbf{W}_{hh}^\top\right)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}. \tag{8.7.15}$$

Podemos ver em (8.7.15) que este exemplo linear simples já exibe alguns problemas-chave de modelos de sequência longa: envolve potências potencialmente muito grandes de $\mathbf{W}_{hh}^\top$. Nele, autovalores menores que 1 desaparecem e os autovalores maiores que 1 divergem. Isso é numericamente instável, que se manifesta na forma de desaparecimento e gradientes explosivos. Uma maneira de resolver isso é truncar as etapas de tempo em um tamanho computacionalmente conveniente conforme discutido em Section 8.7.1. Na prática, esse truncamento é efetuado destacando-se o gradiente após um determinado número de etapas de tempo. Mais tarde veremos como modelos de sequência mais sofisticados, como a memória de curto prazo longa, podem aliviar ainda mais isso.

Finalmente, Fig. 8.7.2 mostra que a função objetivo L depende dos parâmetros do modelo $\mathbf{W}_{hx}$ e $\mathbf{W}_{hh}$ na camada oculta via estados ocultos $\mathbf{h}_1, \ldots, \mathbf{h}_T$. Para calcular gradientes com respeito a tais parâmetros $\partial L/\partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ e $\partial L/\partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, aplicamos a regra da cadeia que dá

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top,\\
\frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top,
\end{aligned} \tag{8.7.16}$$

Onde $\partial L/\partial \mathbf{h}_t$ que é calculado recorrentemente por (8.7.13) e (8.7.14) é a quantidade chave que afeta a estabilidade numérica.

Como a retropropagação através do tempo é a aplicação de retropropagação em RNNs, como explicamos em Section 4.7, o treinamento de RNNs alterna a propagação direta com retropropagação através do tempo. Além do mais, retropropagação através do tempo calcula e armazena os gradientes acima por sua vez. Especificamente, valores intermediários armazenados são reutilizados para evitar cálculos duplicados, como armazenar $\partial L/\partial \mathbf{h}_t$ para ser usado no cálculo de $\partial L/\partial \mathbf{W}_{hx}$ e $\partial L/\partial \mathbf{W}_{hh}$.

8.7.3 Resumo

- A retropropagação através do tempo é meramente uma aplicação da retropropagação para sequenciar modelos com um estado oculto.
- O truncamento é necessário para conveniência computacional e estabilidade numérica, como truncamento regular e truncamento aleatório.
- Altos poderes de matrizes podem levar a autovalores divergentes ou desaparecendo. Isso se manifesta na forma de gradientes explodindo ou desaparecendo.
- Para computação eficiente, os valores intermediários são armazenados em cache durante a retropropagação ao longo do tempo.

8.7.4 Exercícios

1. Suponha que temos uma matriz simétrica $\mathbf{M} \in \mathbb{R}^{n\times n}$ with eigenvalues λ_i cujos autovetores correspondentes são $\mathbf{v}_i$ ($i = 1, \ldots, n$). Sem perda de generalidade, assuma que eles estão ordenados na ordem $|\lambda_i| \geq |\lambda_{i+1}|$.
    1. Mostre que $\mathbf{M}^k$ tem autovalores λ_i^k.
    2. Prove que para um vetor aleatório $\mathbf{x} \in \mathbb{R}^n$, com alta probabilidade $\mathbf{M}^k\mathbf{x}$ estará muito alinhado com o autovetor $\mathbf{v}_1$ de $\mathbf{M}$. Formalize esta declaração.
    3. O que o resultado acima significa para gradientes em RNNs?
2. Além do recorte de gradiente, você consegue pensar em outros métodos para lidar com a explosão de gradiente em redes neurais recorrentes?

Discussions[91]

[91] https://discuss.d2l.ai/t/334

9 | Redes Neurais Recorrentes Modernas

Introduzimos os conceitos básicos de RNNs, que podem lidar melhor com dados de sequência. Para demonstração, implementamos modelos de linguagem baseados em RNN em dados de texto. No entanto, tais técnicas podem não ser suficiente para os profissionais quando eles enfrentam uma ampla gama de problemas de aprendizagem de sequência hoje em dia.

Por exemplo, um problema notável na prática é a instabilidade numérica dos RNNs. Embora tenhamos truques de implementação aplicados, como recorte de gradiente, esse problema pode ser aliviado ainda mais com designs mais sofisticados de modelos de sequência. Especificamente, os RNNs controlados são muito mais comuns na prática. Começaremos apresentando duas dessas redes amplamente utilizadas, chamadas de *gated recurrent units* (GRUs) e *long short-term memory* (LSTM). Além disso, vamos expandir o RNN arquitetura com uma única camada oculta indireta que foi discutida até agora. Descreveremos arquiteturas profundas com múltiplas camadas ocultas e discutiremos o projeto bidirecional com cálculos recorrentes para frente e para trás. Essas expansões são frequentemente adotadas em redes recorrentes modernas. Ao explicar essas variantes RNN, continuamos a considerar o mesmo problema de modelagem de linguagem apresentado no Chapter 8.

Na verdade, a modelagem de linguagem revela apenas uma pequena fração do que o aprendizado de sequência é capaz. Em uma variedade de problemas de aprendizagem de sequência, como reconhecimento automático de fala, conversão de texto em fala, e tradução automática, tanto as entradas quanto as saídas são sequências de comprimento arbitrário. Explicar como ajustar este tipo de dados, tomaremos a tradução automática como exemplo e apresentaremos o arquitetura codificador-decodificador baseada em RNNs e busca de feixe para geração de sequência.

9.1 Gated Recurrent Units (GRU)

Em Section 8.7, discutimos como os gradientes são calculados em RNNs. Em particular, descobrimos que produtos longos de matrizes podem levar para gradientes desaparecendo ou explodindo. Vamos pensar brevemente sobre como anomalias de gradiente significam na prática:

- Podemos encontrar uma situação em que uma observação precoce é altamente significativo para prever todas as observações futuras. Considere o caso inventado em que a primeira observação contém uma soma de verificação e o objetivo é para discernir se a soma de verificação está correta no final da sequência. Nesse caso, a influência do primeiro token é vital. Gostaríamos de ter algum mecanismos para armazenar informações precoces vitais em uma *célula de memória*. Sem tal um mecanismo, teremos que atribuir um gradiente muito grande a esta observação, uma vez que afeta todas as observações subsequentes.

- Podemos encontrar situações em que alguns tokens não carreguem observação. Por exemplo, ao analisar uma página da web, pode haver Código HTML que é irrelevante para o propósito de avaliar o sentimento transmitido na página. Gostaríamos de ter algum mecanismo para *pular* tais tokens na representação do estado latente.
- Podemos encontrar situações em que haja uma quebra lógica entre as partes de uma sequência. Por exemplo, pode haver uma transição entre capítulos em um livro, ou uma transição entre um mercado de valores em baixa e em alta. Neste caso seria bom ter um meio de *redefinir* nosso estado interno representação.

Vários métodos foram propostos para resolver isso. Uma das mais antigas é a memória de curto prazo longa (Hochreiter & Schmidhuber, 1997) que nós iremos discutir em Section 9.2. A unidade recorrente fechada (GRU) (Cho et al., 2014a) é uma variante um pouco mais simplificada que muitas vezes oferece desempenho comparável e é significativamente mais rápido para computar (Chung et al., 2014). Por sua simplicidade, comecemos com o GRU.

9.1.1 Estado Oculto Fechado

A principal distinção entre RNNs vanilla e GRUs é que o último suporta o bloqueio do estado oculto. Isso significa que temos mecanismos dedicados para quando um estado oculto deve ser *atualizado* e também quando deve ser *redefinido*. Esses mecanismos são aprendidos e atendem às questões listadas acima. Por exemplo, se o primeiro token é de grande importância aprenderemos a não atualizar o estado oculto após a primeira observação. Da mesma forma, aprenderemos a pular observações temporárias irrelevantes. Por último, aprenderemos a redefinir o estado latente sempre que necessário. Discutimos isso em detalhes abaixo.

Porta de Reinicialização e a Porta de Atualização

A primeira coisa que precisamos apresentar é a *porta de reinicialização* e a *porta de atualização*. Nós os projetamos para serem vetores com entradas em $(0, 1)$ para que possamos realizar combinações convexas. Por exemplo, uma porta de reinicialização nos permitiria controlar quanto do estado anterior ainda podemos querer lembrar. Da mesma forma, uma porta de atualização nos permitiria controlar quanto do novo estado é apenas uma cópia do antigo estado.

Começamos projetando esses portões. Fig. 9.1.1 ilustra as entradas para ambos as portas de reset e atualização em uma GRU, dada a entrada da etapa de tempo atual e o estado oculto da etapa de tempo anterior. As saídas de duas portas são fornecidos por duas camadas totalmente conectadas com uma função de ativação sigmóide.

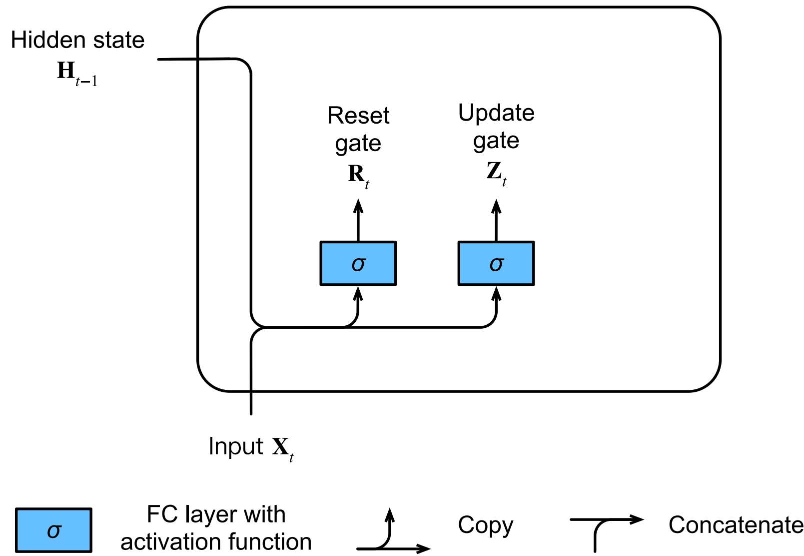


Fig. 9.1.1: Calculando a porta de reinicialização e a porta de atualização em um modelo GRU.

Matematicamente, para um determinado intervalo de tempo $ t $, suponha que a entrada seja um minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (número de exemplos: $ n $, número de entradas: $ d $) e o estado oculto da etapa de tempo anterior é $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ (número de unidades ocultas: $ h $). Então, reinicie o portão $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ e atualize o portão $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ são calculados da seguinte forma:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),\\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),\end{aligned} \tag{9.1.1}$$

onde $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ e $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ são pesos de parâmetros e $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ são viéses. Observe que a transmissão (consulte Section 2.1.3) é acionada durante a soma. Usamos funções sigmóides (como introduzidas em Section 4.1) para transformar os valores de entrada no intervalo $(0, 1)$.

Estado Oculto do Candidato

Em seguida, vamos integrar a porta de reset $\mathbf{R}_t$ com o mecanismo regular de atualização de estado latente in (8.4.5). Isso leva ao seguinte *estado oculto candidato* $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ no passo de tempo t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \left(\mathbf{R}_t \odot \mathbf{H}_{t-1}\right) \mathbf{W}_{hh} + \mathbf{b}_h), \tag{9.1.2}$$

onde $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ são parâmetros de pesos, $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ é o viés, e o símbolo $\odot$ é o operador de produto Hadamard (elemento a elemento). Aqui, usamos uma não linearidade na forma de tanh para garantir que os valores no estado oculto candidato permaneçam no intervalo $(-1, 1)$.

O resultado é um *candidato*, pois ainda precisamos incorporar a ação da porta de atualização. Comparando com (8.4.5), agora a influência dos estados anteriores pode ser reduzido com o multiplicação elementar de $\mathbf{R}_t$ e$\mathbf{H}_{t-1}$ em (9.1.2). Sempre que as entradas na porta de reset $\mathbf{R}_t$ estão perto de 1, recuperamos um RNN vanilla como em (8.4.5). Para todas as entradas da porta de reset $\mathbf{R}_t$ que estão próximas de 0, o estado oculto candidato é o resultado de um MLP com $\mathbf{X}_t$ como a entrada. Qualquer estado oculto pré-existente é, portanto, *redefinido* para os padrões.

Fig. 9.1.2 ilustra o fluxo computacional após aplicar a porta de reset.

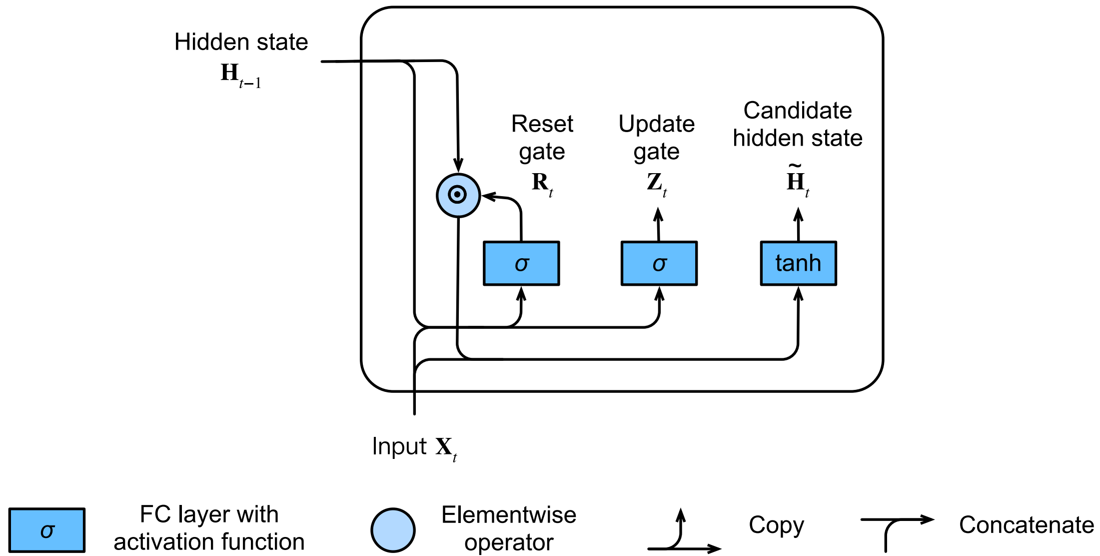


Fig. 9.1.2: Calculando o estado oculto candidato em um modelo GRU.

Estados Escondidos

Finalmente, precisamos incorporar o efeito da porta de atualização $\mathbf{Z}_t$. Isso determina até que ponto o novo estado oculto $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ é apenas o antigo estado $\mathbf{H}_{t-1}$ e em quanto o novo estado candidato $\tilde{\mathbf{H}}_t$ é usado. A porta de atualização $\mathbf{Z}_t$ pode ser usada para este propósito, simplesmente tomando combinações convexas elemento a elemento entre $\mathbf{H}_{t-1}$ e $\tilde{\mathbf{H}}_t$. Isso leva à equação de atualização final para a GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{9.1.3}$$

Sempre que a porta de atualização $\mathbf{Z}_t$ está próxima de 1, simplesmente mantemos o estado antigo. Neste caso, a informação de $\mathbf{X}_t$ é essencialmente ignorada, pulando efetivamente o passo de tempo t na cadeia de dependências. Em contraste, sempre que $\mathbf{Z}_t$ está próximo de 0, o novo estado latente $\mathbf{H}_t$ se aproxima do estado latente candidato $\tilde{\mathbf{H}}_t$. Esses projetos podem nos ajudar a lidar com o problema do gradiente de desaparecimento em RNNs e melhor capturar dependências para sequências com grandes distâncias de intervalo de tempo. Por exemplo, se a porta de atualização estiver perto de 1 para todas as etapas de tempo de uma subsequência inteira, o antigo estado oculto na etapa do tempo de seu início será facilmente retido e aprovado até o fim, independentemente do comprimento da subsequência.

Fig. 9.1.3 ilustra o fluxo computacional depois que a porta de atualização está em ação.

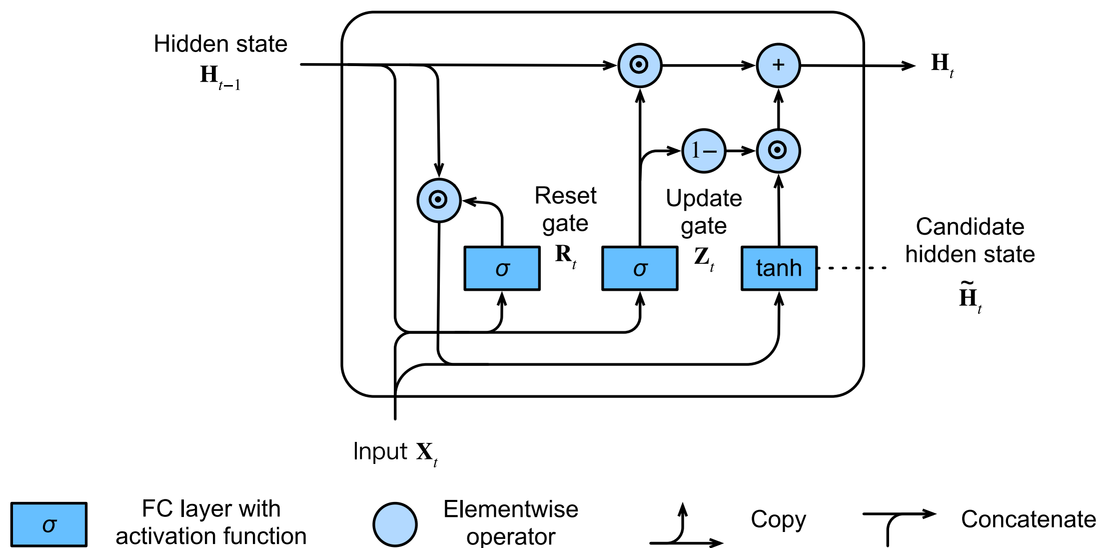


Fig. 9.1.3: Computing the hidden state in a GRU model. Calculando o estado oculto em um modelo GRU.

Em resumo, GRUs têm as duas características distintas a seguir:

- As portas de redefinição ajudam a capturar dependências de curto prazo em sequências.
- Portas de atualização ajudam a capturar dependências de longo prazo em sequências.

9.1.2 Implementação do zero

Para entender melhor o modelo GRU, vamos implementá-lo do zero. Começamos lendo o conjunto de dados da máquina do tempo que usamos em Section 8.5. O código para ler o conjunto de dados é fornecido abaixo.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Inicializando os parâmetros do modelo

A próxima etapa é inicializar os parâmetros do modelo. Tiramos os pesos de uma distribuição gaussiana com desvio padrão de 0,01 e defina o bias como 0. O hiperparâmetro `num_hiddens` define o número de unidades ocultas. Instanciamos todos os pesos e vieses relacionados à porta de atualização, a porta de redefinição, o estado oculto candidato, e a camada de saída.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01
```

(continues on next page)

(continued from previous page)

```
    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xz, W_hz, b_z = three()  # Update gate parameters
    W_xr, W_hr, b_r = three()  # Reset gate parameters
    W_xh, W_hh, b_h = three()  # Candidate hidden state parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

Definindo o modelo

Agora vamos definir a função de inicialização de estado oculto `init_gru_state`. Assim como a função `init_rnn_state` definida em Section 8.5, esta função retorna um tensor com uma forma (tamanho do lote, número de unidades ocultas) cujos valores são todos zeros.

```
def init_gru_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

Agora estamos prontos para definir o modelo GRU. Sua estrutura é a mesma da célula RNN básica, exceto que as equações de atualização são mais complexas.

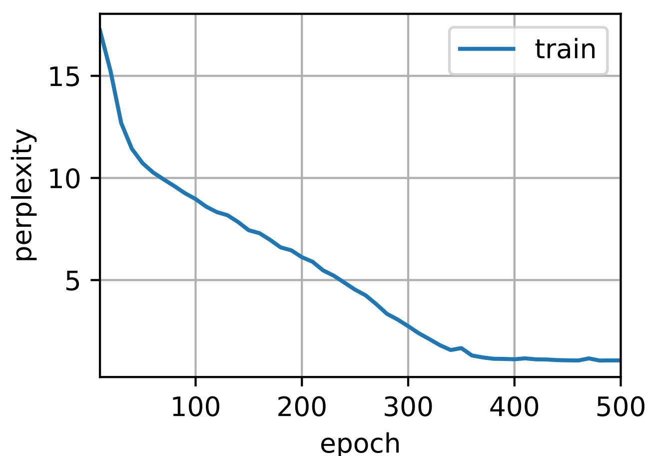
```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid((X @ W_xz) + (H @ W_hz) + b_z)
        R = torch.sigmoid((X @ W_xr) + (H @ W_hr) + b_r)
        H_tilda = torch.tanh((X @ W_xh) + ((R * H) @ W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = H @ W_hq + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

Treinamento e previsão

O treinamento e a previsão funcionam exatamente da mesma maneira que em Section 8.5. Após o treinamento, imprimimos a perplexidade no conjunto de treinamento e a sequência prevista seguindo os prefixos fornecidos "viajante do tempo" e "viajante", respectivamente.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_params,
                            init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 23915.5 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```

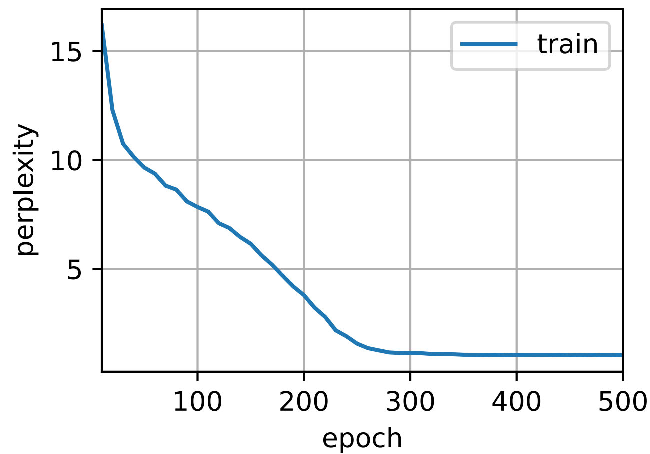


9.1.3 Implementação concisa

Em APIs de alto nível, nós podemos diretamente instanciar um modelo de GPU. Isso encapsula todos os detalhes de configuração que tornamos explícitos acima. O código é significativamente mais rápido, pois usa operadores compilados em vez de Python para muitos detalhes que explicamos antes.

```
num_inputs = vocab_size
gru_layer = nn.GRU(num_inputs, num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 302984.3 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



9.1.4 Sumário

- RNNs bloqueados podem capturar melhor as dependências para sequências com grandes distâncias de intervalo de tempo.
- As portas de redefinição ajudam a capturar dependências de curto prazo em sequências.
- Portas de atualização ajudam a capturar dependências de longo prazo em sequências.
- GRUs contêm RNNs básicos como seu caso extremo sempre que a porta de reinicialização é ativada. Eles também podem pular subseqüências ativando a porta de atualização.

9.1.5 Exercícios

1. Suponha que queremos apenas usar a entrada na etapa de tempo t' para prever a saída na etapa de tempo $t > t'$. Quais são os melhores valores para as portas de reset e atualização para cada intervalo de tempo?
2. Ajuste os hiperparâmetros e analise sua influência no tempo de execução, perplexidade e sequência de saída.
3. Compare o tempo de execução, a perplexidade e as strings de saída para as implementações `rnn.RNN` e `rnn.GRU` entre si.
4. O que acontece se você implementar apenas partes de uma GRU, por exemplo, com apenas uma porta de reinicialização ou apenas uma porta de atualização?

Discussão[92]

[92] https://discuss.d2l.ai/t/1056

9.2 Memória Longa de Curto Prazo (LSTM)

O desafio de abordar a preservação de informações de longo prazo e entrada de curto prazo pulando modelos de variáveis latentes existem há muito tempo. Uma das primeiras abordagens para resolver isso foi a longa memória de curto prazo (LSTM) (Hochreiter & Schmidhuber, 1997). Ele compartilha muitos dos as propriedades da GRU. Curiosamente, os LSTMs têm um design um pouco mais complexo do que os GRUs mas antecede GRUs em quase duas décadas.

9.2.1 Célula de Memória Bloqueada

Indiscutivelmente, o design da LSTM é inspirado pelas portas lógicas de um computador. LSTM introduz uma *célula de memória* (ou *célula* para abreviar) que tem a mesma forma que o estado oculto (algumas literaturas consideram a célula de memória como um tipo especial de estado oculto), projetado para registrar informações adicionais. Para controlar a célula de memória precisamos de vários portões. Um portão é necessário para ler as entradas do célula. Vamos nos referir a isso como o *portão de saída*. Uma segunda porta é necessária para decidir quando ler os dados para o célula. Chamamos isso de *porta de entrada*. Por último, precisamos de um mecanismo para redefinir o conteúdo da célula, governado por um *portão de esquecimento*. A motivação para tal design é o mesmo das GRUs, ou seja, ser capaz de decidir quando lembrar e quando ignorar entradas no estado oculto por meio de um mecanismo dedicado. Deixe-nos ver como isso funciona na prática.

Porta de entrada, porta de esquecimento e porta de saída

Assim como em GRUs, os dados que alimentam as portas LSTM são a entrada na etapa de tempo atual e o estado oculto da etapa de tempo anterior, conforme ilustrado em Fig. 9.2.1. Eles são processados por três camadas totalmente conectadas com uma função de ativação sigmóide para calcular os valores de a entrada, esqueça. e portas de saída. Como resultado, os valores das três portas estão na faixa de $(0, 1)$.

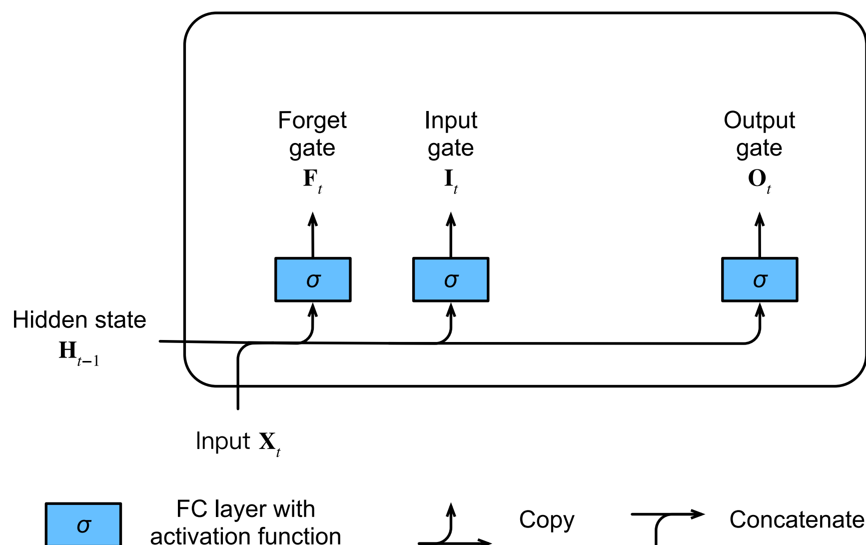


Fig. 9.2.1: Calculando a porta de entrada, a porta de esquecimento e a porta de saída em um modelo LSTM.

Matematicamente, suponha que existam h unidades ocultas, o tamanho do lote é n e o número de entradas é d. Assim, a entrada é $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ e o estado oculto da etapa de tempo anterior é $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$. Correspondentemente, as portas na etapa de tempo t são definidos da seguinte forma: a porta de entrada é $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, a porta de esquecimento é $\mathbf{F}_t \in \mathbb{R}^{n\times h}$, e a porta de saída é $\mathbf{O}_t \in \mathbb{R}^{n\times h}$. Eles são calculados da seguinte forma:

$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned} \tag{9.2.1}$$

onde $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ e $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ são parâmetros de pesos e $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ são parâmetros viéses.

Célula de Memória Candidata

Em seguida, projetamos a célula de memória. Como ainda não especificamos a ação das várias portas, primeiro introduzimos a célula de memória *candidata* $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$. Seu cálculo é semelhante ao das três portas descritas acima, mas usando uma função tanh com um intervalo de valores para $(-1, 1)$ como a função de ativação. Isso leva à seguinte equação na etapa de tempo t:

$$\tilde{\mathbf{C}}_t = \text{tanh}(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c), \tag{9.2.2}$$

onde $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ são parâmetros de pesos $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ é um parâmetro de viés.

Uma ilustração rápida da célula de memória candidata é mostrada em Fig. 9.2.2.

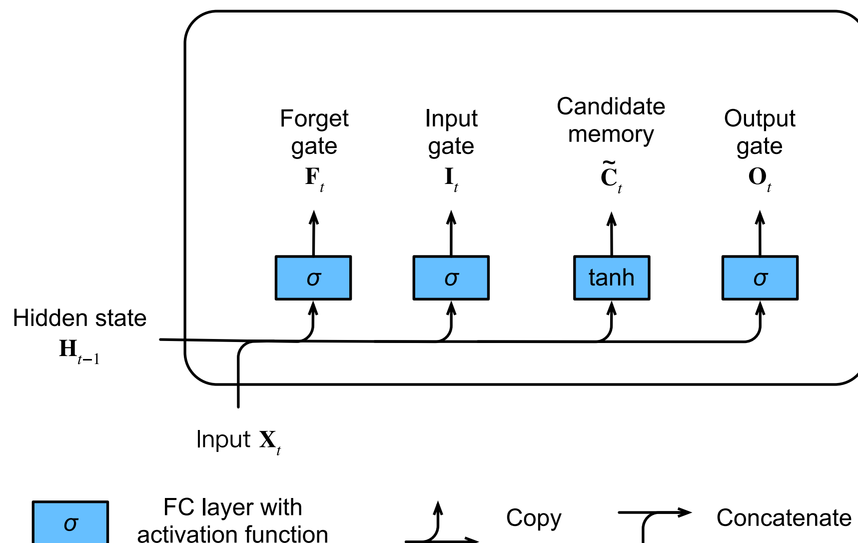


Fig. 9.2.2: Computando a célula de memória candidata em um modelo LSTM.

Célula de Memória

Em GRUs, temos um mecanismo para controlar a entrada e o esquecimento (ou salto). De forma similar, em LSTMs, temos duas portas dedicadas para tais propósitos: a porta de entrada $\mathbf{I}_t$ governa o quanto levamos os novos dados em conta via $\tilde{\mathbf{C}}_t$ e a porta de esquecer $\mathbf{F}_t$ aborda quanto do conteúdo da célula de memória antiga $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ retemos. Usando o mesmo truque de multiplicação pontual de antes, chegamos à seguinte equação de atualização:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \tag{9.2.3}$$

Se a porta de esquecimento é sempre aproximadamente 1 e a porta de entrada é sempre aproximadamente 0, as células de memória anteriores $\mathbf{C}_{t-1}$ serão salvas ao longo do tempo e passadas para o intervalo de tempo atual. Este projeto é introduzido para aliviar o problema do gradiente de desaparecimento e para melhor capturar dependências de longo alcance dentro de sequências.

Assim, chegamos ao diagrama de fluxo em Fig. 9.2.3.

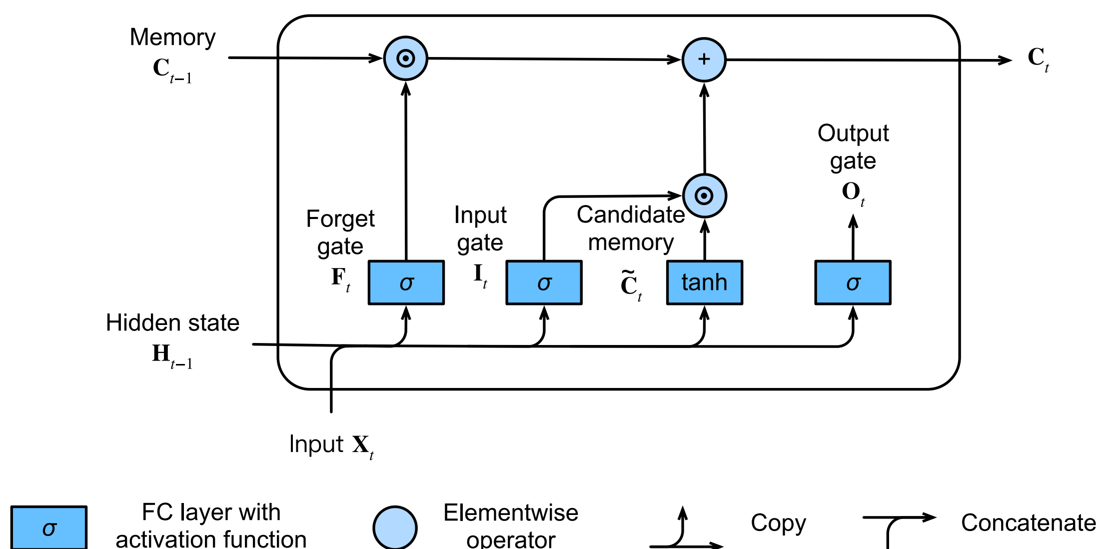


Fig. 9.2.3: Calculando a célula de memória em um modelo LSTM.

Estado Oculto

Por último, precisamos definir como calcular o estado oculto $\mathbf{H}_t \in \mathbb{R}^{n \times h}$. É aqui que a porta de saída entra em ação. No LSTM, é simplesmente uma versão bloqueada do tanh da célula de memória. Isso garante que os valores de $\mathbf{H}_t$ estejam sempre no intervalo $(-1, 1)$.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \tag{9.2.4}$$

Sempre que a porta de saída se aproxima de 1, passamos efetivamente todas as informações da memória para o preditor, enquanto para a porta de saída próxima de 0, retemos todas as informações apenas dentro da célula de memória e não realizamos nenhum processamento posterior.

Fig. 9.2.4 tem uma ilustração gráfica do fluxo de dados.

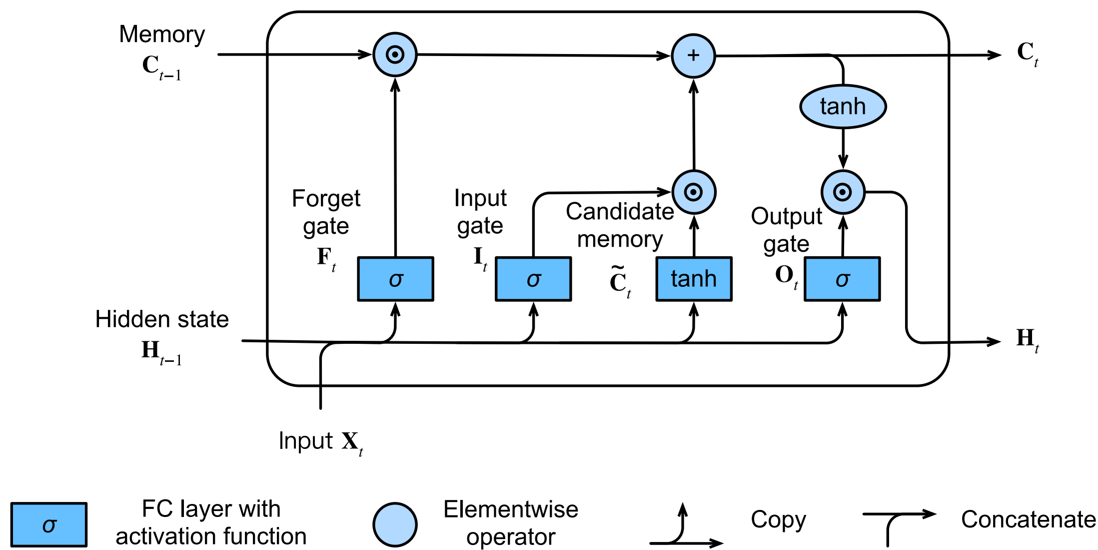


Fig. 9.2.4: Calculando o estado oculto em um modelo LSTM.

9.2.2 Implementação do zero

Agora, vamos implementar um LSTM do zero. Da mesma forma que os experimentos em Section 8.5, primeiro carregamos o conjunto de dados da máquina do tempo.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Inicializando os parâmetros do modelo

Em seguida, precisamos definir e inicializar os parâmetros do modelo. Como anteriormente, o hiperparâmetro `num_hiddens` define o número de unidades ocultas. Inicializamos os pesos seguindo uma distribuição gaussiana com desvio padrão de 0,01 e definimos os vieses como 0.

```
def get_lstm_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xi, W_hi, b_i = three()  # Input gate parameters
    W_xf, W_hf, b_f = three()  # Forget gate parameters
    W_xo, W_ho, b_o = three()  # Output gate parameters
```

(continues on next page)

(continued from previous page)

```
    W_xc, W_hc, b_c = three()  # Candidate memory cell parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

Definindo o modelo

Na função de inicialização, o estado oculto do LSTM precisa retornar uma célula de memória *adicional* com um valor de 0 e uma forma de (tamanho do lote, número de unidades ocultas). Consequentemente, obtemos a seguinte inicialização de estado.

```
def init_lstm_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),
            torch.zeros((batch_size, num_hiddens), device=device))
```

O modelo real é definido exatamente como o que discutimos antes: fornecer três portas e uma célula de memória auxiliar. Observe que apenas o estado oculto é passado para a camada de saída. A célula de memória $\mathbf{C}_t$ não participa diretamente no cálculo de saída.

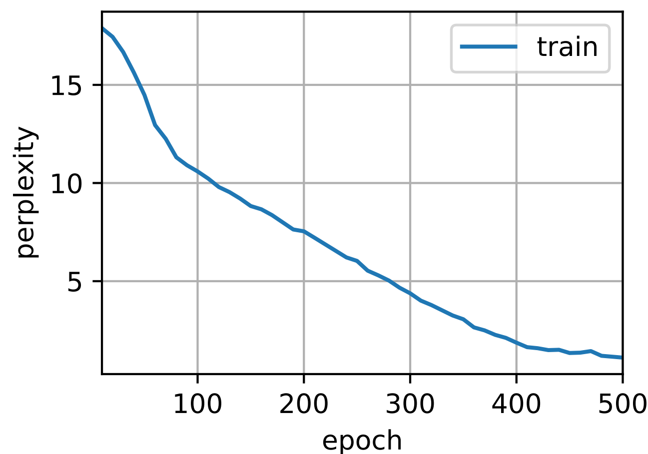
```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid((X @ W_xi) + (H @ W_hi) + b_i)
        F = torch.sigmoid((X @ W_xf) + (H @ W_hf) + b_f)
        O = torch.sigmoid((X @ W_xo) + (H @ W_ho) + b_o)
        C_tilda = torch.tanh((X @ W_xc) + (H @ W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * torch.tanh(C)
        Y = (H @ W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H, C)
```

Treinamento e previsão

Vamos treinar um LSTM da mesma forma que fizemos em Section 9.1, instanciando a classe RNNModelScratch como introduzida em Section 8.5.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_lstm_params,
                            init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 20015.8 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
travelleryou can show black is white by argument said filby
```

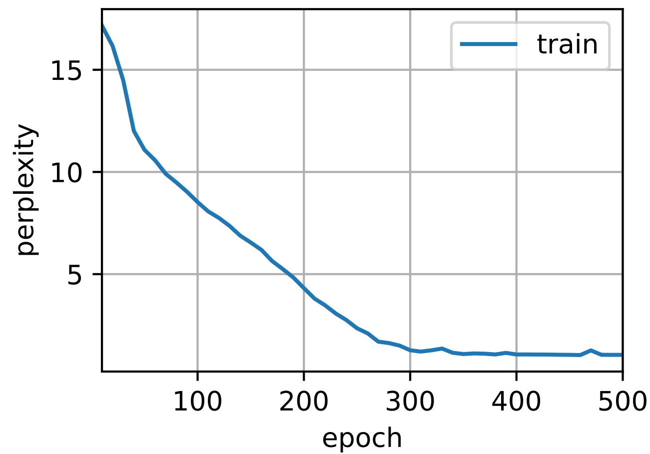


9.2.3 Implementação concisa

Usando APIs de alto nível, podemos instanciar diretamente um modelo LSTM. Isso encapsula todos os detalhes de configuração que tornamos explícitos acima. O código é significativamente mais rápido, pois usa operadores compilados em vez de Python para muitos detalhes que explicamos em detalhes antes.

```
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 282016.6 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller with a slight accession ofcheerfulness really thi
```



LSTMs são o modelo autorregressivo de variável latente prototípica com controle de estado não trivial. Muitas variantes foram propostas ao longo dos anos, por exemplo, camadas múltiplas, conexões residuais, diferentes tipos de regularização. No entanto, treinar LSTMs e outros modelos de sequência (como GRUs) são bastante caros devido à dependência de longo alcance da sequência. Mais tarde, encontraremos modelos alternativos, como transformadores, que podem ser usados em alguns casos.

9.2.4 Resumo

- Os LSTMs têm três tipos de portas: portas de entrada, portas de esquecimento e portas de saída que controlam o fluxo de informações.
- A saída da camada oculta do LSTM inclui o estado oculto e a célula de memória. Apenas o estado oculto é passado para a camada de saída. A célula de memória é totalmente interna.
- LSTMs podem aliviar gradientes que desaparecem e explodem.

9.2.5 Exercícios

1. Ajuste os hiperparâmetros e analise sua influência no tempo de execução, perplexidade e sequência de saída.
2. Como você precisaria mudar o modelo para gerar palavras adequadas em vez de sequências de caracteres?
3. Compare o custo computacional para GRUs, LSTMs e RNNs regulares para uma determinada dimensão oculta. Preste atenção especial ao custo de treinamento e inferência.
4. Uma vez que a célula de memória candidata garante que o intervalo de valores está entre -1 e 1 usando a função tanh, por que o estado oculto precisa usar a função tanh novamente para garantir que a saída o intervalo de valores está entre -1 e 1?
5. Implemente um modelo LSTM para predição de série temporal em vez de predição de sequência de caracteres.

Discussão[93]

[93] https://discuss.d2l.ai/t/1057

9.3 Redes neurais recorrentes profundas

Até agora, discutimos apenas RNNs com uma única camada oculta unidirecional. Nele, a forma funcional específica de como as variáveis latentes e as observações interagem é bastante arbitrária. Este não é um grande problema, desde que tenhamos flexibilidade suficiente para modelar diferentes tipos de interações. Com uma única camada, no entanto, isso pode ser bastante desafiador. No caso dos modelos lineares, corrigimos esse problema adicionando mais camadas. Em RNNs, isso é um pouco mais complicado, pois primeiro precisamos decidir como e onde adicionar não linearidade extra.

Na verdade, poderíamos empilhar várias camadas de RNNs umas sobre as outras. Isso resulta em um mecanismo flexível, devido à combinação de várias camadas simples. Em particular, os dados podem ser relevantes em diferentes níveis da pilha. Por exemplo, podemos querer manter disponíveis dados de alto nível sobre as condições do mercado financeiro (bear ou bull market), ao passo que em um nível mais baixo registramos apenas dinâmicas temporais de curto prazo.

Além de toda a discussão abstrata acima provavelmente é mais fácil entender a família de modelos em que estamos interessados revisando Fig. 9.3.1. Ele descreve um RNN profundo com L camadas ocultas. Cada estado oculto é continuamente passado para a próxima etapa da camada atual e para a etapa atual da próxima camada.

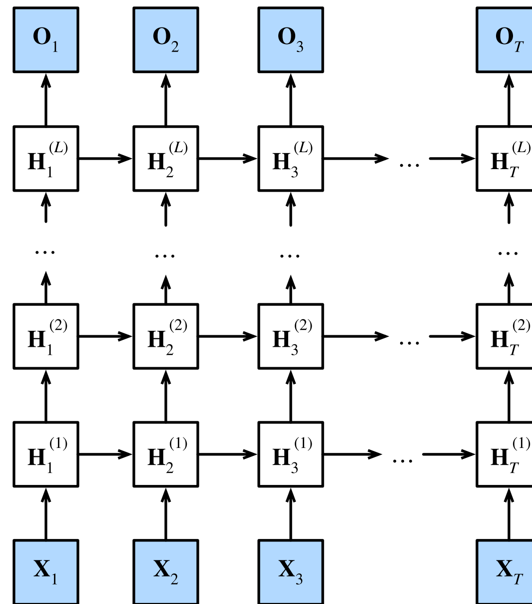


Fig. 9.3.1: Arquitetura de RNN profunda.

9.3.1 Dependência Funcional

Podemos formalizar o dependências funcionais dentro da arquitetura profunda de L camadas ocultas representado em Fig. 9.3.1. Nossa discussão a seguir se concentra principalmente em o modelo vanilla RNN, mas também se aplica a outros modelos de sequência.

Suponha que temos uma entrada de minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (número de exemplos: n, número de entradas em cada exemplo: d) no passo de tempo t. Ao mesmo tempo, deixar o estado oculto da camada oculta l^th ($ l = 1, ldots, L $) é $l = 1, \ldots, L$) é $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (número de unidades ocultas: h) e a variável da camada de saída é $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (número de saídas: q). Configurando $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, o

estado oculto de a camada oculta l^th que usa a função de ativação ϕ_l é expresso da seguinte forma:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}), \tag{9.3.1}$$

onde os pesos $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$ e $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, junto com o viés $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$, são os parâmetros do modelo de a camada oculta l^th.

No fim, o cálculo da camada de saída é baseado apenas no estado oculto da camada oculta final L^th:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q, \tag{9.3.2}$$

onde o peso $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ e o viés $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ são os parâmetros do modelo da camada de saída.

Assim como acontece com os MLPs, o número de camadas ocultas L e o número de unidades ocultas h são hiperparâmetros. Em outras palavras, eles podem ser ajustados ou especificados por nós. Além disso, podemos facilmente obter um RNN com portas profundas substituindo o cálculo do estado oculto em (9.3.1) com aquele de um GRU ou um LSTM.

9.3.2 Implementação Concisa

Felizmente, muitos dos detalhes logísticos necessários para implementar várias camadas de um RNN estão prontamente disponíveis em APIs de alto nível. Para manter as coisas simples, apenas ilustramos a implementação usando essas funcionalidades integradas. Tomemos um modelo LSTM como exemplo. O código é muito semelhante ao que usamos anteriormente em Section 9.2. Na verdade, a única diferença é que especificamos o número de camadas explicitamente, em vez de escolher o padrão de uma única camada. Como de costume, começamos carregando o conjunto de dados.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

As decisões arquitetônicas, como a escolha de hiperparâmetros, são muito semelhantes às de Section 9.2. Escolhemos o mesmo número de entradas e saídas, pois temos tokens distintos, ou seja, `vocab_size`. O número de unidades ocultas ainda é 256. A única diferença é que agora selecionamos um número não trivial de camadas ocultas, especificando o valor de `núm_camadas`.

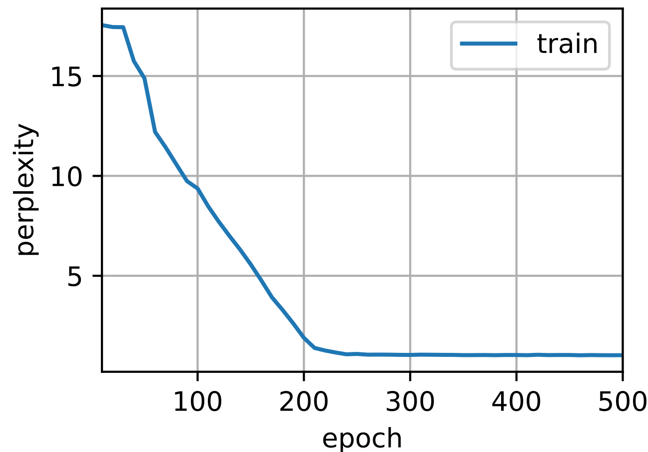
```
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
device = d2l.try_gpu()
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
```

9.3.3 Treinamento e Predição

Como agora instanciamos duas camadas com o modelo LSTM, essa arquitetura um tanto mais complexa retarda o treinamento consideravelmente.

```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 201228.1 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



9.3.4 Sumário

- Em RNNs profundos, as informações de estado oculto são passadas para a próxima etapa da camada atual e a etapa atual da próxima camada.
- Existem muitos sabores diferentes de RNNs profundos, como LSTMs, GRUs ou RNNs vanilla. Convenientemente, esses modelos estão todos disponíveis como partes das APIs de alto nível de estruturas de aprendizado profundo.
- A inicialização de modelos requer cuidados. No geral, os RNNs profundos requerem uma quantidade considerável de trabalho (como taxa de aprendizado e recorte) para garantir a convergência adequada.

9.3.5 Exercícios

1. Tente implementar um RNN de duas camadas do zero usando a implementação de camada única que discutimos em Section 8.5.
2. Substitua o LSTM por um GRU e compare a precisão e a velocidade de treinamento.
3. Aumente os dados de treinamento para incluir vários livros. Quão baixo você pode ir na escala de perplexidade?
4. Você gostaria de combinar fontes de diferentes autores ao modelar um texto? Por que isso é uma boa ideia? O que poderia dar errado?

Discussão[94]

9.4 Redes Neurais Recorrentes Bidirecionais

No aprendizado sequencial, até agora, assumimos que nosso objetivo é modelar a próxima saída, dado o que vimos até agora, por exemplo, no contexto de uma série temporal ou no contexto de um modelo de linguagem. Embora este seja um cenário típico, não é o único que podemos encontrar. Para ilustrar o problema, considere as três tarefas a seguir para preencher o espaço em branco em uma sequência de texto:

- Eu estou___.
- Eu estou ___ faminto.
- Eu estou ___ faminto, e poderia comer meio porco.

Dependendo da quantidade de informações disponíveis, podemos preencher os espaços em branco com palavras muito diferentes, como "feliz", "não" e "muito". Claramente, o final da frase (se disponível) transmite informações significativas sobre qual palavra escolher. Um modelo de sequência que é incapaz de tirar vantagem disso terá um desempenho ruim em tarefas relacionadas. Por exemplo, para se sair bem no reconhecimento de entidade nomeada (por exemplo, para reconhecer se "Verde" se refere a "Sr. Verde" ou à cor) contexto de longo alcance é igualmente vital. Para obter alguma inspiração para abordar o problema, façamos um desvio para modelos gráficos probabilísticos.

9.4.1 Programação dinâmica em modelos de Markov ocultos

Esta subseção serve para ilustrar o problema de programação dinâmica. Os detalhes técnicos específicos não importam para a compreensão dos modelos de aprendizagem profunda mas ajudam a motivar por que se pode usar o aprendizado profundo e por que se pode escolher arquiteturas específicas.

Se quisermos resolver o problema usando modelos gráficos probabilísticos, poderíamos, por exemplo, projetar um modelo de variável latente como segue. A qualquer momento, passo t, assumimos que existe alguma variável latente h_t que governa nossa emissão observada x_t via $P(x_t \mid h_t)$. Além disso, qualquer transição $h_t \to h_{t+1}$ é dada por alguma probabilidade de transição de estado $P(h_{t+1} \mid h_t)$. Este modelo gráfico probabilístico é então um *modelo Markov oculto* como em :numref: `=fig_hmm`.

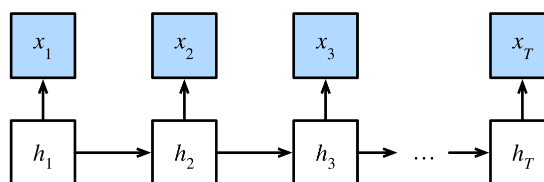


Fig. 9.4.1: Um modelo de Markov oculto.

Desse modo, para uma sequência de observações T, temos a seguinte distribuição de probabili-

[94] https://discuss.d2l.ai/t/1058

dade conjunta sobre os estados observados e ocultos:

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t), \text{ onde} P(h_1 \mid h_0) = P(h_1). \tag{9.4.1}$$

Agora suponha que observamos todos os x_i com exceção de alguns x_j e nosso objetivo é calcular $P(x_j \mid x_{-j})$, onde $x_{-j} = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_T)$. Uma vez que não há variável latente em $P(x_j \mid x_{-j})$, nós consideramos resumir todas as combinações possíveis de escolhas para $h_1, \ldots, h_T$. No caso de qualquer h_i poder assumir valores distintos de k (um número finito de estados), isso significa que precisamos somar mais de k^T termos — normalmente missão impossível! Felizmente, existe uma solução elegante para isso: *programação dinâmica*.

Para ver como funciona, considere somar as variáveis latentes $h_1, \ldots, h_T$ por sua vez. De acordo com (9.4.1), isso produz:

$$\begin{aligned}
&P(x_1, \ldots, x_T) \\
=& \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
=& \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
=& \sum_{h_2, \ldots, h_T} \underbrace{\left[\sum_{h_1} P(h_1) P(x_1 \mid h_1) P(h_2 \mid h_1)\right]}_{\pi_2(h_2) \stackrel{\mathrm{def}}{=}} P(x_2 \mid h_2) \prod_{t=3}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
=& \sum_{h_3, \ldots, h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2) P(x_2 \mid h_2) P(h_3 \mid h_2)\right]}_{\pi_3(h_3) \stackrel{\mathrm{def}}{=}} P(x_3 \mid h_3) \prod_{t=4}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
=& \ldots \\
=& \sum_{h_T} \pi_T(h_T) P(x_T \mid h_T).
\end{aligned} \tag{9.4.2}$$

Em geral, temos a *recursão direta* como

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t) P(x_t \mid h_t) P(h_{t+1} \mid h_t). \tag{9.4.3}$$

A recursão é inicializada como $\pi_1(h_1) = P(h_1)$. Em termos abstratos, isso pode ser escrito como $\pi_{t+1} = f(\pi_t, x_t)$, onde f é alguma função aprendível. Isso se parece muito com a equação de atualização nos modelos de variáveis latentes que discutimos até agora no contexto de RNNs!

De forma totalmente análoga à recursão direta, nós também podemos soma sobre o mesmo con-

junto de variáveis latentes com uma recursão para trás. Isso produz:

$$\begin{aligned}
& P(x_1, \ldots, x_T) \\
= & \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
= & \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot P(h_T \mid h_{T-1}) P(x_T \mid h_T) \\
= & \sum_{h_1, \ldots, h_{T-1}} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_T} P(h_T \mid h_{T-1}) P(x_T \mid h_T)\right]}_{\rho_{T-1}(h_{T-1}) \stackrel{\text{def}}{=}} \\
= & \sum_{h_1, \ldots, h_{T-2}} \prod_{t=1}^{T-2} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_{T-1}} P(h_{T-1} \mid h_{T-2}) P(x_{T-1} \mid h_{T-1}) \rho_{T-1}(h_{T-1})\right]}_{\rho_{T-2}(h_{T-2}) \stackrel{\text{def}}{=}} \\
= & \ldots \\
= & \sum_{h_1} P(h_1) P(x_1 \mid h_1) \rho_1(h_1).
\end{aligned} \tag{9.4.4}$$

Podemos, portanto, escrever a *recursão para trás* como

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \rho_t(h_t), \tag{9.4.5}$$

com inicialização $\rho_T(h_T) = 1$. Ambas as recursões para frente e para trás nos permitem somar mais de T variáveis latentes em $\mathcal{O}(kT)$ (linear) tempo sobre todos os valores de $(h_1, \ldots, h_T)$ em vez de em tempo exponencial . Este é um dos grandes benefícios da inferência probabilística com modelos gráficos. Isto é também uma instância muito especial de um algoritmo geral de passagem de mensagens (Aji & McEliece, 2000). Combinando recursões para frente e para trás, somos capazes de calcular

$$P(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j) \rho_j(h_j) P(x_j \mid h_j). \tag{9.4.6}$$

Observe que, em termos abstratos, a recursão para trás pode ser escrita como $\rho_{t-1} = g(\rho_t, x_t)$, onde g é uma função que pode ser aprendida. Novamente, isso se parece muito com uma equação de atualização, apenas rodando para trás, ao contrário do que vimos até agora em RNNs. Na verdade, os modelos ocultos de Markov se beneficiam do conhecimento de dados futuros, quando estiverem disponíveis. Os cientistas de processamento de sinais distinguem entre os dois casos de conhecimento e não conhecimento de observações futuras como interpolação vs. extrapolação. Veja o capítulo introdutório do livro sobre algoritmos sequenciais de Monte Carlo para mais detalhes (Doucet et al., 2001).

9.4.2 Modelo Bidirecional

Se quisermos ter um mecanismo em RNNs que ofereça capacidade de antecipação comparável aos modelos de Markov ocultos, precisamos modificar o projeto RNN que vimos até agora. Felizmente, isso é fácil conceitualmente. Em vez de executar um RNN apenas no modo de encaminhamento a partir do primeiro token, iniciamos outro a partir do último token executado de trás para a frente. *RNNs bidirecionais* adicionam uma camada oculta que transmite informações para trás para processar essas informações com mais flexibilidade. Fig. 9.4.2 ilustra a arquitetura de um RNN bidirecional com uma única camada oculta.

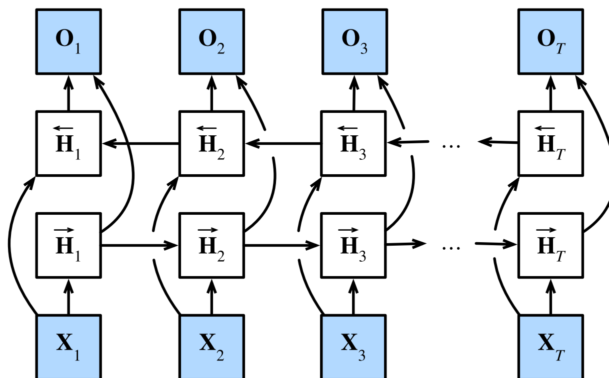


Fig. 9.4.2: Arquitetura de RNN bidirecional.

Na verdade, isso não é muito diferente das recursões para frente e para trás na programação dinâmica de modelos ocultos de Markov. A principal diferença é que no caso anterior essas equações tinham um significado estatístico específico. Agora eles estão desprovidos de tais interpretações facilmente acessíveis e podemos apenas tratá-los como funções genéricas e aprendíveis. Essa transição resume muitos dos princípios que orientam o projeto de redes profundas modernas: primeiro, use o tipo de dependências funcionais dos modelos estatísticos clássicos e, em seguida, os parametrize de uma forma genérica.

Definição

RNNs bidirecionais foram introduzidos por (Schuster & Paliwal, 1997). Para uma discussão detalhada das várias arquiteturas, consulte também o artigo (Graves & Schmidhuber, 2005). Vejamos as especificidades dessa rede.

Para qualquer etapa de tempo t, dado um minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (número de exemplos: n, número de entradas em cada exemplo: d) e deixe o oculto a função de ativação da camada seja ϕ. Na arquitetura bidirecional, assumimos que os estados ocultos para frente e para trás para este intervalo de tempo são $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ e $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$, respectivamente, onde h é o número de unidades ocultas. As atualizações de estado oculto para frente e para trás são as seguintes:

$$\begin{aligned}
\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\
\overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),
\end{aligned} \tag{9.4.7}$$

onde os pesos $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, e vieses $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ são todos os parâmetros do modelo.

A seguir, concatenamos os estados ocultos para frente e para trás $\overrightarrow{\mathbf{H}}_t$ e $\overleftarrow{\mathbf{H}}_t$ para obter o estado oculto $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ para alimentar a camada de saída. Em RNNs bidirecionais profundos com

várias camadas ocultas, tal informação é passado como *entrada* para a próxima camada bidirecional. Por último, a camada de saída calcula a saída $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (número de saídas: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{9.4.8}$$

Aqui, a matriz de peso $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ e o viés $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ são os parâmetros do modelo da camada de saída. Na verdade, as duas direções podem ter diferentes números de unidades ocultas.

Custo computacional e aplicativos

Uma das principais características de um RNN bidirecional é que as informações de ambas as extremidades da sequência são usadas para estimar a saída. Ou seja, usamos informações de observações futuras e passadas para prever a atual. No caso da previsão do próximo token, isso não é bem o que desejamos. Afinal, não podemos nos dar ao luxo de saber o próximo token ao prever o próximo. Portanto, se fôssemos usar um RNN bidirecional ingenuamente, não obteríamos uma precisão muito boa: durante o treinamento, temos dados passados e futuros para estimar o presente. Durante o tempo de teste, temos apenas dados anteriores e, portanto, pouca precisão. Vamos ilustrar isso em um experimento abaixo.

Para piorar a situação, os RNNs bidirecionais também são excessivamente lentos. As principais razões para isso são que a propagação para a frente requer recursões para frente e para trás em camadas bidirecionais e que a retropropagação depende dos resultados da propagação direta. Portanto, os gradientes terão uma cadeia de dependência muito longa.

Na prática, as camadas bidirecionais são usadas com muito moderação e apenas para um conjunto estreito de aplicações, como preencher palavras que faltam, tokens de anotação (por exemplo, para reconhecimento de entidade nomeada) e sequências de codificação por atacado como uma etapa em um pipeline de processamento de sequência (por exemplo, para tradução automática). Em Section 14.7 e Section 15.2, apresentaremos como usar RNNs bidirecionais para codificar sequências de texto.

9.4.3 Treinar um RNN bidirecional para uma aplicação errada

Se ignorarmos todos os conselhos sobre o fato de que RNNs bidirecionais usam dados passados e futuros e simplesmente os aplicam a modelos de linguagem, obteremos estimativas com perplexidade aceitável. No entanto, a capacidade do modelo de prever tokens futuros está seriamente comprometida, conforme ilustra o experimento abaixo. Apesar da perplexidade razoável, ele só gera rabiscos mesmo depois de muitas iterações. Incluímos o código abaixo como um exemplo de advertência contra usá-los no contexto errado.

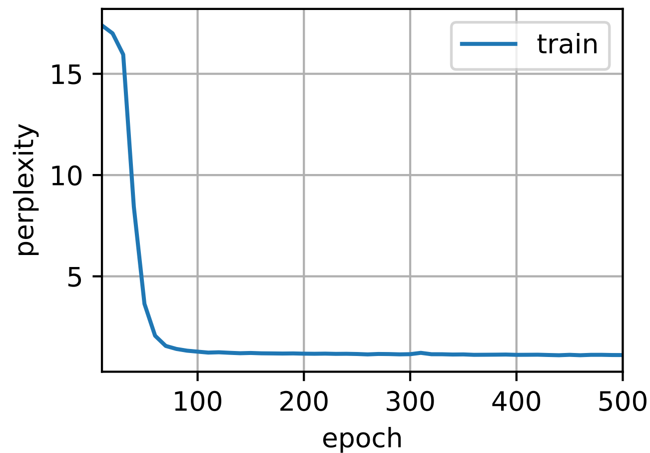
```
import torch
from torch import nn
from d2l import torch as d2l

# Load data
batch_size, num_steps, device = 32, 35, d2l.try_gpu()
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Define the bidirectional LSTM model by setting `bidirectional=True`
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers, bidirectional=True)
```

(continues on next page)

(continued from previous page)

```
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
# Train the model
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 110977.8 tokens/sec on cuda:0
time travellerererererererererererererererererererererererer
travellererererererererererererererererererererererererer
```



O resultado é claramente insatisfatório pelos motivos descritos acima. Para discussão de usos mais eficazes de RNNs bidirecionais, consulte o sentimento aplicação de análise em Section 15.2.

9.4.4 Sumário

- Em RNNs bidirecionais, o estado oculto para cada intervalo de tempo é simultaneamente determinado pelos dados antes e depois do intervalo de tempo atual.
- Os RNNs bidirecionais apresentam uma semelhança notável com o algoritmo para frente e para trás em modelos gráficos probabilísticos.
- RNNs bidirecionais são mais úteis para codificação de sequência e estimativa de observações em contexto bidirecional.
- RNNs bidirecionais são muito caros para treinar devido às longas cadeias de gradiente.

9.4.5 Exercícios

1. Se as diferentes direções usarem um número diferente de unidades ocultas, como a forma de $\mathbf{H}_t$ mudará?
2. Projete um RNN bidirecional com várias camadas ocultas.
3. A polissemia é comum em línguas naturais. Por exemplo, a palavra "banco" tem significados diferentes em contextos "fui ao banco para depositar dinheiro" e "fui ao banco para me sentar". Como podemos projetar um modelo de rede neural de modo que, dada uma sequência de contexto e uma palavra, uma representação vetorial da palavra no contexto seja retornada? Que tipo de arquitetura neural é preferida para lidar com a polissemia?

Discussão[95]

9.5 Tradução Automática e o Conjunto de Dados

Usamos RNNs para projetar modelos de linguagem, que são essenciais para o processamento de linguagem natural. Outro benchmark emblemático é a *tradução automática*, um domínio de problema central para modelos de *transdução de sequência* que transformam sequências de entrada em sequências de saída. Desempenhando um papel crucial em várias aplicações modernas de IA, modelos de transdução de sequência formarão o foco do restante deste capítulo e Chapter 10. Para este fim, esta seção apresenta o problema da tradução automática e seu conjunto de dados que será usado posteriormente.

Tradução automática refere-se ao tradução automática de uma sequência de um idioma para outro. Na verdade, este campo pode remontar a 1940 logo depois que os computadores digitais foram inventados, especialmente considerando o uso de computadores para decifrar códigos de linguagem na Segunda Guerra Mundial. Por décadas, abordagens estatísticas tinha sido dominante neste campo (Brown et al., 1988)[Brown.Cocke.Della-Pietra.ea.1990] antes da ascensão de aprendizagem ponta a ponta usando redes neurais. O último é frequentemente chamado *tradução automática neural* para se distinguir de *tradução automática de estatística* que envolve análise estatística em componentes como o modelo de tradução e o modelo de linguagem.

Enfatizando o aprendizado de ponta a ponta, este livro se concentrará em métodos de tradução automática neural. Diferente do nosso problema de modelo de linguagem in Section 8.3 cujo corpus está em um único idioma, conjuntos de dados de tradução automática são compostos por pares de sequências de texto que estão em o idioma de origem e o idioma de destino, respectivamente. Desse modo, em vez de reutilizar a rotina de pré-processamento para modelagem de linguagem, precisamos de uma maneira diferente de pré-processar conjuntos de dados de tradução automática. Na sequência, nós mostramos como carregar os dados pré-processados em minibatches para treinamento.

```
import os
import torch
from d2l import torch as d2l
```

[95] https://discuss.d2l.ai/t/1059

9.5.1 Download e Pré-processamento do Conjunto de Dados

Começar com, baixamos um conjunto de dados inglês-francês que consiste em pares de frases bilíngues do Projeto Tatoeba[96]. Cada linha no conjunto de dados é um par delimitado por tabulação de uma sequência de texto em inglês e a sequência de texto traduzida em francês. Observe que cada sequência de texto pode ser apenas uma frase ou um parágrafo de várias frases. Neste problema de tradução automática onde o inglês é traduzido para o francês, Inglês é o *idioma de origem* e o francês é o *idioma de destino*.

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    """Load the English-French dataset."""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[:75])
```

```
Downloading ../data/fra-eng.zip from http://d2l-data.s3-accelerate.amazonaws.com/fra-eng.zip.
↪..
Go. Va !
Hi. Salut !
Run!        Cours !
Run!        Courez !
Who?        Qui ?
Wow!        Ça alors !
```

Depois de baixar o conjunto de dados, continuamos com várias etapas de pré-processamento para os dados de texto brutos. Por exemplo, substituímos o espaço ininterrupto por espaço, converter letras maiúsculas em minúsculas, e insira espaço entre palavras e sinais de pontuação.

```
#@save
def preprocess_nmt(text):
    """Preprocess the English-French dataset."""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # Replace non-breaking space with space, and convert uppercase letters to
    # lowercase ones
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # Insert space between words and punctuation marks
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[:80])
```

[96] http://www.manythings.org/anki/

```
go .        va !
hi .        salut !
run !       cours !
run !       courez !
who ?       qui ?
wow !       ça alors !
```

9.5.2 Tokenização

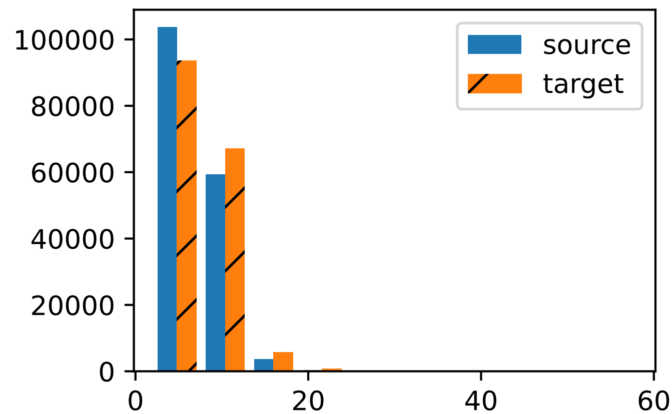
Diferente da tokenização em nível de personagem in:numref:`sec_language_model`, para tradução automática preferimos tokenização em nível de palavra aqui (modelos de última geração podem usar técnicas de tokenização mais avançadas). A seguinte função `tokenize_nmt` tokeniza os primeiros pares de sequência de texto `num_examples`, Onde cada token é uma palavra ou um sinal de pontuação. Esta função retorna duas listas de listas de tokens: `source` e`target`. Especificamente, `source [i]` é uma lista de tokens do i^{th} sequência de texto no idioma de origem (inglês aqui) e `target [i]` é a do idioma de destino (francês aqui).

```
#@save
def tokenize_nmt(text, num_examples=None):
    """Tokenize the English-French dataset."""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[:6], target[:6]
```

```
([['go', '.'],
  ['hi', '.'],
  ['run', '!'],
  ['run', '!'],
  ['who', '?'],
  ['wow', '!']],
 [['va', '!'],
  ['salut', '!'],
  ['cours', '!'],
  ['courez', '!'],
  ['qui', '?'],
  ['ça', 'alors', '!']])
```

Deixe-nos representar graficamente o histograma do número de tokens por sequência de texto. Neste conjunto de dados inglês-francês simples, a maioria das sequências de texto tem menos de 20 tokens.

```python
d2l.set_figsize()
_, _, patches = d2l.plt.hist(
    [[len(l) for l in source], [len(l) for l in target]],
    label=['source', 'target'])
for patch in patches[1].patches:
    patch.set_hatch('/')
d2l.plt.legend(loc='upper right');
```



9.5.3 Vocabulário

Uma vez que o conjunto de dados da tradução automática consiste em pares de línguas, podemos construir dois vocabulários para tanto o idioma de origem quanto o idioma de destino separadamente. Com tokenização em nível de palavra, o tamanho do vocabulário será significativamente maior do que usando tokenização em nível de caractere. Para aliviar isso, aqui tratamos tokens infrequentes que aparecem menos de 2 vezes como o mesmo token desconhecido ("<bos>"). Além disso, especificamos tokens especiais adicionais como para sequências de preenchimento ("<bos>") com o mesmo comprimento em minibatches, e para marcar o início ("<bos>") ou o fim ("<bos>") das sequências. Esses tokens especiais são comumente usados em tarefas de processamento de linguagem natural.

```python
src_vocab = d2l.Vocab(source, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)
```

```
10012
```

9.5.4 Carregando o Conjunto de Dados

Lembre-se de que na modelagem de linguagem cada exemplo de sequência, ou um segmento de uma frase ou uma extensão de várias frases, tem um comprimento fixo. Isso foi especificado pelo `num_steps` (número de etapas de tempo ou tokens) argumento em Section 8.3. Na tradução automática, cada exemplo é um par de sequências de texto de origem e destino, onde cada sequência de texto pode ter comprimentos diferentes.

Para eficiência computacional, ainda podemos processar um minibatch de sequências de texto ao mesmo tempo por *truncamento* e *preenchimento*. Suponha que cada sequência no mesmo mini-

batch deve ter o mesmo comprimento num_steps. Se uma sequência de texto tiver menos de tokens num_steps, continuaremos acrescentando a seção especial “<pad>” símbolo ao final até que seu comprimento alcance num_steps. Por outro lado, vamos truncar a sequência de texto pegando apenas seus primeiros tokens num_steps e descartando o restante. Desta maneira, cada sequência de texto terá o mesmo comprimento para ser carregado em minibatches do mesmo formato.

A seguinte função truncate_pad trunca ou preenche sequências de texto conforme descrito anteriormente.

```
#@save
def truncate_pad(line, num_steps, padding_token):
    """Truncate or pad sequences."""
    if len(line) > num_steps:
        return line[:num_steps]  # Truncate
    return line + [padding_token] * (num_steps - len(line))  # Pad

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])
```

```
[47, 4, 1, 1, 1, 1, 1, 1, 1, 1]
```

Agora definimos uma função para transformar sequências de texto em minibatches para treinamento. Anexamos o especial “<eos>” símbolo ao final de cada sequência para indicar o fim da sequência. Quando um modelo está prevendo de gerar um token de sequência após o token, a geração do “<eos>” símbolo pode sugerir que a sequência de saída está completa. Além do mais, nós também gravamos o comprimento de cada sequência de texto, excluindo os tokens de preenchimento. Esta informação será necessária por alguns modelos que nós cobriremos mais tarde.

```
#@save
def build_array_nmt(lines, vocab, num_steps):
    """Transform text sequences of machine translation into minibatches."""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = torch.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
    return array, valid_len
```

9.5.5 Juntando todas as coisas

Finalmente, definimos a função load_data_nmt para retornar o iterador de dados, junto com os vocabulários do idioma de origem e do idioma de destino.

```
#@save
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """Return the iterator and the vocabularies of the translation dataset."""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
```

(continues on next page)

(continued from previous page)

```
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

Vamos ler o primeiro minibatch do conjunto de dados inglês-francês.

```
train_iter, src_vocab, tgt_vocab = load_data_nmt(batch_size=2, num_steps=8)
for X, X_valid_len, Y, Y_valid_len in train_iter:
    print('X:', X.type(torch.int32))
    print('valid lengths for X:', X_valid_len)
    print('Y:', Y.type(torch.int32))
    print('valid lengths for Y:', Y_valid_len)
    break
```

```
X: tensor([[  6,  18, 153,   4,   3,   1,   1,   1],
        [166,  15,   5,   3,   1,   1,   1,   1]], dtype=torch.int32)
valid lengths for X: tensor([5, 4])
Y: tensor([[  6,   7,  60,   4,   3,   1,   1,   1],
        [  0,  50, 174,   5,   3,   1,   1,   1]], dtype=torch.int32)
valid lengths for Y: tensor([5, 5])
```

9.5.6 Sumário

- Tradução automática refere-se à tradução automática de uma sequência de um idioma para outro.
- Usando tokenização em nível de palavra, o tamanho do vocabulário será significativamente maior do que usando tokenização em nível de caractere. Para aliviar isso, podemos tratar tokens raros como o mesmo token desconhecido.
- Podemos truncar e preencher sequências de texto para que todas tenham o mesmo comprimento para serem carregadas em minibatches.

9.5.7 Exercícios

1. Tente valores diferentes do argumento `num_examples` na função`load_data_nmt`. Como isso afeta os tamanhos do vocabulário do idioma de origem e do idioma de destino?
2. O texto em alguns idiomas, como chinês e japonês, não tem indicadores de limite de palavras (por exemplo, espaço). A tokenização em nível de palavra ainda é uma boa ideia para esses casos? Por que ou por que não?

Discussions[97]

[97] https://discuss.d2l.ai/t/1060

9.6 Arquitetura Encoder-Decoder

Como discutimos em Section 9.5, maquina de tradução é um domínio de problema principal para modelos de transdução de sequência, cuja entrada e saída são ambas as sequências de comprimento variável. Para lidar com este tipo de entradas e saídas, podemos projetar uma arquitetura com dois componentes principais. O primeiro componente é um *codificador*: ele pega uma sequência de comprimento variável como entrada e a transforma em um estado com uma forma fixa. O segundo componente é um *decodificador*: ele mapeia o estado codificado de uma forma fixa a uma sequência de comprimento variável. Isso é chamado de arquitetura *codificador-decodificador*, que é representado em Fig. 9.6.1.

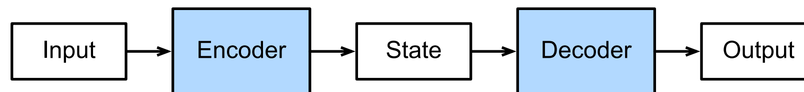


Fig. 9.6.1: A arquitetura encoder-decoder.

Vamos fazer uma tradução automática de inglês para francês como um exemplo. Dada uma sequência de entrada em inglês: "They", "are", "watching", ".", esta arquitetura de codificador-decodificador primeiro codifica a entrada de comprimento variável em um estado, então decodifica o estado para gerar o token de sequência traduzido por token como saída: "Ils", "respectent", ".". Uma vez que a arquitetura codificador-decodificador forma a base de diferentes modelos de transdução de sequência nas seções subsequentes, esta seção irá converter esta arquitetura em uma interface que será implementada posteriormente.

9.6.1 Encoder

Na interface do codificador, nós apenas especificamos isso o codificador recebe sequências de comprimento variável como `X` de entrada. A implementação será fornecida por qualquer modelo que herde esta classe `Encoder` base.

```python
from torch import nn


#@save
class Encoder(nn.Module):
    """The base encoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError
```

9.6.2 Decoder

Na seguinte interface do decodificador, adicionamos uma função adicional `init_state` para converter a saída do codificador (`enc_outputs`) no estado codificado. Observe que esta etapa pode precisar de entradas extras, como o comprimento válido da entrada, o que foi explicado in Section 9.5.4. Para gerar um token de sequência de comprimento variável por token, toda vez que o decodificador pode mapear uma entrada (por exemplo, o token gerado na etapa de tempo anterior) e o estado codificado em um token de saída na etapa de tempo atual.

```
#@save
class Decoder(nn.Module):
    """The base decoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

9.6.3 Somando o Encoder e o Decoder

No fim, a arquitetura codificador-decodificador contém um codificador e um decodificador, com argumentos opcionalmente extras. Na propagação direta, a saída do codificador é usado para produzir o estado codificado, e este estado será posteriormente usado pelo decodificador como uma de suas entradas.

```
#@save
class EncoderDecoder(nn.Module):
    """The base class for the encoder-decoder architecture."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

O termo "estado" na arquitetura do codificador-decodificador provavelmente inspirou você a implementar este arquitetura usando redes neurais com estados. Na próxima seção, veremos como aplicar RNNs para projetar modelos de transdução de sequência baseados em esta arquitetura de codificador-decodificador.

9.6.4 Sumário

- A arquitetura do codificador-decodificador pode lidar com entradas e saídas que são sequências de comprimento variável, portanto, é adequada para problemas de transdução de sequência, como tradução automática.
- O codificador pega uma sequência de comprimento variável como entrada e a transforma em um estado com forma fixa.
- O decodificador mapeia o estado codificado de uma forma fixa para uma sequência de comprimento variável.

9.6.5 Exercícios

1. Suponha que usamos redes neurais para implementar a arquitetura codificador-decodificador. O codificador e o decodificador precisam ser do mesmo tipo de rede neural?
2. Além da tradução automática, você consegue pensar em outro aplicativo em que a arquitetura codificador-decodificador possa ser aplicada?

Discussão[98]

9.7 Aprendizado Sequência a Sequência

Como vimos em Section 9.5, na tradução automática tanto a entrada quanto a saída são uma sequência de comprimento variável. Para resolver este tipo de problema, nós projetamos uma arquitetura geral de codificador-decodificador in Section 9.6. Nesta secção, nós vamos use dois RNNs para projetar o codificador e o decodificador de esta arquitetura e aplicá-lo a *sequência a sequência* de aprendizagem para tradução automática (Sutskever et al., 2014)[Cho.Van-Merrienboer.Gulcehre.ea.2014].

Seguindo o princípio de design da arquitetura codificador-decodificador, o codificador RNN pode pega uma sequência de comprimento variável como entrada e a transforma em um estado oculto de forma fixa. Em outras palavras, informação da sequência de entrada (fonte) é *codificado* no estado oculto do codificador RNN. Para gerar o token de sequência de saída por token, um decodificador RNN separado pode prever o próximo token com base em quais tokens foram vistos (como na modelagem de linguagem) ou gerados, junto com as informações codificadas da sequência de entrada. Fig. 9.7.1 ilustra como usar dois RNNs para aprendizagem de sequência para sequência na tradução automática.

[98] https://discuss.d2l.ai/t/1061

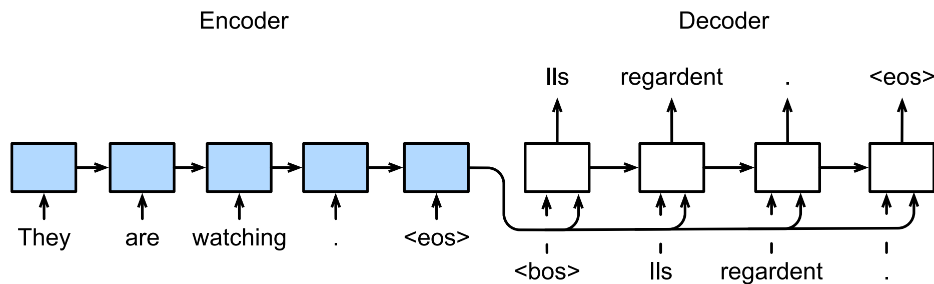


Fig. 9.7.1: Aprendizado sequência a sequência com um codificador RNN e um decodificador RNN.

Em Fig. 9.7.1, o especial "<bos>" símbolo marca o fim da sequência. O modelo pode parar de fazer previsões uma vez que este token é gerado. Na etapa de tempo inicial do decodificador RNN, existem duas decisões de design especiais. Primeiro, o início de sequência especial "<bos>" token é uma entrada. Segundo, o estado final oculto do codificador RNN é usado para iniciar o estado oculto do decodificador. Em designs como (Sutskever et al., 2014), isto é exatamente como as informações da sequência de entrada codificada é alimentado no decodificador para gerar a sequência de saída (destino). Em alguns outros designs, como (Cho et al., 2014b), o estado oculto final do codificador também é alimentado no decodificador como parte das entradas em cada etapa de tempo, conforme mostrado em Fig. 9.7.1. Semelhante ao treinamento de modelos de linguagem em Section 8.3, podemos permitir que os rótulos sejam a sequência de saída original, deslocado por um token: "<bos>", "Ils", "regardent", "." → "Ils", "regardent", ".", "<eos>".

Na sequência, vamos explicar o design de Fig. 9.7.1 em maiores detalhes. Vamos treinar este modelo para tradução automática no conjunto de dados inglês-francês, conforme apresentado em Section 9.5.

```
import collections
import math
import torch
from torch import nn
from d2l import torch as d2l
```

9.7.1 Encoder

Tecnicamente falando, o codificador transforma uma sequência de entrada de comprimento variável em um formato fixo *variável de contexto* $mathbf{c}$ e codifica as informações da sequência de entrada nesta variável de contexto. Conforme descrito em Fig. 9.7.1, podemos usar um RNN para projetar o codificador.

Vamos considerar um exemplo de sequência (tamanho do lote: 1). Suponha que a sequência de entrada é $x_1, \ldots, x_T$, de modo que x_t é o token t^{th} na sequência de texto de entrada. No passo de tempo t, o RNN transforma o vetor de característica de entrada $\mathbf{x}_t$ para x_t e o estado oculto $\mathbf{h}_{t-1}$ da etapa de tempo anterior no estado oculto atual $\mathbf{h}_t$. Podemos usar a função f para expressar a transformação da camada recorrente do RNN:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{9.7.1}$$

Em geral, o codificador transforma os estados ocultos em todos os passos do tempo na variável de contexto por meio de uma função personalizada q:

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T). \tag{9.7.2}$$

Por exemplo, ao escolher $q(\mathbf{h}_1, \ldots, \mathbf{h}_T) = \mathbf{h}_T$ como em Fig. 9.7.1, a variável de contexto é apenas o estado oculto $\mathbf{h}_T$ da sequência de entrada na etapa de tempo final.

Até agora, usamos um RNN unidirecional para projetar o codificador, Onde um estado oculto depende apenas de a subsequência de entrada na e antes da etapa de tempo do estado oculto. Também podemos construir codificadores usando RNNs bidirecionais. Neste caso, um estado oculto depende de a subsequência antes e depois da etapa de tempo (incluindo a entrada na etapa de tempo atual), que codifica as informações de toda a sequência.

Agora, vamos implementar o codificador RNN. Observe que usamos uma *camada de incorporação* para obter o vetor de recurso para cada token na sequência de entrada. O peso de uma camada de incorporação é uma matriz cujo número de linhas é igual ao tamanho do vocabulário de entrada (`vocab_size`) e o número de colunas é igual à dimensão do vetor de recursos (`embed_size`). Para qualquer índice de token de entrada i, a camada de incorporação busca a ith linha (começando em 0) da matriz de peso para retornar seu vetor de recurso. Além do mais, aqui, escolhemos um GRU multicamadas para implementar o codificador.

```
#@save
class Seq2SeqEncoder(d2l.Encoder):
    """The RNN encoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers,
                          dropout=dropout)

    def forward(self, X, *args):
        # The output `X` shape: (`batch_size`, `num_steps`, `embed_size`)
        X = self.embedding(X)
        # In RNN models, the first axis corresponds to time steps
        X = X.permute(1, 0, 2)
        # When state is not mentioned, it defaults to zeros
        output, state = self.rnn(X)
        # `output` shape: (`num_steps`, `batch_size`, `num_hiddens`)
        # `state` shape: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

As variáveis retornadas de camadas recorrentes foram explicados em Section 8.6. Vamos ainda usar um exemplo concreto para ilustrar a implementação do codificador acima. Abaixo de nós instanciamos um codificador GRU de duas camadas cujo número de unidades ocultas é 16. Dado um minibatch de entradas de sequência `X` (tamanho do lote: 4, número de etapas de tempo: 7), os estados ocultos da última camada em todas as etapas do tempo (retorno de `saída` pelas camadas recorrentes do codificador) são um tensor de forma (número de etapas de tempo, tamanho do lote, número de unidades ocultas).

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)
output, state = encoder(X)
output.shape
```

```
torch.Size([7, 4, 16])
```

Uma vez que um GRU é empregado aqui, a forma dos estados ocultos multicamadas na etapa final do tempo é (número de camadas ocultas, tamanho do lote, número de unidades ocultas). Se um LSTM for usado, as informações da célula de memória também estarão contidas em `estado`.

```
state.shape
```

```
torch.Size([2, 4, 16])
```

9.7.2 Decoder

Como acabamos de mencionar, a variável de contexto $\mathbf{c}$ da saída do codificador codifica toda a sequência de entrada $x_1, \ldots, x_T$. Dada a sequência de saída $y_1, y_2, \ldots, y_{T'}$ do conjunto de dados de treinamento, para cada passo de tempo t' (o símbolo difere da etapa de tempo t das sequências de entrada ou codificadores), a probabilidade de saída do decodificador $y_{t'}$ é condicional na subsequência de saída anterior $y_1, \ldots, y_{t'-1}$ e a variável de contexto $\mathbf{c}$, i.e., $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$.

Para modelar essa probabilidade condicional em sequências, podemos usar outro RNN como decodificador. A qualquer momento, passo t' na sequência de saída, o RNN pega a saída $y_{t'-1}$ da etapa de tempo anterior e a variável de contexto $\mathbf{c}$ como sua entrada, então se transforma eles e o estado oculto anterior $\mathbf{s}_{t'-1}$ no estado oculto $\mathbf{s}_{t'}$ no intervalo de tempo atual. Como resultado, podemos usar uma função g para expressar a transformação da camada oculta do decodificador:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{9.7.3}$$

Depois de obter o estado oculto do decodificador, podemos usar uma camada de saída e a operação softmax para calcular a distribuição de probabilidade condicional $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$ para a saída na etapa de tempo t'.

Seguindo Fig. 9.7.1, ao implementar o decodificador da seguinte forma, usamos diretamente o estado oculto na etapa de tempo final do codificador para inicializar o estado oculto do decodificador. Isso requer que o codificador RNN e o decodificador RNN tenham o mesmo número de camadas e unidades ocultas. Para incorporar ainda mais as informações da sequência de entrada codificada, a variável de contexto é concatenada com a entrada do decodificador em todas as etapas de tempo. Para prever a distribuição de probabilidade do token de saída, uma camada totalmente conectada é usada para transformar o estado oculto na camada final do decodificador RNN.

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens, num_layers,
                          dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, *args):
```

(continues on next page)

(continued from previous page)

```
        return enc_outputs[1]

    def forward(self, X, state):
        # The output `X` shape: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).permute(1, 0, 2)
        # Broadcast `context` so it has the same `num_steps` as `X`
        context = state[-1].repeat(X.shape[0], 1, 1)
        X_and_context = torch.cat((X, context), 2)
        output, state = self.rnn(X_and_context, state)
        output = self.dense(output).permute(1, 0, 2)
        # `output` shape: (`batch_size`, `num_steps`, `vocab_size`)
        # `state` shape: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

Para ilustrar o decodificador implementado, abaixo, nós o instanciamos com os mesmos hiperparâmetros do codificador mencionado. Como podemos ver, a forma de saída do decodificador torna-se (tamanho do lote, número de etapas de tempo, tamanho do vocabulário), onde a última dimensão do tensor armazena a distribuição de token prevista.

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
decoder.eval()
state = decoder.init_state(encoder(X))
output, state = decoder(X, state)
output.shape, state.shape
```

```
(torch.Size([4, 7, 10]), torch.Size([2, 4, 16]))
```

Para resumir, as camadas no modelo de codificador-decodificador RNN acima são ilustradas em Fig. 9.7.2.

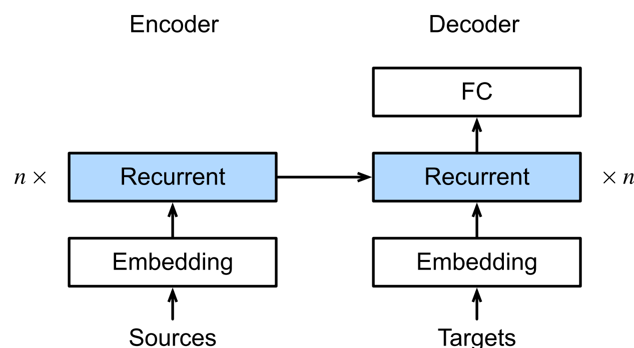


Fig. 9.7.2: Camadas em um modelo de codificador-decodificador RNN.

9.7.3 Função de Perdas

Em cada etapa de tempo, o decodificador prevê uma distribuição de probabilidade para os tokens de saída. Semelhante à modelagem de linguagem, podemos aplicar softmax para obter a distribuição e calcular a perda de entropia cruzada para otimização. Lembre-se de Section 9.5 que os tokens de preenchimento especiais são anexados ao final das sequências então sequências de comprimentos variados pode ser carregado de forma eficiente em minibatches da mesma forma. Contudo, previsão de tokens de preenchimento devem ser excluídos dos cálculos de perdas.

Para este fim, podemos usar o seguinte função `sequence_mask` para mascarar entradas irrelevantes com valores zero então mais tarde multiplicação de qualquer previsão irrelevante com zero igual a zero. Por exemplo, se o comprimento válido de duas sequências excluindo tokens de preenchimento são um e dois, respectivamente, as entradas restantes após o primeiro e as duas primeiras entradas são zeradas.

```
#@save
def sequence_mask(X, valid_len, value=0):
    """Mask irrelevant entries in sequences."""
    maxlen = X.size(1)
    mask = torch.arange((maxlen), dtype=torch.float32,
                        device=X.device)[None, :] < valid_len[:, None]
    X[~mask] = value
    return X

X = torch.tensor([[1, 2, 3], [4, 5, 6]])
sequence_mask(X, torch.tensor([1, 2]))
```

```
tensor([[1, 0, 0],
        [4, 5, 0]])
```

Também podemos mascarar todas as entradas do último alguns eixos. Se quiser, você pode até especificar para substituir essas entradas por um valor diferente de zero.

```
X = torch.ones(2, 3, 4)
sequence_mask(X, torch.tensor([1, 2]), value=-1)
```

```
tensor([[[ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.],
         [-1., -1., -1., -1.]],

        [[ 1.,  1.,  1.,  1.],
         [ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.]]])
```

Agora podemos estender a perda de entropia cruzada softmax para permitir o mascaramento de previsões irrelevantes. Inicialmente, máscaras para todos os tokens previstos são definidas como um. Uma vez que o comprimento válido é fornecido, a máscara correspondente a qualquer token de preenchimento será zerado. No fim, a perda de todos os tokens será multiplicado pela máscara para filtrar previsões irrelevantes de tokens de preenchimento na perda.

```
#@save
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
```

(continues on next page)

(continued from previous page)

```
    """The softmax cross-entropy loss with masks."""
    # `pred` shape: (`batch_size`, `num_steps`, `vocab_size`)
    # `label` shape: (`batch_size`, `num_steps`)
    # `valid_len` shape: (`batch_size`,)
    def forward(self, pred, label, valid_len):
        weights = torch.ones_like(label)
        weights = sequence_mask(weights, valid_len)
        self.reduction='none'
        unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
            pred.permute(0, 2, 1), label)
        weighted_loss = (unweighted_loss * weights).mean(dim=1)
        return weighted_loss
```

Para uma verificação de sanidade, podemos criar três sequências idênticas. Então nós podemos especificar que os comprimentos válidos dessas sequências são 4, 2 e 0, respectivamente. Como resultado, a perda da primeira sequência deve ser duas vezes maior que o da segunda sequência, enquanto a terceira sequência deve ter uma perda zero.

```
loss = MaskedSoftmaxCELoss()
loss(torch.ones(3, 4, 10), torch.ones((3, 4), dtype=torch.long),
     torch.tensor([4, 2, 0]))
```

```
tensor([2.3026, 1.1513, 0.0000])
```

9.7.4 Treinamento

No ciclo de treinamento a seguir, nós concatenamos o token especial de início de sequência e a sequência de saída original excluindo o token final como a entrada para o decodificador, conforme mostrado em Fig. 9.7.1. Isso é chamado de *aprendizado forçado* porque a sequência de saída original (rótulos de token) é alimentada no decodificador. Alternativamente, também poderíamos alimentar o token *predito* da etapa de tempo anterior como a entrada atual para o decodificador.

```
#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
    """Train a model for sequence to sequence."""
    def xavier_init_weights(m):
        if type(m) == nn.Linear:
            nn.init.xavier_uniform_(m.weight)
        if type(m) == nn.GRU:
            for param in m._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(m._parameters[param])
    net.apply(xavier_init_weights)
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    loss = MaskedSoftmaxCELoss()
    net.train()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
```

(continues on next page)

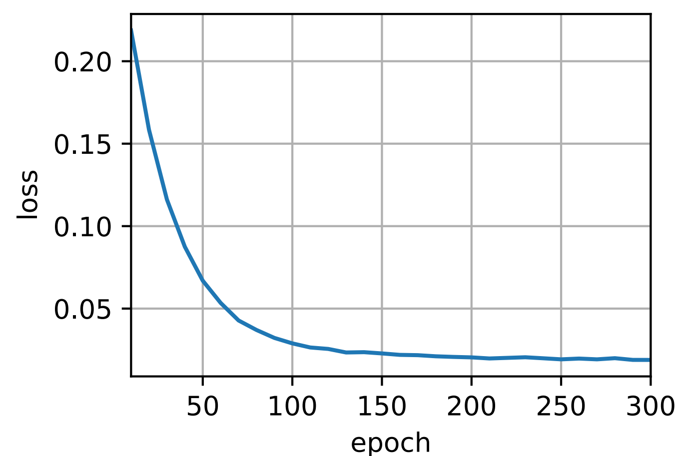
(continued from previous page)

```
        metric = d2l.Accumulator(2)  # Sum of training loss, no. of tokens
        for batch in data_iter:
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
                               device=device).reshape(-1, 1)
            dec_input = torch.cat([bos, Y[:, :-1]], 1)  # Teacher forcing
            Y_hat, _ = net(X, dec_input, X_valid_len)
            l = loss(Y_hat, Y, Y_valid_len)
            l.sum().backward()  # Make the loss scalar for `backward`
            d2l.grad_clipping(net, 1)
            num_tokens = Y_valid_len.sum()
            optimizer.step()
            with torch.no_grad():
                metric.add(l.sum(), num_tokens)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')
```

Agora podemos criar e treinar um modelo de codificador-decodificador RNN para aprendizado de sequência para sequência no conjunto de dados de tradução automática.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 300, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.019, 10054.7 tokens/sec on cuda:0
```



9.7.5 Predição

Para prever a sequência de saída token por token, em cada etapa de tempo do decodificador o token previsto do anterior intervalo de tempo é alimentado no decodificador como uma entrada. Semelhante ao treinamento, na etapa de tempo inicial o token de início de sequência ("<eos>") é alimentado no decodificador. Este processo de previsão é ilustrado em Fig. 9.7.3. Quando o token de fim de sequência ("<eos>") é previsto, a previsão da sequência de saída está completa.

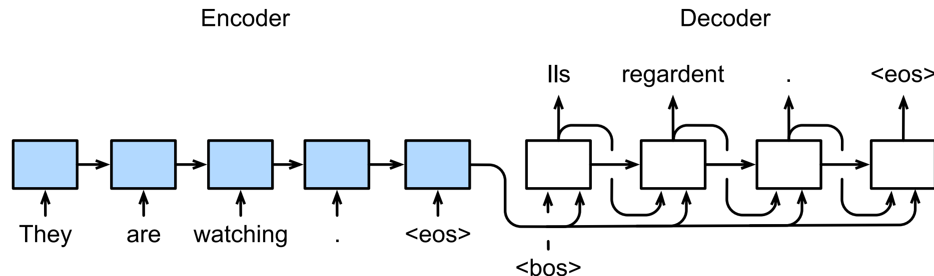


Fig. 9.7.3: Predicting the output sequence token by token using an RNN encoder-decoder.

Vamos apresentar diferentes estratégias para geração de sequência em Section 9.8.

```
#@save
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device, save_attention_weights=False):
    """Predict for sequence to sequence."""
    # Set `net` to eval mode for inference
    net.eval()
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # Add the batch axis
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device), dim=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # Add the batch axis
    dec_X = torch.unsqueeze(torch.tensor(
        [tgt_vocab['<bos>']], dtype=torch.long, device=device), dim=0)
    output_seq, attention_weight_seq = [], []
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        # We use the token with the highest prediction likelihood as the input
        # of the decoder at the next time step
        dec_X = Y.argmax(dim=2)
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # Once the end-of-sequence token is predicted, the generation of the
        # output sequence is complete
        if pred == tgt_vocab['<eos>']:
            break
```

(continues on next page)

(continued from previous page)

```
        output_seq.append(pred)
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

9.7.6 Avaliação de Sequências Preditas

Podemos avaliar uma sequência prevista comparando-o com o sequência de rótulos (a verdade fundamental). BLEU (Understudy de Avaliação Bilíngue), embora originalmente proposto para avaliação resultados da tradução automática (Papineni et al., 2002), tem sido amplamente utilizado na medição a qualidade das sequências de saída para diferentes aplicações. Em princípio, para quaisquer n-gramas na sequência prevista, BLEU avalia se este n-grams aparece na sequência do rótulo.

Denotado por p_n a precisão de n-grams, qual é a proporção de o número de n-grams correspondentes em as sequências preditas e rotuladas para o número de n-gramas na sequência prevista. Explicar, dada uma sequência de rótulo A, B, C, D, E, F, e uma sequência prevista A, B, B, C, D, temos $p_1 = 4/5$, $p_2 = 3/4$, $p_3 = 1/3$, e $p_4 = 0$. Além do mais, deixe $\text{len}_{\text{label}}$ e len_{pred} ser os números de tokens na sequência do rótulo e na sequência prevista, respectivamente. Então, BLEU é definido como

$$\exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}, \tag{9.7.4}$$

onde k são os n-gramas mais longos para correspondência.

Com base na definição de BLEU em (9.7.4), sempre que a sequência prevista for igual à sequência do rótulo, BLEU será 1. Além disso, já que combinar n-gramas mais longos é mais difícil, BLEU atribui um peso maior para uma precisão maior de n-gram. Especificamente, quando p_n é corrigido, $p_n^{1/2^n}$ aumenta à medida que n cresce (o artigo original usa $p_n^{1/n}$). Além disso, Desde a predição de sequências mais curtas tende a obter um valor maior de p_n, o coeficiente antes do termo de multiplicação em (9.7.4) penaliza sequências preditas mais curtas. Por exemplo, quando $k = 2$, dada a sequência de rótulo A, B, C, D, E, F e a sequência prevista A, B, embora $p_1 = p_2 = 1$, o fator de penalidade $\exp(1 - 6/2) \approx 0.14$ reduz o BLEU.

Implementamos a medida BLEU da seguinte forma.

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[''.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[''.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[''.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

No fim, usamos o codificador-decodificador RNN treinado traduzir algumas frases em inglês para o francês e calcular o BLEU dos resultados.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, attention_weight_seq = predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device)
    print(f'{eng} => {translation}, bleu {bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => il est paresseux ., bleu 0.658
i'm home . => je suis chez moi en pas question ., bleu 0.640
```

9.7.7 Sumário

- Seguindo o projeto da arquitetura do codificador-decodificador, podemos usar dois RNNs para projetar um modelo para o aprendizado de sequência para sequência.
- Ao implementar o codificador e o decodificador, podemos usar RNNs multicamadas.
- Podemos usar máscaras para filtrar cálculos irrelevantes, como ao calcular a perda.
- No treinamento de codificador-decodificador, a abordagem de força do professor alimenta as sequências de saída originais (em contraste com as previsões) para o decodificador.
- BLEU é uma medida popular para avaliar sequências de saída combinando n-gramas entre a sequência prevista e a sequência de rótulo.

9.7.8 Exercícios

1. Você pode ajustar os hiperparâmetros para melhorar os resultados da tradução?
2. Repita o experimento sem usar máscaras no cálculo da perda. Que resultados você observa? Por quê?
3. Se o codificador e o decodificador diferem no número de camadas ou no número de unidades ocultas, como podemos inicializar o estado oculto do decodificador?
4. No treinamento, substitua o forçamento do professor com a alimentação da previsão da etapa de tempo anterior no decodificador. Como isso influencia o desempenho?
5. Execute novamente o experimento substituindo GRU por LSTM.
6. Existem outras maneiras de projetar a camada de saída do decodificador?

Discussão[99]

[99] https://discuss.d2l.ai/t/1062

9.8 Pesquisa de feixe

Em Section 9.7, previmos a sequência de saída token por token até o final da sequênceia especial “<eos>” token é predito. Nesta secção, começaremos formalizando essa estratégia de *busca gananciosa* e explorando problemas com isso, em seguida, comparamos essa estratégia com outras alternativas: * pesquisa exaustiva* e *pesquisa por feixe*.

Antes de uma introdução formal à busca gananciosa, vamos formalizar o problema de pesquisa usando a mesma notação matemática de Section 9.7. A qualquer momento, passo t', a probabilidade de saída do decodificador $y_{t'}$ é condicional na subsequência de saída $y_1, \ldots, y_{t'-1}$ antes de t' e a variável de contexto $\mathbf{c}$ que codifica as informações da sequência de entrada. Para quantificar o custo computacional, denotar por $\mathcal{Y}$ (contém “<eos>”) o vocabulário de saída. Portanto, a cardinalidade $|\mathcal{Y}|$ deste conjunto de vocabulário é o tamanho do vocabulário. Vamos também especificar o número máximo de tokens de uma sequência de saída como T'. Como resultado, nosso objetivo é procurar um resultado ideal de todo o $\mathcal{O}(|\mathcal{Y}|^{T'})$ possíveis sequências de saída. Claro, para todas essas sequências de saída, porções incluindo e após “<eos>” será descartado na saída real.

9.8.1 Busca Gulosa

Primeiro, vamos dar uma olhada em uma estratégia simples: *busca gananciosa*. Esta estratégia foi usada para prever sequências em Section 9.7. Em busca gananciosa, a qualquer momento, etapa t' da sequência de saída, nós procuramos pelo token com a maior probabilidade condicional de $\mathcal{Y}$, ou seja,

$$y_{t'} = \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{9.8.1}$$

como a saída. Uma vez que “<eos>” é emitida ou a sequência de saída atingiu seu comprimento máximo T', a sequência de saída é concluída.

Então, o que pode dar errado com a busca gananciosa? Na verdade, a *sequência ideal* deve ser a sequência de saída com o máximo $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$, qual é a probabilidade condicional de gerar uma sequência de saída com base na sequência de entrada. Infelizmente, não há garantia que a sequência ótima será obtida por busca gananciosa.

Time step	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<eos>	0.1	0.2	0.2	0.6

Fig. 9.8.1: At each time step, greedy search selects the token with the highest conditional probability.

Vamos ilustrar com um exemplo. Suponha que existam quatro tokens “A”, “B”, “C” e “<eos>” no dicionário de saída. Em: numref: `fig_s2s-prob1`, os quatro números em cada etapa de tempo representam as probabilidades condicionais de gerar “A”, “B”, “C” e “<eos>” nessa etapa de tempo,

respectivamente. Em cada etapa de tempo, a pesquisa gananciosa seleciona o token com a probabilidade condicional mais alta. Portanto, a sequência de saída "A", "B", "C" e "<eos>" será previsto in Fig. 9.8.1. A probabilidade condicional desta sequência de saída é $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$.

Time step	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<eos>	0.1	0.2	0.1	0.6

Fig. 9.8.2: Os quatro números em cada etapa de tempo representam as probabilidades condicionais de gerar "A", "B", "C" e "<eos>" nessa etapa de tempo. Na etapa de tempo 2, o token "C", que tem a segunda maior probabilidade condicional, é selecionado.

A seguir, vejamos outro exemplo in Fig. 9.8.2. Ao contrário de Fig. 9.8.1, no passo de tempo 2 selecionamos o token "C" in Fig. 9.8.2, que tem a *segunda* probabilidade condicional mais alta. Uma vez que as subseqüências de saída nas etapas de tempo 1 e 2, em que a etapa de tempo 3 se baseia, mudaram de "A" e "B" em Fig. 9.8.1 para"A" e "C" em Fig. 9.8.2, a probabilidade condicional de cada token na etapa 3 também mudou em Fig. 9.8.2. Suponha que escolhemos o token "B" na etapa de tempo 3. Agora, a etapa 4 está condicionada a a subseqüência de saída nas três primeiras etapas de tempo "A", "C" e "B", que é diferente de "A", "B" e "C" em Fig. 9.8.1. Portanto, a probabilidade condicional de gerar cada token na etapa de tempo 4 em Fig. 9.8.2 também é diferente daquela em Fig. 9.8.1. Como resultado, a probabilidade condicional da sequência de saída "A", "C", "B" e "<eos>" in Fig. 9.8.2 é $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$, que é maior do que a busca gananciosa em Fig. 9.8.1. Neste exemplo, a sequência de saída "A", "B", "C" e "<eos>" obtida pela busca gananciosa não é uma sequência ótima.

9.8.2 Busca Exaustiva

Se o objetivo é obter a sequência ideal, podemos considerar o uso de *pesquisa exaustiva*: enumerar exaustivamente todas as sequências de saída possíveis com suas probabilidades condicionais, em seguida, envie o um com a probabilidade condicional mais alta.

Embora possamos usar uma pesquisa exaustiva para obter a sequência ideal, seu custo computacional $\mathcal{O}(|\mathcal{Y}|^{T'})$ é provavelmente excessivamente alto. Por exemplo, quando $|\mathcal{Y}| = 10000$ e $T' = 10$, precisaremos avaliar $10000^{10} = 10^{40}$ sequências. Isso é quase impossível! Por outro lado, o custo computacional da busca gananciosa é $\mathcal{O}(|\mathcal{Y}| T')$: geralmente é significativamente menor do que o da pesquisa exaustiva. Por exemplo, quando $|\mathcal{Y}| = 10000$ e$T' = 10$, só precisamos avaliar $10000 \times 10 = 10^5$ sequências.

9.8.3 Busca de Feixe

Decisões sobre estratégias de busca de sequência mentem em um espectro, com perguntas fáceis em qualquer um dos extremos. E se apenas a precisão importasse? Obviamente, pesquisa exaustiva. E se apenas o custo computacional importa? Claramente, busca gananciosa. Um aplicativo do mundo real geralmente pergunta uma pergunta complicada, em algum lugar entre esses dois extremos.

Pesquisa de feixe é uma versão aprimorada da pesquisa gananciosa. Ele tem um hiperparâmetro denominado *tamanho do feixe*, k. Na etapa de tempo 1, selecionamos k tokens com as probabilidades condicionais mais altas. Cada um deles será o primeiro símbolo de k sequências de saída candidatas, respectivamente. Em cada etapa de tempo subsequente, com base nas sequências de saída do candidato k na etapa de tempo anterior, continuamos a selecionar sequências de saída candidatas a k com as maiores probabilidades condicionais de $k\left|\mathcal{Y}\right|$ escolhas possíveis.

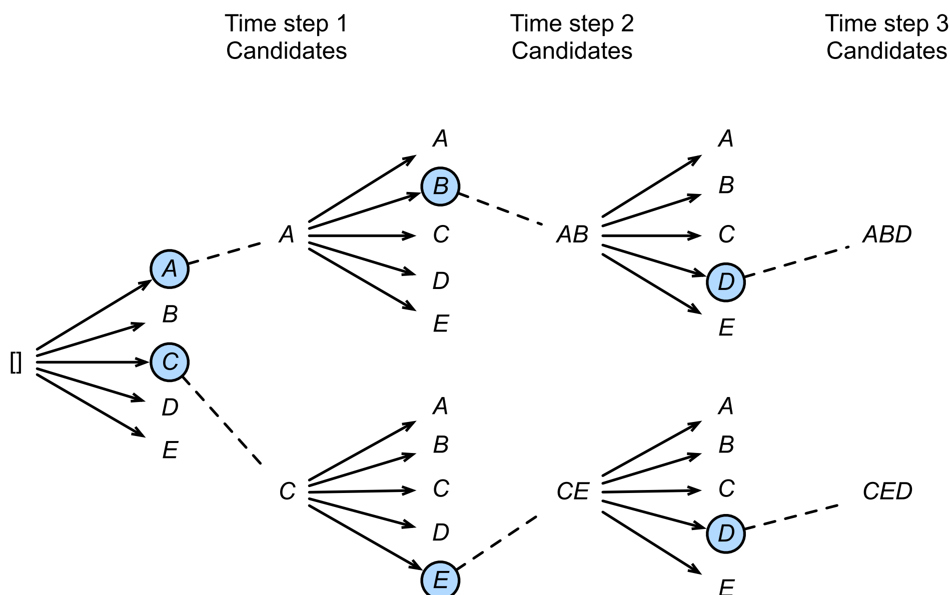


Fig. 9.8.3: O processo de busca do feixe (tamanho do feixe: 2, comprimento máximo de uma sequência de saída: 3). As sequências de saída candidatas são A, C, AB, CE, ABD, eCED.

: numref: `fig_beam-search` demonstra o processo de pesquisa de feixe com um exemplo. Suponha que o vocabulário de saída contém apenas cinco elementos: $\mathcal{Y} = \{A, B, C, D, E\}$, onde um deles é "<eos>". Deixe o tamanho do feixe ser 2 e o comprimento máximo de uma sequência de saída é 3. Na etapa de tempo 1, suponha que os tokens com as probabilidades condicionais mais altas $P(y_1 \mid \mathbf{c})$ sejam A e C No passo de tempo 2, para todos os $y_2 \in \mathcal{Y}$, calculamos

$$\begin{aligned}P(A, y_2 \mid \mathbf{c}) = P(A \mid \mathbf{c})P(y_2 \mid A, \mathbf{c}),\\ P(C, y_2 \mid \mathbf{c}) = P(C \mid \mathbf{c})P(y_2 \mid C, \mathbf{c}),\end{aligned} \tag{9.8.2}$$

e escolha os dois maiores entre esses dez valores, digamos $P(A, B \mid \mathbf{c})$ e$P(C, E \mid \mathbf{c})$. Depois para o passo de tempo 3, para todos $y_3 \in \mathcal{Y}$, nós computamos

$$\begin{aligned}P(A, B, y_3 \mid \mathbf{c}) = P(A, B \mid \mathbf{c})P(y_3 \mid A, B, \mathbf{c}),\\ P(C, E, y_3 \mid \mathbf{c}) = P(C, E \mid \mathbf{c})P(y_3 \mid C, E, \mathbf{c}),\end{aligned} \tag{9.8.3}$$

e escolha os dois maiores entre esses dez valores, digamos $P(A, B, D \mid \mathbf{c})$ e $P(C, E, D \mid \mathbf{c})$. Como resultado, obtemos seis sequências de saída de candidatos: (i) A; (ii) C; (iii) A, B; (iv) C, E; (v) A, B, D; e (vi) C, E, D.

No final, obtemos o conjunto de sequências de saída candidatas finais com base nessas seis sequências (por exemplo, descarte porções incluindo e após “<eos>”). Então escolhemos a sequência com a maior das seguintes pontuações como a sequência de saída:

$$\frac{1}{L^\alpha} \log P(y_1, \ldots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{9.8.4}$$

onde L é o comprimento da sequência candidata final e α é geralmente definido como 0,75. Uma vez que uma sequência mais longa tem mais termos logarítmicos na soma de (9.8.4), o termo L^α no denominador penaliza longas sequências.

O custo computacional da pesquisa do feixe é $\mathcal{O}(k\,|\mathcal{Y}|\,T')$. Esse resultado está entre o da busca gananciosa e o da busca exaustiva. Na verdade, a pesquisa gananciosa pode ser tratada como um tipo especial de pesquisa de feixe com um tamanho de feixe de 1. Com uma escolha flexível do tamanho do feixe, pesquisa de feixe fornece uma compensação entre precisão versus custo computacional.

9.8.4 Sumário

- As estratégias de busca de sequência incluem busca gananciosa, busca exaustiva e busca de feixe.
- A pesquisa de feixe oferece uma compensação entre precisão e custo computacional por meio de sua escolha flexível do tamanho do feixe.

9.8.5 Exercícios

1. Podemos tratar a pesquisa exaustiva como um tipo especial de pesquisa de feixe? Por que ou por que não?
2. Aplique a pesquisa de feixe no problema de tradução automática em Section 9.7. Como o tamanho do feixe afeta os resultados da tradução e a velocidade de previsão?
3. Usamos modelagem de linguagem para gerar texto seguindo prefixos fornecidos pelo usuário em Section 8.5. Que tipo de estratégia de pesquisa ele usa? Você pode melhorar isso?

Discussão[100]

[100] https://discuss.d2l.ai/t/338

10 | Mecanismos de Atenção

O nervo óptico do sistema visual de um primata recebe entrada sensorial massiva, excedendo em muito o que o cérebro pode processar totalmente. Felizmente, nem todos os estímulos são criados iguais. Focalização e concentração de consciência permitiram que os primatas direcionassem a atenção para objetos de interesse, como presas e predadores, no ambiente visual complexo. A capacidade de prestar atenção a apenas uma pequena fração das informações tem significado evolutivo, permitindo seres humanos para viver e ter sucesso.

Os cientistas têm estudado a atenção no campo da neurociência cognitiva desde o século XIX. Neste capítulo, começaremos revisando uma estrutura popular explicando como a atenção é implantada em uma cena visual. Inspirado pelas dicas de atenção neste quadro, nós iremos projetar modelos que alavancam tais dicas de atenção. Notavelmente, a regressão do kernel Nadaraya-Waston em 1964 é uma demonstração simples de aprendizado de máquina com *mecanismos de atenção*.

A seguir, iremos apresentar as funções de atenção que têm sido amplamente usadas em o desenho de modelos de atenção em *deep learning*. Especificamente, vamos mostrar como usar essas funções para projetar a *atenção Bahdanau*, um modelo de atenção inovador em *deep learning* que pode se alinhar bidirecionalmente e é diferenciável.

No fim, equipados com a mais recente *atenção de várias cabeças* e designs de *autoatenção*, iremos descrever a arquitetura do *transformador* baseado unicamente em mecanismos de atenção. Desde sua proposta em 2017, transformadores têm sido difundidos na modernidade aplicativos de *deep learning*, como em áreas de língua, visão, fala, e aprendizagem por reforço.

10.1 Dicas para atenção

Obrigado pela sua atenção a este livro. Atenção é um recurso escasso: no momento você está lendo este livro e ignorando o resto. Assim, semelhante ao dinheiro, sua atenção está sendo paga com um custo de oportunidade. Para garantir que seu investimento de atenção agora vale a pena, estamos altamente motivados a prestar nossa atenção com cuidado para produzir um bom livro. Atenção é a pedra angular do arco da vida e detém a chave para o excepcionalismo de qualquer trabalho.

Já que a economia estuda a alocação de recursos escassos, Nós estamos na era da economia da atenção, onde a atenção humana é tratada como uma mercadoria limitada, valiosa e escassa que pode ser trocada. Numerosos modelos de negócios foram desenvolvido para capitalizar sobre ele. Em serviços de streaming de música ou vídeo, ou prestamos atenção aos seus anúncios ou pagar para escondê-los. Para crescer no mundo dos jogos online, nós ou prestamos atenção a participar de batalhas, que atraem novos jogadores, ou pagamos dinheiro para nos tornarmos poderosos instantaneamente. Nada vem de graça.

Contudo, as informações em nosso ambiente não são escassas, atenção é. Ao inspecionar uma cena visual, nosso nervo óptico recebe informações na ordem de 10^8 bits por segundo, excedendo em muito o que nosso cérebro pode processar totalmente. Felizmente, nossos ancestrais aprenderam com a experiência (também conhecido como dados) que *nem todas as entradas sensoriais são criadas iguais*. Ao longo da história humana, a capacidade de direcionar a atenção para apenas uma fração da informação de interesse habilitou nosso cérebro para alocar recursos de forma mais inteligente para sobreviver, crescer e se socializar, como a detecção de predadores, presas e companheiros.

10.1.1 Dicas de Atenção em Biologia

Para explicar como nossa atenção é implantada no mundo visual, uma estrutura de dois componentes surgiu e tem sido generalizado. Essa ideia remonta a William James na década de 1890, que é considerado o "pai da psicologia americana" (James, 2007). Nesta estrutura, assuntos direcionam seletivamente o holofote da atenção usando a *dica não-voluntária* e a *dica volitiva*.

A sugestão não-voluntária é baseada em a saliência e conspicuidade de objetos no ambiente. Imagine que há cinco objetos à sua frente: um jornal, um artigo de pesquisa, uma xícara de café, um caderno e um livro como em Fig. 10.1.1. Embora todos os produtos de papel sejam impressos em preto e branco, a xícara de café é vermelha. Em outras palavras, este café é intrinsecamente saliente e conspícuo neste ambiente visual, chamando a atenção automática e involuntariamente. Então você traz a fóvea (o centro da mácula onde a acuidade visual é mais alta) para o café como mostrado em Fig. 10.1.1.

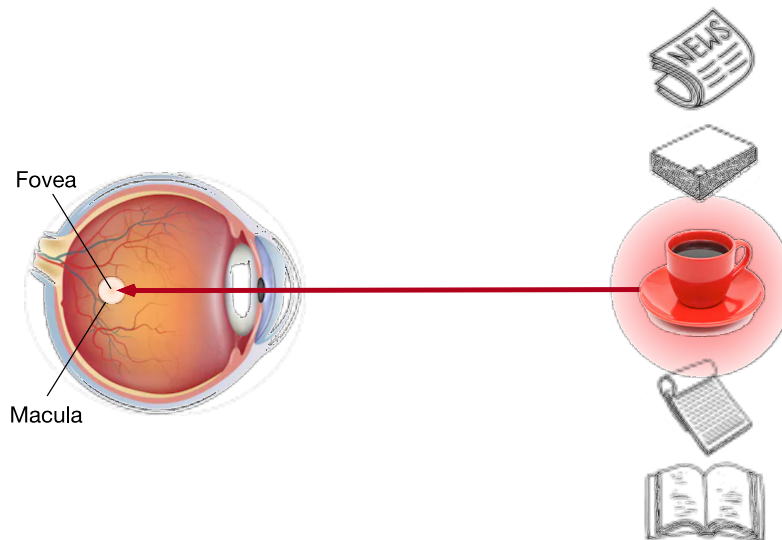


Fig. 10.1.1: Usando a sugestão não-voluntária baseada na saliência (xícara vermelha, não papel), a atenção é involuntariamente voltada para o café.

Depois de beber café, você se torna cafeinado e quer ler um livro. Então você vira sua cabeça, reorienta seus olhos, e olha para o livro conforme descrito em Fig. 10.1.2. Diferente do caso em Fig. 10.1.1 onde o café o inclina para selecionar com base na saliência, neste caso dependente da tarefa, você seleciona o livro em controle cognitivo e volitivo. Usando a dica volitiva com base em critérios de seleção de variáveis, esta forma de atenção é mais deliberada. Também é mais poderoso com o esforço voluntário do sujeito.

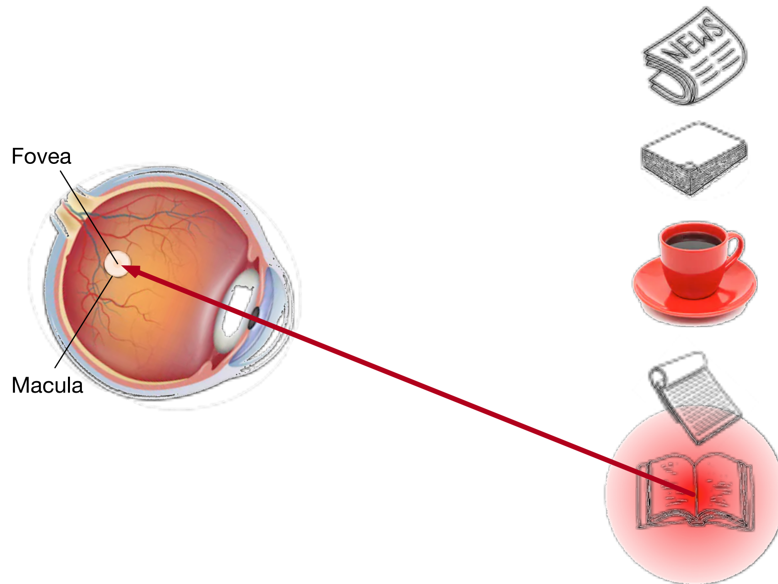


Fig. 10.1.2: Usando a dica volitiva (quero ler um livro) que depende da tarefa, a atenção é direcionada para o livro sob controle volitivo.

10.1.2 Consultas, Chaves e Valores

Inspirado pelas dicas de atenção não-voluntárias e volitivas que explicam a implantação da atenção, a seguir iremos descrever uma estrutura para projetando mecanismos de atenção incorporando essas duas pistas de atenção.

Para começar, considere o caso mais simples, onde apenas sugestões não-tradicionais estão disponíveis. Para influenciar a seleção sobre as entradas sensoriais, podemos simplesmente usar uma camada parametrizada totalmente conectada ou mesmo não parametrizada de agrupamento máximo ou médio.

Portanto, o que define mecanismos de atenção além dessas camadas totalmente conectadas ou camadas de *pooling* é a inclusão das dicas volitivas. No contexto dos mecanismos de atenção, nos referimos às dicas volitivas como *consultas*. Dada qualquer consulta, mecanismos de atenção seleção de *bias* sobre entradas sensoriais (por exemplo, representações de recursos intermediários) via *pooling de atenção*. Essas entradas sensoriais são chamadas de *valores* no contexto dos mecanismos de atenção. De forma geral, cada valor é emparelhado com uma *chave*, que pode ser pensado como a sugestão não-voluntária dessa entrada sensorial. Conforme mostrado em Fig. 10.1.3, podemos projetar concentração de atenção para que a consulta dada (dica volitiva) possa interagir com as chaves (dicas não-volitivas), que orienta a seleção de *bias* sobre os valores (entradas sensoriais).

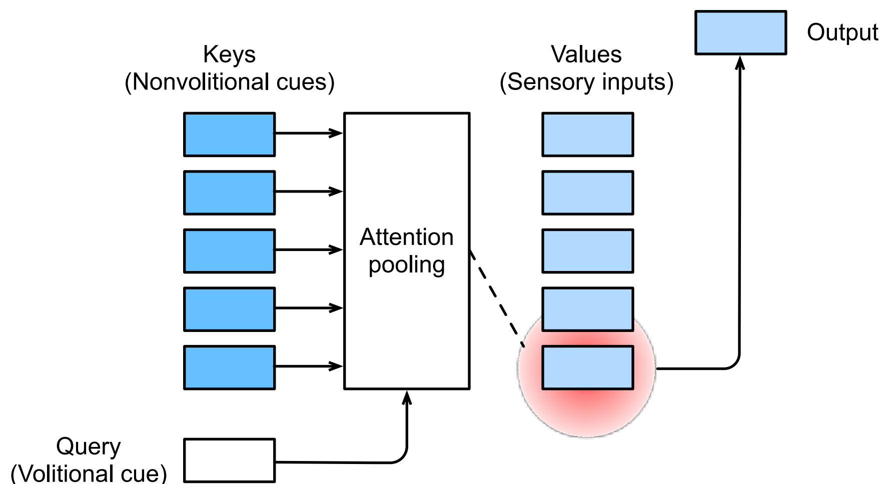


Fig. 10.1.3: Os mecanismos de atenção colocam *bias* na seleção sobre os valores (entradas sensoriais) por meio do agrupamento de atenção, que incorpora consultas (dicas volitivas) e chaves (dicas não-volitivas).

Observe que existem muitas alternativas para o design de mecanismos de atenção. Por exemplo, podemos projetar um modelo de atenção não diferenciável que pode ser treinado usando métodos de aprendizagem por reforço (Mnih et al., 2014). Dado o domínio do framework em Fig. 10.1.3, modelos sob esta estrutura serão o centro de nossa atenção neste capítulo.

10.1.3 Visualização da Atenção

Pooling médio pode ser tratado como uma média ponderada de entradas, onde os pesos são uniformes. Na prática, O *pooling* de atenção agrega valores usando a média ponderada, onde os pesos são calculados entre a consulta fornecida e chaves diferentes.

```
import torch
from d2l import torch as d2l
```

Para visualizar pesos de atenção, definimos a função `show_heatmaps`. Sua entrada `matrices` tem a forma (número de linhas para exibição, número de colunas para exibição, número de consultas, número de chaves).

```
#@save
def show_heatmaps(matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5),
                  cmap='Reds'):
    d2l.use_svg_display()
    num_rows, num_cols = matrices.shape[0], matrices.shape[1]
    fig, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize,
                                 sharex=True, sharey=True, squeeze=False)
    for i, (row_axes, row_matrices) in enumerate(zip(axes, matrices)):
        for j, (ax, matrix) in enumerate(zip(row_axes, row_matrices)):
            pcm = ax.imshow(matrix.detach().numpy(), cmap=cmap)
            if i == num_rows - 1:
                ax.set_xlabel(xlabel)
            if j == 0:
                ax.set_ylabel(ylabel)
```

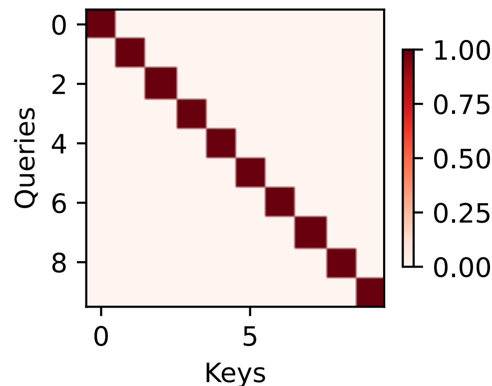
(continues on next page)

(continued from previous page)

```
        if titles:
            ax.set_title(titles[j])
fig.colorbar(pcm, ax=axes, shrink=0.6);
```

Para demonstração, consideramos um caso simples onde o peso da atenção é único apenas quando a consulta e a chave são as mesmas; caso contrário, é zero.

```
attention_weights = torch.eye(10).reshape((1, 1, 10, 10))
show_heatmaps(attention_weights, xlabel='Keys', ylabel='Queries')
```



Nas seções subsequentes, frequentemente invocaremos essa função para visualizar pesos de atenção.

10.1.4 Resumo

- A atenção humana é um recurso limitado, valioso e escasso.
- Os sujeitos direcionam seletivamente a atenção usando as dicas não-volitivas e volitivas. O primeiro é baseado na saliência e o último depende da tarefa.
- Os mecanismos de atenção são diferentes de camadas totalmente conectadas ou camadas de agrupamento devido à inclusão das dicas volitivas.
- Os mecanismos de atenção colocam um *bias* na seleção sobre os valores (entradas sensoriais) por meio do *pooling* de atenção, que incorpora consultas (dicas volitivas) e chaves (dicas não-volitivas). Chaves e valores estão emparelhados.
- Podemos visualizar pesos de atenção entre consultas e chaves.

10.1.5 Exercícios

1. Qual pode ser a dica volitiva ao decodificar um token de sequência por token na tradução automática? Quais são as dicas não-convencionais e as entradas sensoriais?
2. Gere aleatoriamente uma matriz 10×10 e use a operação softmax para garantir que cada linha seja uma distribuição de probabilidade válida. Visualize os pesos de atenção de saída.

Discussions[101]

10.2 *Pooling* de Atenção: Regressão de Kernel de Nadaraya-Watson

Agora você conhece os principais componentes dos mecanismos de atenção sob a estrutura em Fig. 10.1.3. Para recapitular, as interações entre consultas (dicas volitivas) e chaves (dicas não volitivas) resultam em *concentração de atenção*. O *pooling* de atenção agrega valores seletivamente (entradas sensoriais) para produzir a saída. Nesta secção, vamos descrever o agrupamento de atenção em mais detalhes para lhe dar uma visão de alto nível como os mecanismos de atenção funcionam na prática. Especificamente, o modelo de regressão do kernel de Nadaraya-Watson proposto em 1964 é um exemplo simples, mas completo para demonstrar o aprendizado de máquina com mecanismos de atenção.

```
import torch
from torch import nn
from d2l import torch as d2l
```

10.2.1 Gerando o Dataset

Para manter as coisas simples, vamos considerar o seguinte problema de regressão: dado um conjunto de dados de pares de entrada-saída $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, como aprender f para prever a saída $\hat{y} = f(x)$ para qualquer nova entrada x?

Aqui, geramos um conjunto de dados artificial de acordo com a seguinte função não linear com o termo de ruído ϵ:

$$y_i = 2\sin(x_i) + x_i^{0.8} + \epsilon, \tag{10.2.1}$$

onde ϵ obedece a uma distribuição normal com média zero e desvio padrão 0,5. Ambos, 50 exemplos de treinamento e 50 exemplos de teste são gerados. Para visualizar melhor o padrão de atenção posteriormente, as entradas de treinamento são classificadas.

```
n_train = 50  # No. of training examples
x_train, _ = torch.sort(torch.rand(n_train) * 5)   # Training inputs
```

```
def f(x):
    return 2 * torch.sin(x) + x**0.8

y_train = f(x_train) + torch.normal(0.0, 0.5, (n_train,))  # Training outputs
x_test = torch.arange(0, 5, 0.1)  # Testing examples
```

(continues on next page)

[101] https://discuss.d2l.ai/t/1592

(continued from previous page)

```
y_truth = f(x_test)  # Ground-truth outputs for the testing examples
n_test = len(x_test)  # No. of testing examples
n_test
```

```
50
```

A função a seguir plota todos os exemplos de treinamento (representados por círculos), a função de geração de dados de verdade básica `f` sem o termo de ruído (rotulado como "Truth"), e a função de predição aprendida (rotulado como "Pred").

```
def plot_kernel_reg(y_hat):
    d2l.plot(x_test, [y_truth, y_hat], 'x', 'y', legend=['Truth', 'Pred'],
             xlim=[0, 5], ylim=[-1, 5])
    d2l.plt.plot(x_train, y_train, 'o', alpha=0.5);
```

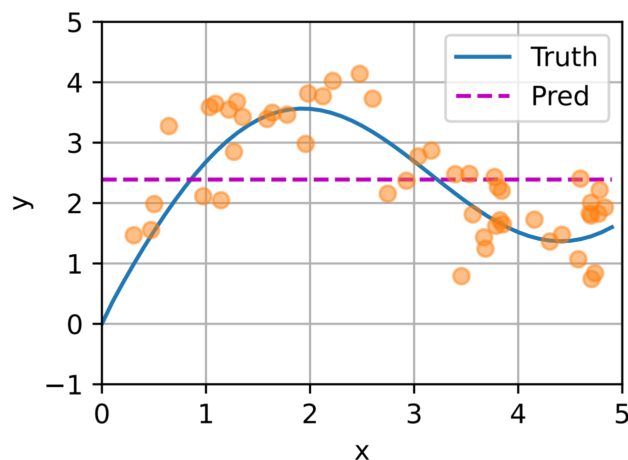
10.2.2 *Pooling* Médio

Começamos com talvez o estimador "mais idiota" do mundo para este problema de regressão: usando o *pooling* médio para calcular a média de todos os resultados do treinamento:

$$f(x) = \frac{1}{n}\sum_{i=1}^n y_i, \tag{10.2.2}$$

que é plotado abaixo. Como podemos ver, este estimador não é tão inteligente.

```
y_hat = torch.repeat_interleave(y_train.mean(), n_test)
plot_kernel_reg(y_hat)
```



10.2.3 *Pooling* de Atenção não-Paramétrico

Obviamente, o agrupamento médio omite as entradas x_i. Uma ideia melhor foi proposta por Nadaraya (Nadaraya, 1964) e Waston (Watson, 1964) para pesar as saídas y_i de acordo com seus locais de entrada:

$$f(x) = \sum_{i=1}^n \frac{K(x - x_i)}{\sum_{j=1}^n K(x - x_j)} y_i, \tag{10.2.3}$$

onde K é um *kernel*. O estimador em (10.2.3) é chamado de *regressão do kernel Nadaraya-Watson*. Aqui não entraremos em detalhes sobre os grãos. Lembre-se da estrutura dos mecanismos de atenção em Fig. 10.1.3. Do ponto de vista da atenção, podemos reescrever (10.2.3) em uma forma mais generalizada de *concentração de atenção*:

$$f(x) = \sum_{i=1}^n \alpha(x, x_i) y_i, \tag{10.2.4}$$

onde x é a consulta e (x_i, y_i) é o par de valores-chave. Comparando (10.2.4) e (10.2.2), a atenção concentrada aqui é uma média ponderada de valores y_i. O *peso de atenção* $\alpha(x, x_i)$ em (10.2.4) é atribuído ao valor correspondente y_i baseado na interação entre a consulta x e a chave x_i modelado por α. Para qualquer consulta, seus pesos de atenção sobre todos os pares de valores-chave são uma distribuição de probabilidade válida: eles não são negativos e somam um.

Para obter intuições de concentração de atenção, apenas considere um *kernel gaussiano* definido como

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{u^2}{2}). \tag{10.2.5}$$

Conectando o kernel gaussiano em (10.2.4) e (10.2.3) dá

$$\begin{aligned} f(x) &= \sum_{i=1}^n \alpha(x, x_i) y_i \\ &= \sum_{i=1}^n \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^n \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^n \mathrm{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned} \tag{10.2.6}$$

In (10.2.6), uma chave x_i que está mais próxima da consulta dada x obterá *mais atenção* por meio de um *peso de atenção maior* atribuído ao valor correspondente da chave y_i.

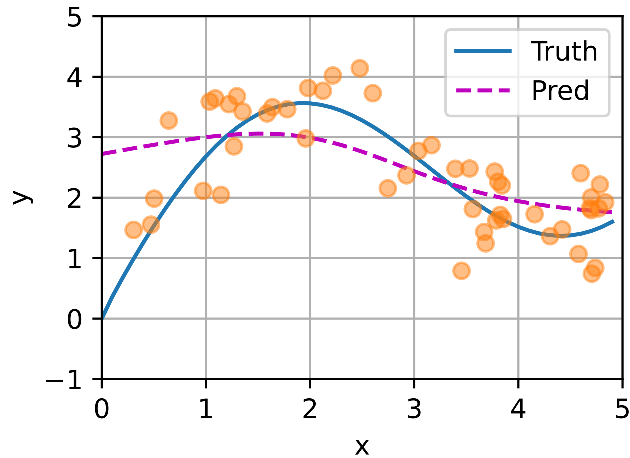
Notavelmente, a regressão do kernel Nadaraya-Watson é um modelo não paramétrico; assim (10.2.6) é um exemplo de *agrupamento de atenção não paramétrica*. A seguir, traçamos a previsão com base neste modelo de atenção não paramétrica. A linha prevista é suave e mais próxima da verdade fundamental do que a produzida pelo agrupamento médio.

```
# Shape of `X_repeat`: (`n_test`, `n_train`), where each row contains the
# same testing inputs (i.e., same queries)
X_repeat = x_test.repeat_interleave(n_train).reshape((-1, n_train))
# Note that `x_train` contains the keys. Shape of `attention_weights`:
# (`n_test`, `n_train`), where each row contains attention weights to be
# assigned among the values (`y_train`) given each query
```

(continues on next page)

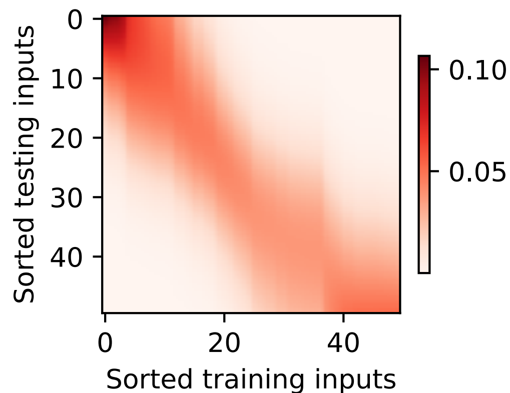
(continued from previous page)

```
attention_weights = nn.functional.softmax(-(X_repeat - x_train)**2 / 2, dim=1)
# Each element of `y_hat` is weighted average of values, where weights are
# attention weights
y_hat = torch.matmul(attention_weights, y_train)
plot_kernel_reg(y_hat)
```



Agora, vamos dar uma olhada nos pesos de atenção. Aqui, as entradas de teste são consultas, enquanto as entradas de treinamento são essenciais. Uma vez que ambas as entradas são classificadas, podemos ver que quanto mais próximo o par de chave de consulta está, o maior peso de atenção está no *pooling* de atenção.

```
d2l.show_heatmaps(attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')
```



10.2.4 *Pooling* de Atenção Paramétrica

A regressão de kernel não paramétrica de Nadaraya-Watson desfruta do benefício de *consistência*: com dados suficientes, esse modelo converge para a solução ótima. Não obstante, podemos facilmente integrar parâmetros aprendíveis no *pooling* de atenção.

Por exemplo, um pouco diferente de (10.2.6), na sequência a distância entre a consulta x e a chave x_i é multiplicado por um parâmetro aprendível w:

$$\begin{aligned} f(x) &= \sum_{i=1}^n \alpha(x, x_i) y_i \\ &= \sum_{i=1}^n \frac{\exp\left(-\frac{1}{2}((x - x_i)w)^2\right)}{\sum_{j=1}^n \exp\left(-\frac{1}{2}((x - x_i)w)^2\right)} y_i \\ &= \sum_{i=1}^n \mathrm{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i. \end{aligned} \tag{10.2.7}$$

No resto da seção, vamos treinar este modelo aprendendo o parâmetro de a concentração de atenção em (10.2.7).

Multiplicação de Matriz de Lote

Para computar a atenção com mais eficiência para *minibatches*, podemos aproveitar os utilitários de multiplicação de matrizes em lote fornecidos por *frameworks* de *deep learning*.

Suponha que o primeiro minibatch contém n matrizes $\mathbf{X}_1, \ldots, \mathbf{X}_n$ de forma $a \times b$, e o segundo minibatch contém n matrizes $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$ da forma $b \times c$. Sua multiplicação da matriz de lote resulta em n matrizes $\mathbf{X}_1\mathbf{Y}_1, \ldots, \mathbf{X}_n\mathbf{Y}_n$ da forma $a \times c$. Portanto, dados dois tensores de forma (n, a, b) e (n, b, c), a forma de sua saída de multiplicação da matriz em lote é (n, a, c).

```
X = torch.ones((2, 1, 4))
Y = torch.ones((2, 4, 6))
torch.bmm(X, Y).shape
```

```
torch.Size([2, 1, 6])
```

No contexto dos mecanismos de atenção, podemos usar a multiplicação da matriz de minibatch para calcular médias ponderadas de valores em um minibatch.

```
weights = torch.ones((2, 10)) * 0.1
values = torch.arange(20.0).reshape((2, 10))
torch.bmm(weights.unsqueeze(1), values.unsqueeze(-1))
```

```
tensor([[[ 4.5000]],

        [[14.5000]]])
```

Definindo o Modelo

Usando a multiplicação de matriz de minibatch, abaixo nós definimos a versão paramétrica da regressão do kernel Nadaraya-Watson com base no agrupamento de atenção paramétrica em (10.2.7).

```
class NWKernelRegression(nn.Module):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.w = nn.Parameter(torch.rand((1,), requires_grad=True))

    def forward(self, queries, keys, values):
        # Shape of the output `queries` and `attention_weights`:
        # (no. of queries, no. of key-value pairs)
        queries = queries.repeat_interleave(keys.shape[1]).reshape((-1, keys.shape[1]))
        self.attention_weights = nn.functional.softmax(
            -((queries - keys) * self.w)**2 / 2, dim=1)
        # Shape of `values`: (no. of queries, no. of key-value pairs)
        return torch.bmm(self.attention_weights.unsqueeze(1),
                         values.unsqueeze(-1)).reshape(-1)
```

Treinamento

A seguir, transformamos o conjunto de dados de treinamento às chaves e valores para treinar o modelo de atenção. No agrupamento paramétrico de atenção, qualquer entrada de treinamento pega pares de valores-chave de todos os exemplos de treinamento, exceto ela mesma, para prever sua saída.

```
# Shape of `X_tile`: (`n_train`, `n_train`), where each column contains the
# same training inputs
X_tile = x_train.repeat((n_train, 1))
# Shape of `Y_tile`: (`n_train`, `n_train`), where each column contains the
# same training outputs
Y_tile = y_train.repeat((n_train, 1))
# Shape of `keys`: ('n_train', 'n_train' - 1)
keys = X_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -1))
# Shape of `values`: ('n_train', 'n_train' - 1)
values = Y_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -1))
```

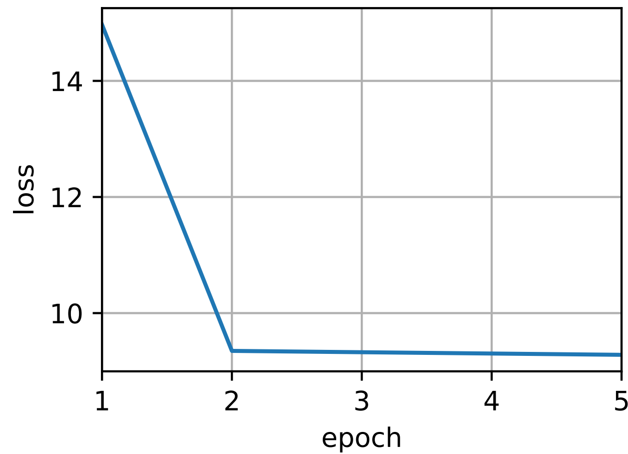
Usando a perda quadrada e a descida do gradiente estocástico, treinamos o modelo paramétrico de atenção.

```
net = NWKernelRegression()
loss = nn.MSELoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[1, 5])

for epoch in range(5):
    trainer.zero_grad()
    # Note: L2 Loss = 1/2 * MSE Loss. PyTorch has MSE Loss which is slightly
    # different from MXNet's L2Loss by a factor of 2. Hence we halve the loss
    l = loss(net(x_train, keys, values), y_train) / 2
    l.sum().backward()
```

(continues on next page)

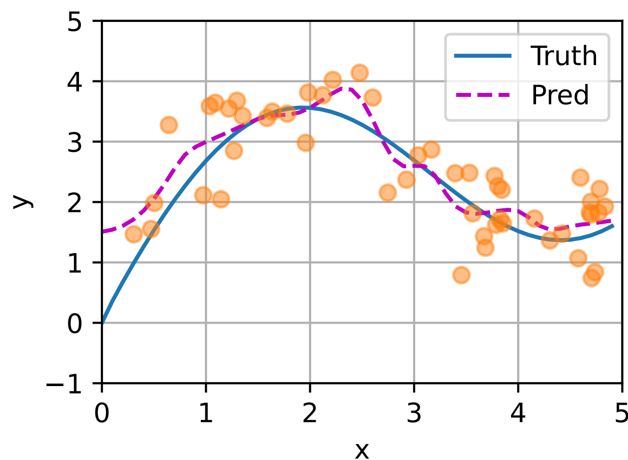
(continued from previous page)

```
    trainer.step()
    print(f'epoch {epoch + 1}, loss {float(l.sum()):.6f}')
    animator.add(epoch + 1, float(l.sum()))
```



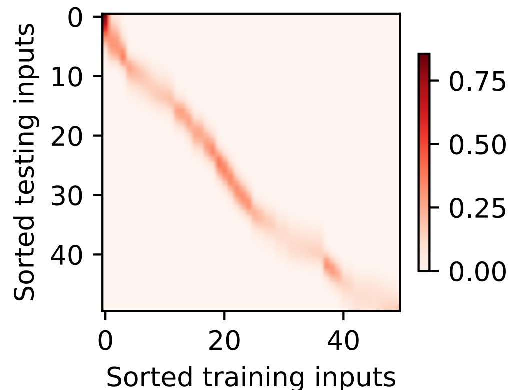
Depois de treinar o modelo paramétrico de atenção, podemos traçar sua previsão. Tentando ajustar o conjunto de dados de treinamento com ruído, a linha prevista é menos suave do que sua contraparte não paramétrica que foi traçada anteriormente.

```
# Shape of `keys`: (`n_test`, `n_train`), where each column contains the same
# training inputs (i.e., same keys)
keys = x_train.repeat((n_test, 1))
# Shape of `value`: (`n_test`, `n_train`)
values = y_train.repeat((n_test, 1))
y_hat = net(x_test, keys, values).unsqueeze(1).detach()
plot_kernel_reg(y_hat)
```



Comparando com o *pooling* de atenção não paramétrico, a região com grandes pesos de atenção torna-se mais nítida na configuração programável e paramétrica.

```
d2l.show_heatmaps(net.attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')
```



10.2.5 Resumo

- A regressão do kernel Nadaraya-Watson é um exemplo de *machine learning* com mecanismos de atenção.
- O agrupamento de atenção da regressão do kernel Nadaraya-Watson é uma média ponderada dos resultados do treinamento. Do ponto de vista da atenção, o peso da atenção é atribuído a um valor com base em uma função de uma consulta e a chave que está emparelhada com o valor.
- O *pooling* de atenção pode ser não paramétrico ou paramétrico.

10.2.6 Exercícios

1. Aumente o número de exemplos de treinamento. Você pode aprender melhor a regressão de kernel não paramétrica Nadaraya-Watson?
2. Qual é o valor de nosso w aprendido no experimento paramétrico de concentração de atenção? Por que torna a região ponderada mais nítida ao visualizar os pesos de atenção?
3. Como podemos adicionar hiperparâmetros à regressão de kernel Nadaraya-Watson não paramétrica para prever melhor?
4. Projete outro agrupamento de atenção paramétrica para a regressão do kernel desta seção. Treine este novo modelo e visualize seus pesos de atenção.

Discussions[102]

[102] https://discuss.d2l.ai/t/1599

10.3 Funções de Pontuação de Atenção

Em Section 10.2, usamos um kernel gaussiano para modelar interações entre consultas e chaves. Tratando o expoente do kernel gaussiano em (10.2.6) como uma *função de pontuação de atenção* (ou *função de pontuação* para abreviar), os resultados desta função foram essencialmente alimentados em uma operação softmax. Como resultado, Nós obtivemos uma distribuição de probabilidade (pesos de atenção) sobre valores que estão emparelhados com chaves. No fim, a saída do *pooling* de atenção é simplesmente uma soma ponderada dos valores com base nesses pesos de atenção.

Em alto nível, podemos usar o algoritmo acima para instanciar a estrutura de mecanismos de atenção em Fig. 10.1.3. Denotando uma função de pontuação de atenção por a, Fig. 10.3.1 ilustra como a saída do *pooling* de atenção pode ser calculado como uma soma ponderada de valores. Uma vez que os pesos de atenção são uma distribuição de probabilidade, a soma ponderada é essencialmente uma média ponderada.

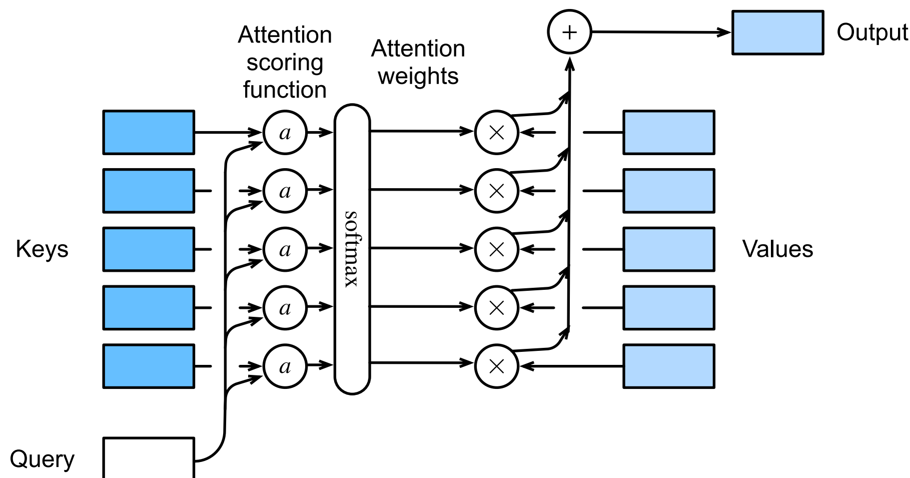


Fig. 10.3.1: Calculando a saída do *pooling* de atenção como uma média ponderada de valores.

Matematicamente, suponha que temos uma consulta $\mathbf{q} \in \mathbb{R}^q$ e m pares de valores-chave $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)$, onde qualquer $\mathbf{k}_i \in \mathbb{R}^k$ e qualquer $\mathbf{v}_i \in \mathbb{R}^v$. O *pooling* de atenção f é instanciado como uma soma ponderada dos valores:

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i \in \mathbb{R}^v, \tag{10.3.1}$$

Onde o peso da atenção (escalar) para a consulta $\mathbf{q}$ e a chave $\mathbf{k}_i$ é calculado pela operação softmax de uma função de pontuação de atenção a que mapeia dois vetores para um escalar:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^m \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \tag{10.3.2}$$

Como podemos ver, diferentes escolhas da função de pontuação de atenção a levam a diferentes comportamentos de concentração de atenção. Nesta secção, apresentamos duas funções populares de pontuação que usaremos para desenvolver mais mecanismos sofisticados de atenção posteriormente.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.3.1 Operação *Softmax* Mascarada

Como acabamos de mencionar, uma operação softmax é usada para produzir uma distribuição de probabilidade como pesos de atenção. Em alguns casos, nem todos os valores devem ser incluídos no agrupamento de atenção. Por exemplo, para processamento eficiente de minibatch em Section 9.5, algumas sequências de texto são preenchidas com tokens especiais que não possuem significado. Para obter um *pooling* de atenção sobre apenas tokens significativos como valores, podemos especificar um comprimento de sequência válido (em número de tokens) para filtrar aqueles que estão além deste intervalo especificado ao calcular softmax. Desta maneira, podemos implementar tal *operação de softmax mascarada* na seguinte função `masked_softmax`, onde qualquer valor além do comprimento válido é mascarado como zero.

```
#@save
def masked_softmax(X, valid_lens):
    """Perform softmax operation by masking elements on the last axis."""
    # `X`: 3D tensor, `valid_lens`: 1D or 2D tensor
    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = d2l.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
                              value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

Para demonstrar como essa função funciona, considere um minibatch de dois exemplos de matriz 2×4, onde os comprimentos válidos para esses dois exemplos são dois e três, respectivamente. Como resultado da operação mascarada softmax, valores além dos comprimentos válidos são todos mascarados como zero.

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([2, 3]))
```

```
tensor([[[0.3520, 0.6480, 0.0000, 0.0000],
         [0.5525, 0.4475, 0.0000, 0.0000]],

        [[0.2764, 0.4460, 0.2776, 0.0000],
         [0.3825, 0.3849, 0.2327, 0.0000]]])
```

Da mesma forma, também podemos use um tensor bidimensional para especificar comprimentos válidos para cada linha em cada exemplo de matriz.

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([[1, 3], [2, 4]]))
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000],
         [0.4436, 0.2773, 0.2791, 0.0000]],

        [[0.4437, 0.5563, 0.0000, 0.0000],
         [0.2422, 0.3533, 0.2061, 0.1984]]])
```

10.3.2 Atenção Aditiva

Em geral, quando as consultas e as chaves são vetores de comprimentos diferentes, podemos usar atenção aditiva como a função de pontuação. Dada uma consulta $\mathbf{q} \in \mathbb{R}^q$ e uma chave **:raw-latex:`\mathbf{k}` :raw-latex:`\in `:raw-latex:`**mathbb{R}`^k$, a função de pontuação *atenção aditiva*

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R}, \tag{10.3.3}$$

onde parâmetros aprendíveis $\mathbf{W}_q \in \mathbb{R}^{h\times q}$, $\mathbf{W}_k \in \mathbb{R}^{h\times k}$, e $\mathbf{w}_v \in \mathbb{R}^{h}$. Equivalente a (10.3.3), a consulta e a chave são concatenadas e alimentado em um MLP com uma única camada oculta cujo número de unidades ocultas é h, um hiperparâmetro. Usando tanh como a função de ativação e desativando termos de *bias*, implementamos atenção aditiva a seguir.

```
#@save
class AdditiveAttention(nn.Module):
    def __init__(self, key_size, query_size, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
        self.w_v = nn.Linear(num_hiddens, 1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of `queries`: (`batch_size`, no. of
        # queries, 1, `num_hiddens`) and shape of `keys`: (`batch_size`, 1,
        # no. of key-value pairs, `num_hiddens`). Sum them up with
        # broadcasting
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # There is only one output of `self.w_v`, so we remove the last
        # one-dimensional entry from the shape. Shape of `scores`:
        # (`batch_size`, no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of `values`: (`batch_size`, no. of key-value pairs, value
        # dimension)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

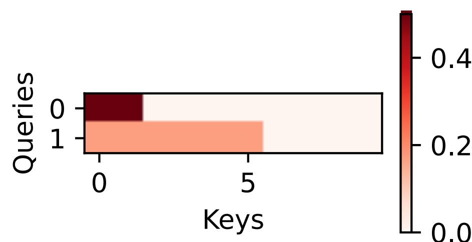
Vamos demonstrar a classe `AdditiveAttention` acima com um exemplo de brinquedo, onde formas (tamanho do lote, número de etapas ou comprimento da sequência em tokens, tamanho da *feature*) de consultas, chaves e valores são $(2, 1, 20)$, $(2, 10, 2)$, e $(2, 10, 4)$, respectivamente. A saída de concentração de atenção tem uma forma de (tamanho do lote, número de etapas para consultas, tamanho do *feature* para valores).

```
queries, keys = torch.normal(0, 1, (2, 1, 20)), torch.ones((2, 10, 2))
# The two value matrices in the `values` minibatch are identical
values = torch.arange(40, dtype=torch.float32).reshape(1, 10, 4).repeat(
    2, 1, 1)
valid_lens = torch.tensor([2, 6])

attention = AdditiveAttention(key_size=2, query_size=20, num_hiddens=8,
                              dropout=0.1)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000]],

        [[10.0000, 11.0000, 12.0000, 13.0000]]], grad_fn=<BmmBackward0>)
```

Embora a atenção aditiva contenha parâmetros que podem ser aprendidos, uma vez que cada chave é a mesma neste exemplo, os pesos de atenção são uniformes, determinados pelos comprimentos válidos especificados.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```



10.3.3 Atenção de Produto Escalar em Escala

Um design mais eficiente do ponto de vista computacional para a função de pontuação pode ser simplesmente o produto escalar. No entanto, a operação de produto escalar requer que a consulta e a chave tenham o mesmo comprimento de vetor, digamos d. Suponha que todos os elementos da consulta e a chave sejam variáveis aleatórias independentes com média zero e variância unitária. O produto escalar de ambos os vetores tem média zero e variância de d. Para garantir que a variação do produto escalar ainda permaneça um, independentemente do comprimento do vetor, a função de pontuação de *atenção ao produto escalar em escala*

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}/\sqrt{d} \tag{10.3.4}$$

divide o produto escalar por $\sqrt{d}$. Na prática, geralmente pensamos em minibatches para eficiência, como computação de atenção para n consultas e m pares de valor-chave, onde consultas e chaves têm comprimento d e os valores têm comprimento v. A atenção do produto escalar das consultas $\mathbf{Q} \in \mathbb{R}^{n\times d}$, chaves $\mathbf{K} \in \mathbb{R}^{m\times d}$, e valores $\mathbf{V} \in \mathbb{R}^{m\times v}$ é

$$\mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right) \mathbf{V} \in \mathbb{R}^{n\times v}. \tag{10.3.5}$$

Na implementação a seguir da atenção ao produto escalar, usamos o *dropout* para regularização do modelo.

```
#@save
class DotProductAttention(nn.Module):
    """Scaled dot product attention."""
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # Shape of `queries`: (`batch_size`, no. of queries, `d`)
    # Shape of `keys`: (`batch_size`, no. of key-value pairs, `d`)
    # Shape of `values`: (`batch_size`, no. of key-value pairs, value
    # dimension)
    # Shape of `valid_lens`: (`batch_size`,) or (`batch_size`, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Set `transpose_b=True` to swap the last two dimensions of `keys`
        scores = torch.bmm(queries, keys.transpose(1,2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

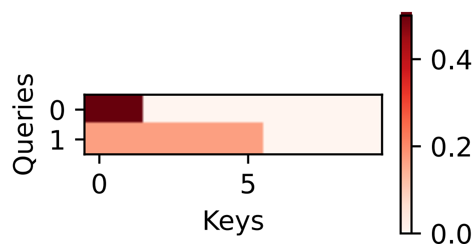
Para demonstrar a classe `DotProductAttention` acima, usamos as mesmas chaves, valores e comprimentos válidos do exemplo de brinquedo anterior para atenção aditiva. Para a operação de produto escalar, fazemos o tamanho da *feature* de consultas o mesmo que o das chaves.

```
queries = torch.normal(0, 1, (2, 1, 2))
attention = DotProductAttention(dropout=0.5)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000]],

        [[10.0000, 11.0000, 12.0000, 13.0000]]])
```

Da mesma forma que na demonstração de atenção aditiva, uma vez que `keys` contém o mesmo elemento que não pode ser diferenciado por nenhuma consulta, pesos uniformes de atenção são obtidos.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```



10.3.4 Resumo

- Podemos calcular a saída do *pooling* de atenção como uma média ponderada de valores, onde diferentes escolhas da função de pontuação de atenção levam a diferentes comportamentos de agrupamento de atenção.
- Quando consultas e chaves são vetores de comprimentos diferentes, podemos usar a função de pontuação de atenção aditiva. Quando são iguais, a função de pontuação de atenção do produto escalonado é mais eficiente do ponto de vista computacional.

10.3.5 Exercícios

1. Modifique as chaves no exemplo do brinquedo e visualize o peso da atenção. A atenção aditiva e a atenção de produto escalar em escala ainda geram os mesmos pesos de atenção? Por que ou por que não?
2. Usando apenas multiplicações de matrizes, você pode projetar uma nova função de pontuação para consultas e chaves com diferentes comprimentos de vetor?
3. Quando as consultas e as chaves têm o mesmo comprimento de vetor, a soma de vetores é um design melhor do que o produto escalar para a função de pontuação? Por que ou por que não?

Discussions[103]

10.4 Atenção de Bahdanau

Nós estudamos a tradução automática problema em Section 9.7, onde projetamos uma arquitetura de codificador-decodificador baseada em duas RNNs para aprendizagem de sequência em sequência. Especificamente, o codificador de RNN transforma uma sequência de comprimento variável em uma variável de contexto de forma fixa, então o decodificador de RNN gera o token de sequência de saída (destino) por token com base nos tokens gerados e na variável de contexto. Contudo, mesmo que nem todos os tokens de entrada (fonte) são úteis para decodificar um certo token, a *mesma* variável de contexto que codifica toda a sequência de entrada ainda é usada em cada etapa de decodificação.

Em um separado, mas relacionado desafio de geração de caligrafia para uma determinada sequência de texto, Graves projetou um modelo de atenção diferenciável para alinhar caracteres de texto com o traço de caneta muito mais longo, onde o alinhamento se move apenas em uma direção (Graves, 2013). Inspirado pela ideia de aprender a alinhar, Bahdanau et al. propôs um modelo de atenção diferenciável sem a limitação severa de alinhamento unidirecional (Bahdanau et al., 2014). Ao prever um token, se nem todos os tokens de entrada forem relevantes, o modelo alinha (ou atende) apenas para partes da sequência de entrada que são relevantes para a previsão atual. Isso é alcançado tratando a variável de contexto como uma saída do agrupamento de atenção.

[103] https://discuss.d2l.ai/t/1064

10.4.1 Modelo

Ao descrever Atenção Bahdanau para o codificador-decodificador RNN abaixo, nós seguiremos a mesma notação em Section 9.7. O novo modelo baseado na atenção é o mesmo que em Section 9.7 exceto que a variável de contexto $\mathbf{c}$ em (9.7.3) é substituída por $\mathbf{c}_{t'}$ em qualquer passo de tempo de decodificação t'. Suponha que existem tokens T na sequência de entrada, a variável de contexto na etapa de tempo de decodificação t' é o resultado do agrupamento de atenção:

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)\mathbf{h}_t, \tag{10.4.1}$$

onde o decodificador está escondido $\mathbf{s}_{t'-1}$ no passo de tempo $t'-1$ é a consulta, e os estados ocultos do codificador $\mathbf{h}_t$ são as chaves e os valores, e o peso de atenção α é calculado como em (10.3.2) usando a função de pontuação de atenção aditiva definida por (10.3.3).

Um pouco diferente da arquitetura do codificador-decodificador Vanilla RNN em Fig. 9.7.2, a mesma arquitetura com atenção de Bahdanau, é retratada em Fig. 10.4.1.

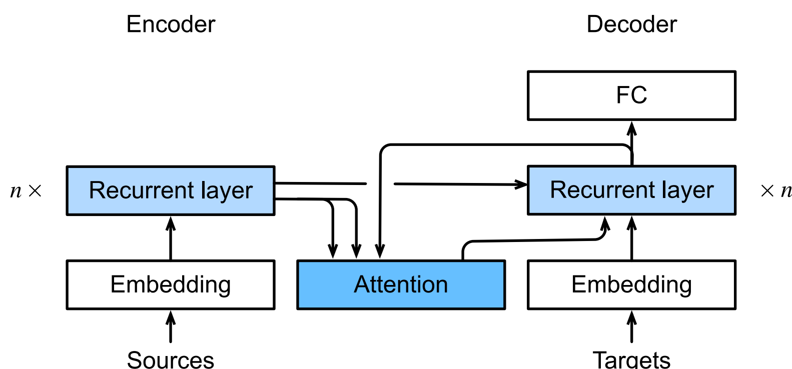


Fig. 10.4.1: Camadas em um modelo de codificador-decodificador RNN com atenção Bahdanau.

```
import torch
from torch import nn
from d2l import torch as d2l
```

10.4.2 Definindo o Decodificador com Atenção

Para implementar o codificador-decodificador RNN com atenção Bahdanau, só precisamos redefinir o decodificador. Para visualizar os pesos de atenção aprendidos de forma mais conveniente, a seguinte classe `AttentionDecoder` define a interface base para decodificadores com mecanismos de atenção.

```
#@save
class AttentionDecoder(d2l.Decoder):
    """The base attention-based decoder interface."""
    def __init__(self, **kwargs):
        super(AttentionDecoder, self).__init__(**kwargs)

    @property
```

(continues on next page)

(continued from previous page)

```
    def attention_weights(self):
        raise NotImplementedError
```

Agora vamos implementar o decodificador RNN com atenção Bahdanau na seguinte classe `Seq2SeqAttentionDecoder`. O estado do decodificador é inicializado com i) os estados ocultos da camada final do codificador em todas as etapas de tempo (como chaves e valores da atenção); ii) o estado oculto de todas as camadas do codificador na etapa de tempo final (para inicializar o estado oculto do decodificador); e iii) o comprimento válido do codificador (para excluir os tokens de preenchimento no agrupamento de atenção). Em cada etapa de tempo de decodificação, o estado oculto da camada final do decodificador na etapa de tempo anterior é usado como a consulta da atenção. Como resultado, tanto a saída de atenção e a incorporação de entrada são concatenadas como entrada do decodificador RNN.

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
        self.attention = d2l.AdditiveAttention(
            num_hiddens, num_hiddens, num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        # Shape of `outputs`: (`num_steps`, `batch_size`, `num_hiddens`).
        # Shape of `hidden_state[0]`: (`num_layers`, `batch_size`,
        # `num_hiddens`)
        outputs, hidden_state = enc_outputs
        return (outputs.permute(1, 0, 2), hidden_state, enc_valid_lens)

    def forward(self, X, state):
        # Shape of `enc_outputs`: (`batch_size`, `num_steps`, `num_hiddens`).
        # Shape of `hidden_state[0]`: (`num_layers`, `batch_size`,
        # `num_hiddens`)
        enc_outputs, hidden_state, enc_valid_lens = state
        # Shape of the output `X`: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:
            # Shape of `query`: (`batch_size`, 1, `num_hiddens`)
            query = torch.unsqueeze(hidden_state[-1], dim=1)
            # Shape of `context`: (`batch_size`, 1, `num_hiddens`)
            context = self.attention(
                query, enc_outputs, enc_outputs, enc_valid_lens)
            # Concatenate on the feature dimension
            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
            # Reshape `x` as (1, `batch_size`, `embed_size` + `num_hiddens`)
            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)
        # After fully-connected layer transformation, shape of `outputs`:
```

(continues on next page)

(continued from previous page)

```
        # (`num_steps`, `batch_size`, `vocab_size`)
        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
                                          enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

A seguir, testamos o decodificador implementado com atenção Bahdanau usando um minibatch de 4 entradas de sequência de 7 etapas de tempo.

```
encoder = d2l.Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                             num_layers=2)
encoder.eval()
decoder = Seq2SeqAttentionDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                                  num_layers=2)
decoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)  # (`batch_size`, `num_steps`)
state = decoder.init_state(encoder(X), None)
output, state = decoder(X, state)
output.shape, len(state), state[0].shape, len(state[1]), state[1][0].shape
```

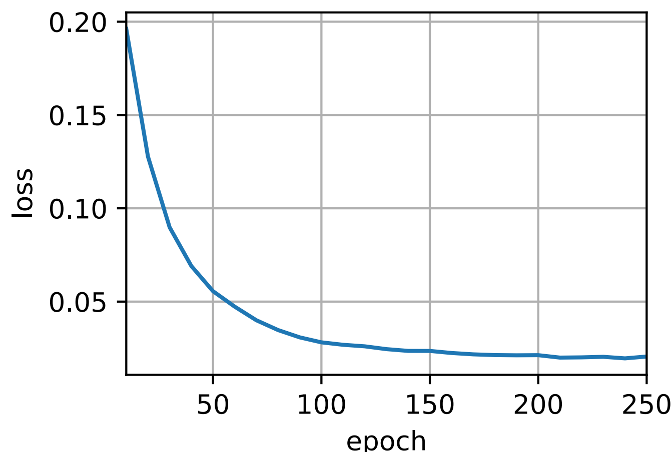
```
(torch.Size([4, 7, 10]), 3, torch.Size([4, 7, 16]), 2, torch.Size([4, 16]))
```

10.4.3 Treinamento

Semelhante a Section 9.7.4, aqui especificamos hiperparâmetros, instanciamos um codificador e um decodificador com atenção Bahdanau, e treinamos este modelo para tradução automática. Devido ao mecanismo de atenção recém-adicionado, este treinamento é muito mais lento do que que em Section 9.7.4 sem mecanismos de atenção.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 250, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = d2l.Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqAttentionDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.021, 4700.2 tokens/sec on cuda:0
```



Depois que o modelo é treinado, nós o usamos para traduzir algumas frases do inglês para o francês e computar suas pontuações BLEU.

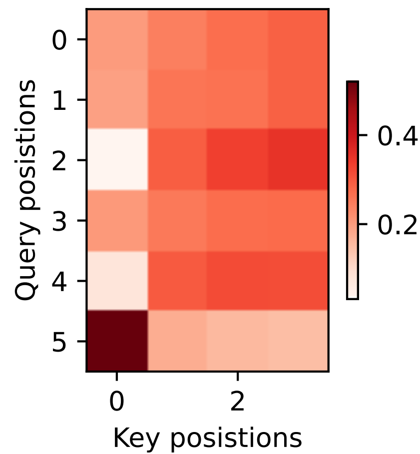
```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 1.000
he's calm . => il est mouillé .,  bleu 0.658
i'm home . => je suis chez moi .,  bleu 1.000
```

```
attention_weights = torch.cat([step[0][0][0] for step in dec_attention_weight_seq], 0).
↪reshape((
    1, 1, -1, num_steps))
```

Visualizando os pesos de atenção ao traduzir a última frase em inglês, podemos ver que cada consulta atribui pesos não uniformes sobre pares de valores-chave. Isso mostra que em cada etapa de decodificação, diferentes partes das sequências de entrada são agregadas seletivamente no pool de atenção.

```
# Plus one to include the end-of-sequence token
d2l.show_heatmaps(
    attention_weights[:, :, :, :len(engs[-1].split()) + 1].cpu(),
    xlabel='Key posistions', ylabel='Query posistions')
```



10.4.4 Resumo

- Ao prever um token, se nem todos os tokens de entrada forem relevantes, o codificador-decodificador RNN com atenção Bahdanau seletivamente agrega diferentes partes da sequência de entrada. Isso é obtido tratando a variável de contexto como uma saída do agrupamento de atenção aditiva.
- No codificador-decodificador RNN, a atenção Bahdanau trata o estado oculto do decodificador na etapa de tempo anterior como a consulta, e os estados ocultos do codificador em todas as etapas de tempo como as chaves e os valores.

10.4.5 Exercícios

1. Substitua GRU por LSTM no experimento.
2. Modifique o experimento para substituir a função de pontuação de atenção aditiva pelo produto escalar escalonado. Como isso influencia a eficiência do treinamento?

Discussions[104]

10.5 Atenção Multi-Head

Na prática, dado o mesmo conjunto de consultas, chaves e valores podemos querer que nosso modelo combine conhecimento de diferentes comportamentos do mesmo mecanismo de atenção, como capturar dependências de vários intervalos (por exemplo, intervalo mais curto vs. intervalo mais longo) dentro de uma sequência. Desse modo, pode ser benéfico permitir nosso mecanismo de atenção para usar em conjunto diferentes subespaços de representação de consultas, chaves e valores.

Para este fim, em vez de realizar um único agrupamento de atenção, consultas, chaves e valores podem ser transformados com h projeções lineares aprendidas independentemente. Então, essas h consultas, chaves e valores projetados são alimentados em agrupamento de atenção em paralelo. No fim, h resultados de concentração de atenção são concatenados e transformados com outra

[104] https://discuss.d2l.ai/t/1065

projeção linear aprendida para produzir a saída final. Este design é chamado de *atenção multi-head*, onde cada uma das saídas de concentração de h é um *head* (Vaswani et al., 2017). Usando camadas totalmente conectadas para realizar transformações lineares que podem ser aprendidas, Fig. 10.5.1 descreve a atenção de *multi-head*.

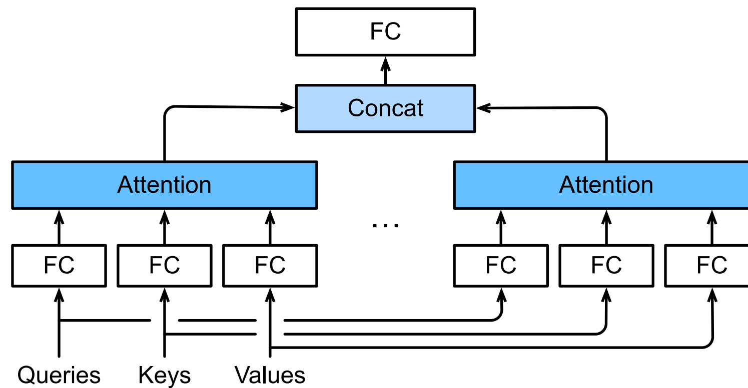


Fig. 10.5.1: Multi-head attention, where multiple heads are concatenated then linearly transformed.

10.5.1 Modelo

Antes de fornecer a implementação da atenção *multi-head*, vamos formalizar este modelo matematicamente. Dada uma consulta $\mathbf{q} \in \mathbb{R}^{d_q}$, uma chave $\mathbf{k} \in \mathbb{R}^{d_k}$, e um valor $\mathbf{v} \in \mathbb{R}^{d_v}$, cada *head* de atenção $\mathbf{h}_i$ ($i = 1, \ldots, h$) é calculado como

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}, \tag{10.5.1}$$

onde parâmetros aprendíveis $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ e $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$, e f é concentração de atenção, tal como atenção aditiva e atenção de produto escalonado em Section 10.3. A saída de atenção *multi-head* é outra transformação linear via parâmetros aprendíveis $\mathbf{W}_o \in \mathbb{R}^{p_o \times h p_v}$ da concatenação de h cabeças:

$$\mathbf{W}_o \begin{bmatrix}\mathbf{h}_1\\\vdots\\\mathbf{h}_h\end{bmatrix} \in \mathbb{R}^{p_o}. \tag{10.5.2}$$

Com base neste design, cada cabeça pode atender a diferentes partes da entrada. Funções mais sofisticadas do que a média ponderada simples podem ser expressadas.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.5.2 Implementação

Em nossa implementação, nós escolhemos a atenção do produto escalonado para cada *head* da atenção de várias cabeças. Para evitar um crescimento significativo de custo computacional e custo de parametrização, montamos $p_q = p_k = p_v = p_o/h$. Observe que h *heads* pode ser calculado em paralelo se definirmos o número de saídas de transformações lineares para a consulta, chave e valor a $p_q h = p_k h = p_v h = p_o$. Na implementação a seguir, p_o é especificado através do argumento `num_hiddens`.

```
#@save
class MultiHeadAttention(nn.Module):
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # Shape of `queries`, `keys`, or `values`:
        # (`batch_size`, no. of queries or key-value pairs, `num_hiddens`)
        # Shape of `valid_lens`:
        # (`batch_size`,) or (`batch_size`, no. of queries)
        # After transposing, shape of output `queries`, `keys`, or `values`:
        # (`batch_size` * `num_heads`, no. of queries or key-value pairs,
        # `num_hiddens` / `num_heads`)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)

        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for
            # `num_heads` times, then copy the next item, and so on
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # Shape of `output`: (`batch_size` * `num_heads`, no. of queries,
        # `num_hiddens` / `num_heads`)
        output = self.attention(queries, keys, values, valid_lens)

        # Shape of `output_concat`:
        # (`batch_size`, no. of queries, `num_hiddens`)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

Para permitir o cálculo paralelo de várias *heads* a classe `MultiHeadAttention` acima usa duas funções de transposição, conforme definido abaixo. Especificamente, a função `transpose_output` reverte a operação da função `transpose_qkv`.

```
#@save
def transpose_qkv(X, num_heads):
    # Shape of input `X`:
```

(continues on next page)

(continued from previous page)

```
    # (`batch_size`, no. of queries or key-value pairs, `num_hiddens`).
    # Shape of output `X`:
    # (`batch_size`, no. of queries or key-value pairs, `num_heads`,
    # `num_hiddens` / `num_heads`)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # Shape of output `X`:
    # (`batch_size`, `num_heads`, no. of queries or key-value pairs,
    # `num_hiddens` / `num_heads`)
    X = X.permute(0, 2, 1, 3)

    # Shape of `output`:
    # (`batch_size` * `num_heads`, no. of queries or key-value pairs,
    # `num_hiddens` / `num_heads`)
    return X.reshape(-1, X.shape[2], X.shape[3])


#@save
def transpose_output(X, num_heads):
    """Reverse the operation of `transpose_qkv`"""
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

Vamos testar nossa classe `MultiHeadAttention` implementada usando um exemplo de brinquedo em que as chaves e os valores são iguais. Como resultado, a forma da saída de atenção *multi-head* é (batch_size,num_queries, num_hiddens).

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                               num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
  (attention): DotProductAttention(
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (W_q): Linear(in_features=100, out_features=100, bias=False)
  (W_k): Linear(in_features=100, out_features=100, bias=False)
  (W_v): Linear(in_features=100, out_features=100, bias=False)
  (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries, num_kvpairs, valid_lens = 2, 4, 6, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kvpairs, num_hiddens))
attention(X, Y, Y, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

10.5.3 Resumo

- A atenção *multi-head* combina o conhecimento do mesmo agrupamento de atenção por meio de diferentes subespaços de representação de consultas, chaves e valores.
- Para calcular várias *heads* de atenção de *multi-heads* em paralelo, é necessária a manipulação adequada do tensor.

10.5.4 Exercícios

1. Visualize o peso da atenção *multi-head* neste experimento.
2. Suponha que temos um modelo treinado com base na atenção *multi-head* e queremos podar as *heads* menos importantes para aumentar a velocidade de previsão. Como podemos projetar experimentos para medir a importância de uma *head* de atenção?

Discussions[105]

10.6 Autoatenção e Codificação Posicional

No aprendizado profundo, costumamos usar CNNs ou RNNs para codificar uma sequência. Agora, com os mecanismos de atenção, imagine que alimentamos uma sequência de tokens no *pooling* de atenção para que o mesmo conjunto de tokens atue como consultas, chaves e valores. Especificamente, cada consulta atende a todos os pares de valores-chave e gera uma saída de atenção. Como as consultas, chaves e valores vêm do mesmo lugar, isso executa *autoatenção* (Lin et al., 2017b; Vaswani et al., 2017), que também é chamado *intra-atenção* (Cheng et al., 2016; Parikh et al., 2016; Paulus et al., 2017). Nesta seção, discutiremos a codificação de sequência usando autoatenção, incluindo o uso de informações adicionais para a ordem da sequência.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.6.1 Autoatenção

Dada uma sequência de tokens de entrada $\mathbf{x}_1, \ldots, \mathbf{x}_n$ onde qualquer $\mathbf{x}_i \in \mathbb{R}^d$ $(1 \leq i \leq n)$, suas saídas de autoatenção é uma sequência do mesmo comprimento $\mathbf{y}_1, \ldots, \mathbf{y}_n$, Onde

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \tag{10.6.1}$$

de acordo com a definição de concentração de f em (10.2.4). Usando a atenção de várias cabeças, o seguinte trecho de código calcula a autoatenção de um tensor com forma (tamanho do lote, número de etapas de tempo ou comprimento da sequência em tokens, d). O tensor de saída tem o mesmo formato.

[105] https://discuss.d2l.ai/t/1635

```python
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                                   num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
  (attention): DotProductAttention(
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (W_q): Linear(in_features=100, out_features=100, bias=False)
  (W_k): Linear(in_features=100, out_features=100, bias=False)
  (W_v): Linear(in_features=100, out_features=100, bias=False)
  (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```python
batch_size, num_queries, valid_lens = 2, 4, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
attention(X, X, X, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

10.6.2 Comparando CNNs, RNNs e Autoatenção

Vamos comparar arquiteturas para mapear uma sequência de n tokens para outra sequência de igual comprimento, onde cada token de entrada ou saída é representado por um vetor d-dimensional. Especificamente, consideraremos CNNs, RNNs e autoatenção. Compararemos sua complexidade computacional, operações sequenciais e comprimentos máximos de caminho. Observe que as operações sequenciais evitam a computação paralela, enquanto um caminho mais curto entre qualquer combinação de posições de sequência torna mais fácil aprender dependências de longo alcance dentro da sequência (Hochreiter et al., 2001).

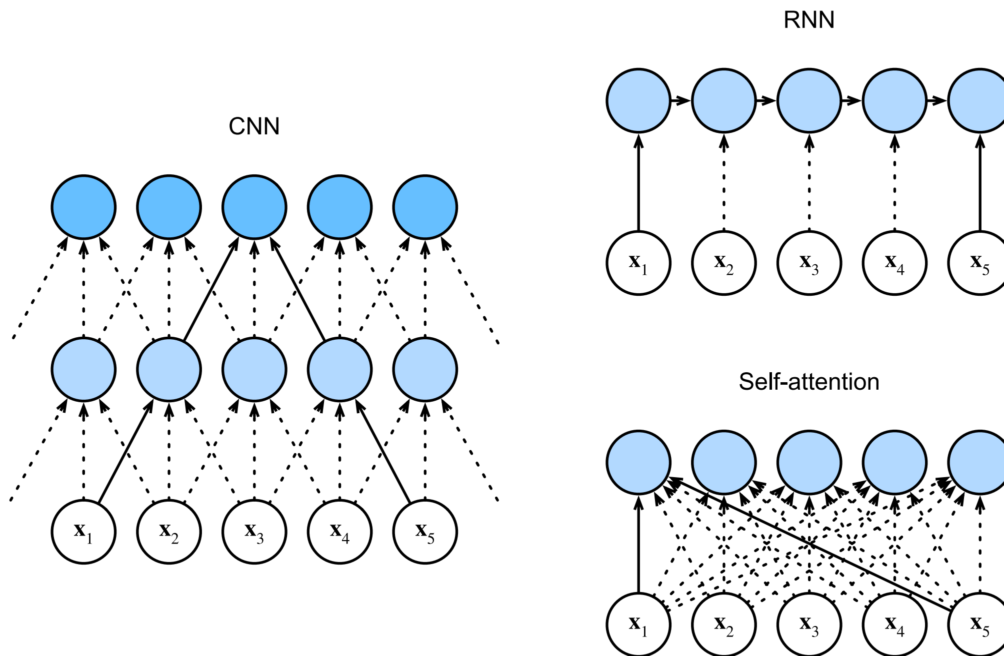


Fig. 10.6.1: Comparando CNN (tokens de preenchimento são omitidos), RNN e arquiteturas de autoatenção.

Considere uma camada convolucional cujo tamanho do kernel é k. Forneceremos mais detalhes sobre o processamento de sequência usando CNNs em capítulos posteriores. Por enquanto, só precisamos saber que, como o comprimento da sequência é n, os números de canais de entrada e saída são d, a complexidade computacional da camada convolucional é $\mathcal{O}(knd^2)$. Como mostra Fig. 10.6.1, CNNs são hierárquicas, então existem $\mathcal{O}(1)$ operações sequenciais e o comprimento máximo do caminho é $\mathcal{O}(n/k)$. Por exemplo, $\mathbf{x}_1$ e $:raw-latex:*mathbf{x}*_5 estão dentro do campo receptivo de um CNN de duas camadas com tamanho de kernel 3 em Fig. 10.6.1.

Ao atualizar o estado oculto de RNNs, multiplicação da matriz de pesos $d \times d$ e o estado oculto d-dimensional tem uma complexidade computacional de $\mathcal{O}(d^2)$. Uma vez que o comprimento da sequência é n, a complexidade computacional da camada recorrente é $\mathcal{O}(nd^2)$. De acordo com Fig. 10.6.1, existem $\mathcal{O}(n)$ operações sequenciais que não pode ser paralelizadas e o comprimento máximo do caminho também é $\mathcal{O}(n)$.

Na autoatenção, as consultas, chaves e valores são todas matrizes $n \times d$ Considere a atenção do produto escalonado em (10.3.5), onde uma matriz $n \times d$ é multiplicada por uma matriz $d \times n$, então a matriz de saída $n \times n$ é multiplicada por uma matriz $n \times d$. Como resultado, a autoatenção tem uma complexidade computacional $\mathcal{O}(n^2d)$ Como podemos ver em Fig. 10.6.1, cada token está diretamente conectado a qualquer outro token via auto-atenção. Portanto, a computação pode ser paralela com $\mathcal{O}(1)$ operações sequenciais e o comprimento máximo do caminho também é $\mathcal{O}(1)$.

Contudo, tanto CNNs quanto autoatenção desfrutam de computação paralela e a autoatenção tem o menor comprimento de caminho máximo. No entanto, a complexidade computacional quadrática em relação ao comprimento da sequência torna a auto-atenção proibitivamente lenta em sequências muito longas.

10.6.3 Codificação Posicional

Ao contrário dos RNNs que processam recorrentemente tokens de uma sequência, um por um, a autoatenção desvia as operações sequenciais em favor de computação paralela. Para usar as informações de ordem de sequência, podemos injetar informações posicionais absolutas ou relativas adicionando *codificação posicional* às representações de entrada. Codificações posicionais podem ser aprendidas ou corrigidas. A seguir, descrevemos uma codificação posicional fixa baseada nas funções seno e cosseno (Vaswani et al., 2017).

Suponha que a representação de entrada $\mathbf{X} \in \mathbb{R}^{n \times d}$ contém as características d-dimensionais embutidas para n tokens de uma sequência. A codificação posicional gera $\mathbf{X} + \mathbf{P}$ usando uma matriz de *embedding* posicional $\mathbf{P} \in \mathbb{R}^{n \times d}$ da mesma forma, cujo elemento na linha i^th row and the $(2j)^\mathrm{th}$ ou a coluna $(2j+1)^\mathrm{th}$ é

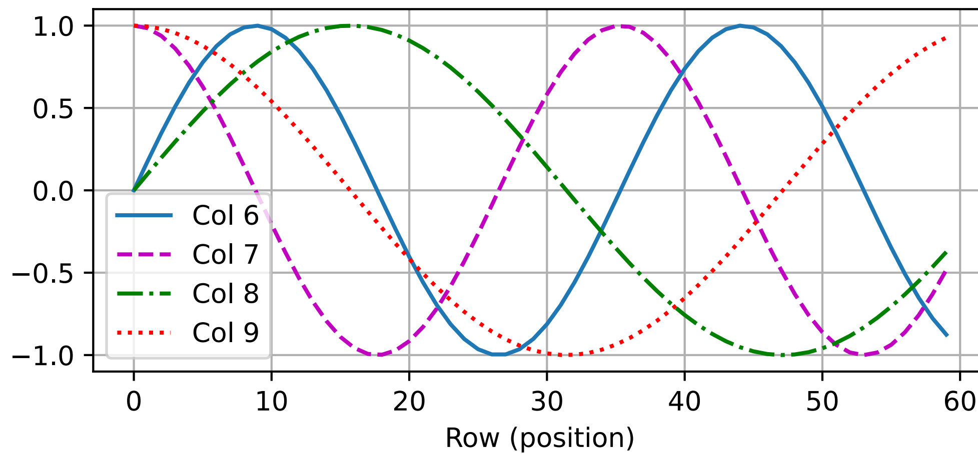
$$\begin{aligned} p_{i, 2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right),\\p_{i, 2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right).\end{aligned} \tag{10.6.2}$$

À primeira vista, esse design de função trigonométrica parece estranho. Antes das explicações deste design, vamos primeiro implementá-lo na seguinte classe `PositionalEncoding`.

```
#@save
class PositionalEncoding(nn.Module):
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        # Create a long enough `P`
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000, torch.arange(
            0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)
```

Na matriz de incorporação posicional $\mathbf{P}$, as linhas correspondem às posições dentro de uma sequência e as colunas representam diferentes dimensões de codificação posicional. No exemplo abaixo, podemos ver que as colunas 6th e 7th da matriz de embedding posicional têm uma frequência maior do que 8th e as colunas 9th. O deslocamento entre o 6th e o 7th (o mesmo para as colunas 8th e 9th) é devido à alternância das funções seno e cosseno.

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(torch.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])
```



Informação Posicional Absoluta

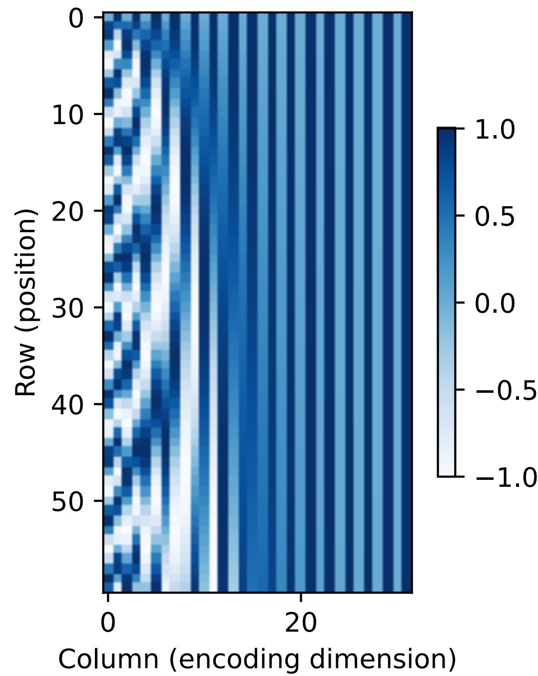
Para ver como a frequência monotonicamente diminuída ao longo da dimensão de codificação se relaciona com a informação posicional absoluta, vamos imprimir as representações binárias de $0, 1, \ldots, 7$. Como podemos ver, o bit mais baixo, o segundo bit mais baixo e o terceiro bit mais baixo se alternam em cada número, a cada dois números e a cada quatro números, respectivamente.

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

Em representações binárias, um bit mais alto tem uma frequência mais baixa do que um bit mais baixo. Da mesma forma, conforme demonstrado no mapa de calor abaixo, a codificação posicional diminui as frequências ao longo da dimensão de codificação usando funções trigonométricas. Uma vez que as saídas são números flutuantes, tais as representações são mais eficientes em termos de espaço do que as representações binárias.

```
P = P[0, :, :].unsqueeze(0).unsqueeze(0)
d2l.show_heatmaps(P, xlabel='Column (encoding dimension)',
                  ylabel='Row (position)', figsize=(3.5, 4), cmap='Blues')
```



Informação Posicional Relativa

Além de capturar informações posicionais absolutas, a codificação posicional acima também permite que um modelo aprenda facilmente a atender por posições relativas. Isso ocorre porque para qualquer deslocamento de posição fixa δ, a codificação posicional na posição $i + \delta$ pode ser representada por uma projeção linear daquela na posição i.

Essa projeção pode ser explicada matematicamente. Denotando $\omega_j = 1/10000^{2j/d}$, qualquer par de $(p_{i,2j}, p_{i,2j+1})$ em (10.6.2) pode ser linearmente projetado para $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$ para qualquer deslocamento fixo δ:

$$\begin{aligned}
&\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} \\
=&\begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\
=&\begin{bmatrix} \sin\left((i+\delta)\omega_j\right) \\ \cos\left((i+\delta)\omega_j\right) \end{bmatrix} \\
=&\begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},
\end{aligned} \tag{10.6.3}$$

onde a matriz de projeção 2×2 não depende de nenhum índice de posição i.

10.6.4 Resumo

- Na atenção própria, as consultas, chaves e valores vêm todos do mesmo lugar.
- Tanto as CNNs quanto a autoatenção desfrutam de computação paralela e a autoatenção tem o menor comprimento de caminho máximo. No entanto, a complexidade computacional quadrática em relação ao comprimento da sequência torna a autoatenção proibitivamente lenta para sequências muito longas.
- Para usar as informações de ordem de sequência, podemos injetar informações posicionais absolutas ou relativas adicionando codificação posicional às representações de entrada.

10.6.5 Exercícios

1. Suponha que projetemos uma arquitetura profunda para representar uma sequência, empilhando camadas de autoatenção com codificação posicional. Quais podem ser os problemas?
2. Você pode projetar um método de codificação posicional que possa ser aprendido?

Discussions[106]

10.7 Transformador

Comparamos CNNs, RNNs e autoatenção em Section 10.6.2. Notavelmente, a auto-atenção desfruta de computação paralela e do comprimento máximo de caminho mais curto. Portanto, naturalmente, é atraente projetar profundamente arquiteturas usando auto-atenção. Ao contrário dos modelos anteriores de autoatenção que ainda contam com RNNs para representações de entrada cite:`Cheng.Dong.Lapata.2016,Lin.Feng.Santos.ea.2017,Paulus.Xiong.Socher.2017`, o modelo do transformador é exclusivamente baseado em mecanismos de atenção sem qualquer camada convolucional ou recorrente (Vaswani et al., 2017). Embora originalmente propostos para aprendizagem de sequência para sequência em dados de texto, os transformadores têm sido difundidos em uma ampla gama de aplicações modernas de aprendizagem profunda, como nas áreas de linguagem, visão, fala e aprendizagem por reforço.

10.7.1 Modelo

Como uma instância da arquitetura codificador-decodificador, a arquitetura geral do transformador é apresentada em Fig. 10.7.1. Como podemos ver, o transformador é composto por um codificador e um decodificador. Diferente de Atenção Bahdanau para o aprendizado de sequência para sequência em Fig. 10.4.1, os *embeddings* de sequência de entrada (origem) e saída (destino) são adicionados com codificação posicional antes de serem alimentados no codificador e no decodificador que empilham módulos baseados em autoatenção.

[106] https://discuss.d2l.ai/t/1652

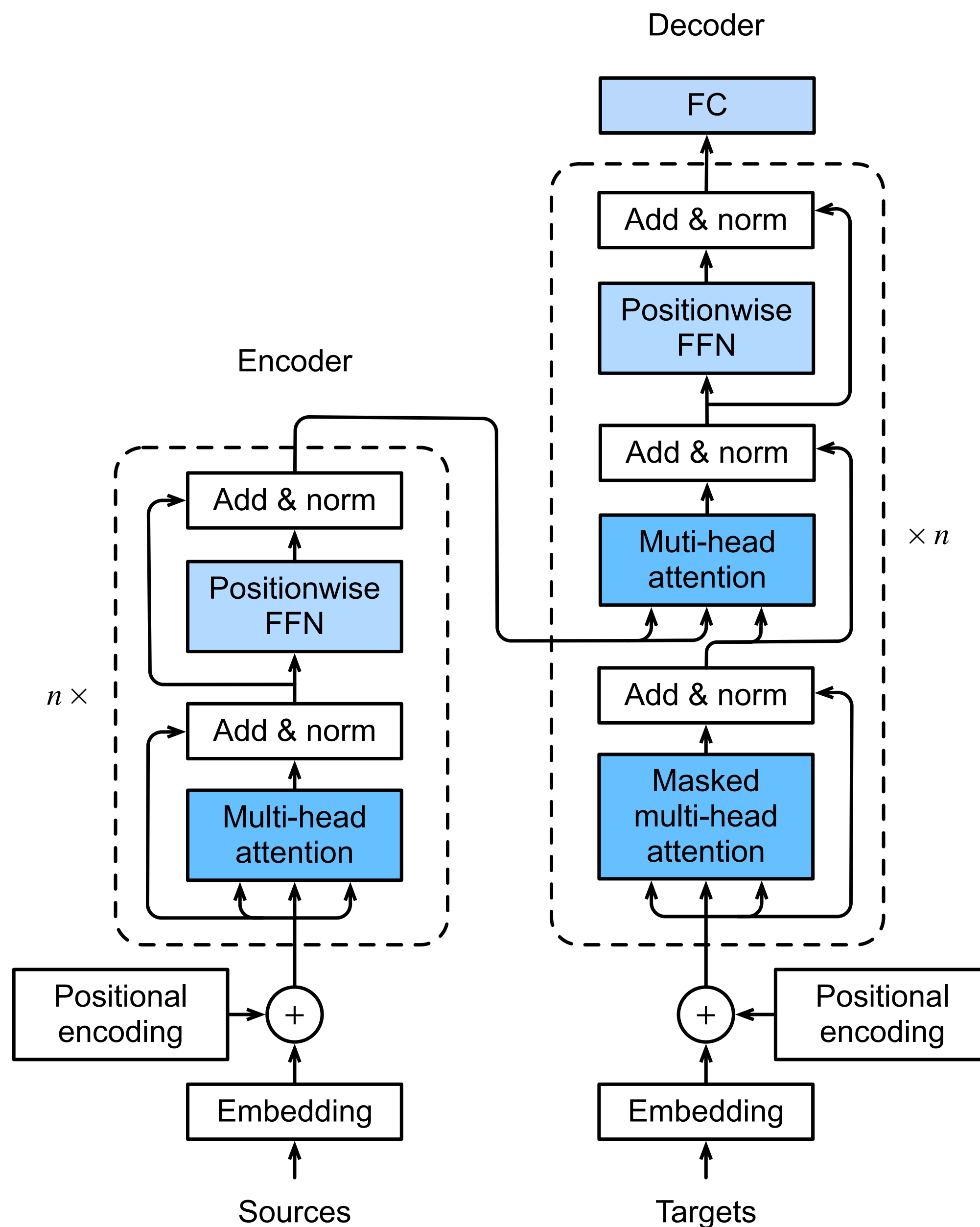


Fig. 10.7.1: A arquitetura do transformador.

Agora fornecemos uma visão geral da arquitetura do transformador em Fig. 10.7.1. Em um alto nível, o codificador do transformador é uma pilha de várias camadas idênticas, onde cada camada tem duas subcamadas (qualquer uma é denotada como sublayer). O primeiro é um pooling de autoatenção com várias *heads* e o segundo é uma rede *feed-forward* posicional. Especificamente, na autoatenção do codificador, as consultas, as chaves e os valores são todos provenientes das saídas da camada do codificador anterior. Inspirado no design ResNet em Section 7.6, uma conexão residual é empregada em torno de ambas as subcamadas. No transformador, para qualquer entrada $\mathbf{x} \in \mathbb{R}^d$ em qualquer posição da sequência, exigimos que $\mathrm{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ para que a conexão residual $\mathbf{x} + \mathrm{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ seja viável. Esta adição da conexão residual é imediatamente seguida pela normalização da camada (Ba et al., 2016). Como resultado, o codificador do transformador produz uma representação vetorial d-dimensional para cada posição da sequência de entrada.

O decodificador do transformador também é uma pilha de várias camadas idênticas com conexões residuais e normalizações de camada. Além das duas subcamadas descritas no codificador, o decodificador insere uma terceira subcamada, conhecida como atenção do codificador-decodificador, entre esses dois. Na atenção do codificador-decodificador, as consultas são das saídas da camada do decodificador anterior e as chaves e valores são das saídas do codificador do transformador. Na autoatenção do decodificador, consultas, chaves e valores são todos provenientes das saídas da camada do decodificador anterior. No entanto, cada posição no decodificador só pode atender a todas as posições no decodificador até aquela posição. Essa atenção *mascarada* preserva a propriedade auto-regressiva, garantindo que a previsão dependa apenas dos tokens de saída que foram gerados.

Já descrevemos e implementamos a atenção multi-head com base em produtos escalonados em Section 10.5 e codificação posicional em Section 10.6.3. A seguir, implementaremos o restante do modelo do transformador.

```
import math
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

10.7.2 Redes *Positionwise Feed-Forward*

A rede feed-forward *positionwise* transforma a representação em todas as posições de sequência usando o mesmo MLP. É por isso que o chamamos de *positionwise*. Na implementação abaixo, o `X` de entrada com forma (tamanho do lote, número de etapas de tempo ou comprimento da sequência em tokens, número de unidades ocultas ou dimensão do recurso) será transformado por um MLP de duas camadas em um tensor de saída de forma (tamanho do lote, número de passos de tempo, `ffn_num_outputs`).

```
#@save
class PositionWiseFFN(nn.Module):
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs,
                 **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

O seguinte exemplo mostra que a dimensão mais interna de um tensor muda para o número de saídas em a rede *feed-forward* posicionada. Uma vez que o mesmo MLP se transforma em todas as posições, quando as entradas em todas essas posições são as mesmas, suas saídas também são idênticas.

```
ffn = PositionWiseFFN(4, 4, 8)
ffn.eval()
ffn(torch.ones((2, 3, 4)))[0]
```

```
tensor([[-0.0037, -0.1095, -0.1937, -0.4333, -0.3985,  0.6414,  0.2048,  1.1088],
        [-0.0037, -0.1095, -0.1937, -0.4333, -0.3985,  0.6414,  0.2048,  1.1088],
        [-0.0037, -0.1095, -0.1937, -0.4333, -0.3985,  0.6414,  0.2048,  1.1088]],
       grad_fn=<SelectBackward>)
```

10.7.3 Conexão residual e normalização de camada

Agora vamos nos concentrar no componente "add & norm" em Fig. 10.7.1 Como descrevemos no início desta seção, esta é uma conexão residual imediatamente seguido pela normalização da camada. Ambos são essenciais para arquiteturas profundas eficazes.

Em Section 7.5, explicamos como a normalização em lote recentraliza e redimensiona os exemplos dentro um minibatch. A normalização de camada é igual à normalização em lote, exceto que a primeira normaliza em toda a dimensão do recurso. Apesar de suas aplicações difundidas em visão computacional, a normalização em lote é geralmente empiricamente menos eficaz do que a normalização de camada em tarefas de processamento de linguagem natural, cujas entradas são frequentemente sequências de comprimento variável.

O fragmento de código a seguir compara a normalização em diferentes dimensões por normalização de camada e normalização de lote.

```
ln = nn.LayerNorm(2)
bn = nn.BatchNorm1d(2)
X = torch.tensor([[1, 2], [2, 3]], dtype=torch.float32)
# Compute mean and variance from `X` in the training mode
print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: tensor([[-1.0000,  1.0000],
        [-1.0000,  1.0000]], grad_fn=<NativeLayerNormBackward>)
batch norm: tensor([[-1.0000, -1.0000],
        [ 1.0000,  1.0000]], grad_fn=<NativeBatchNormBackward>)
```

Agora podemos implementar a classe `AddNorm` usando uma conexão residual seguida pela normalização da camada. O *dropout* também é aplicado para regularização.

```
#@save
class AddNorm(nn.Module):
    def __init__(self, normalized_shape, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

A conexão residual requer que as duas entradas sejam da mesma forma de modo que o tensor de saída também tenha a mesma forma após a operação de adição.

```
add_norm = AddNorm([3, 4], 0.5) # Normalized_shape is input.size()[1:]
add_norm.eval()
add_norm(torch.ones((2, 3, 4)), torch.ones((2, 3, 4))).shape
```

```
torch.Size([2, 3, 4])
```

10.7.4 *Encoder*

Com todos os componentes essenciais para montar o *encoder* do transformador, vamos começar implementando uma única camada dentro do *encoder*. A seguinte classe `EncoderBlock` contém duas subcamadas: autoatenção com várias *heads* e redes de alimentação em posição posicionada, onde uma conexão residual seguida pela normalização da camada é empregada em torno de ambas as subcamadas.

```
#@save
class EncoderBlock(nn.Module):
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout,
            use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(
            ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

Como podemos ver, qualquer camada no *encoder* do transformador não altera a forma de sua entrada.

```
X = torch.ones((2, 100, 24))
valid_lens = torch.tensor([3, 2])
encoder_blk = EncoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5)
encoder_blk.eval()
encoder_blk(X, valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

Na seguinte implementação do *encoder* de transformador, empilhamos instâncias `num_layers` das classes`EncoderBlock` acima. Uma vez que usamos a codificação posicional fixa cujos valores estão sempre entre -1 e 1, nós multiplicamos os valores dos *embeddings* de entrada aprendíveis pela raiz quadrada da dimensão de incorporação para redimensionar antes de resumir a incorporação de entrada e a codificação posicional.

```
#@save
class TransformerEncoder(d2l.Encoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, use_bias))

    def forward(self, X, valid_lens, *args):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

Abaixo, especificamos hiperparâmetros para criar um *encoder* de transformador de duas camadas. A forma da saída do *encoder* do transformador é (tamanho do lote, número de etapas de tempo, `num_hiddens`).

```
encoder = TransformerEncoder(
    200, 24, 24, 24, 24, [100, 24], 24, 48, 8, 2, 0.5)
encoder.eval()
encoder(torch.ones((2, 100), dtype=torch.long), valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

10.7.5 *Decoder*

Conforme mostrado em Fig. 10.7.1, o *decoder* do transformador é composto de várias camadas idênticas. Cada camada é implementada na seguinte classe `DecoderBlock`, que contém três subcamadas: autoatenção do *decoder*, atenção do *encoder-decoder* e redes *feed-forward* posicionais. Essas subcamadas empregam uma conexão residual em torno delas seguida pela normalização da camada.

Como descrevemos anteriormente nesta seção, na autoatenção do *decoder* de várias *heads* mascarada (a primeira subcamada), as consultas, as chaves e os valores vêm todos das saídas da camada do *decoder* anterior. Ao treinar modelos de sequência para sequência, os tokens em todas as posições (etapas de tempo) da sequência de saída são conhecidos. No entanto, durante a predição, a sequência de saída é gerada token por token; assim, em qualquer etapa de tempo do *decoder*, apenas os tokens gerados podem ser usados na autoatenção do *decoder*. Para preservar a autoregressão no *decoder*, sua autoatenção mascarada especifica `dec_valid_lens` para que qualquer consulta atenda apenas a todas as posições no *decoder* até a posição de consulta.

```python
class DecoderBlock(nn.Module):
    # The `i`-th block in the decoder
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
                                   num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
        # at the same time, so `state[2][self.i]` is `None` as initialized.
        # When decoding any output sequence token by token during prediction,
        # `state[2][self.i]` contains representations of the decoded output at
        # the `i`-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # Shape of `dec_valid_lens`: (`batch_size`, `num_steps`), where
            # every row is [1, 2, ..., `num_steps`]
            dec_valid_lens = torch.arange(
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None

        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Encoder-decoder attention. Shape of `enc_outputs`:
        # (`batch_size`, `num_steps`, `num_hiddens`)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

Para facilitar as operações de produtos escalonados na atenção do *encoder-decoder* e operações de adição nas conexões residuais, a dimensão do recurso (`num_hiddens`) do *decoder* é a mesma do *encoder*.

```python
decoder_blk = DecoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5, 0)
decoder_blk.eval()
X = torch.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
```

(continues on next page)

(continued from previous page)

```
decoder_blk(X, state)[0].shape
```

```
torch.Size([2, 100, 24])
```

Agora construímos todo o *decoder* do transformador composto por instâncias `num_layers` de`DecoderBlock`. No final, uma camada totalmente conectada calcula a previsão para todos os possíveis tokens de saída `vocab_size`. Ambos os pesos de autoatenção do *decoder* e os pesos de atenção do *encoder-decoder* são armazenados para visualização posterior.

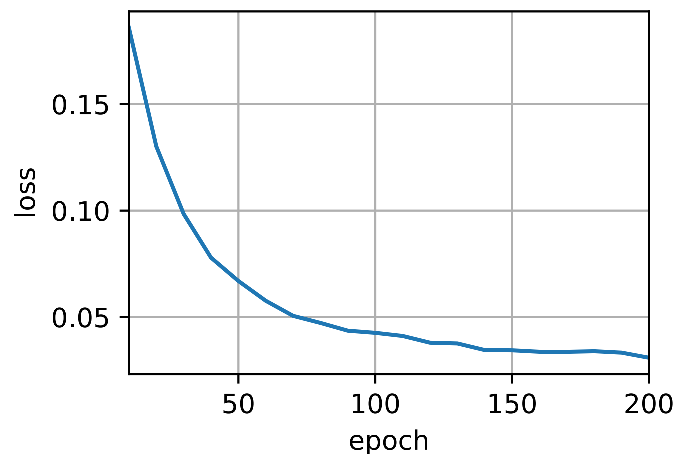
```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                DecoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, i))
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        return [enc_outputs, enc_valid_lens, [None] * self.num_layers]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

10.7.6 Treinamento

Vamos instanciar um modelo de *encoder-decoder* seguindo a arquitetura do transformador. Aqui nós especificamos que tanto o *encoder* do transformador quanto o *decoder* do transformador têm 2 camadas usando a atenção de 4 *heads*. Semelhante a Section 9.7.4, nós treinamos o modelo do transformador para aprendizado de sequência para sequência no conjunto de dados de tradução automática inglês-francês.

```
num_hiddens, num_layers, dropout, batch_size, num_steps = 32, 2, 0.1, 64, 10
lr, num_epochs, device = 0.005, 200, d2l.try_gpu()
ffn_num_input, ffn_num_hiddens, num_heads = 32, 64, 4
key_size, query_size, value_size = 32, 32, 32
norm_shape = [32]

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)

encoder = TransformerEncoder(
    len(src_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
decoder = TransformerDecoder(
    len(tgt_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.031, 5006.3 tokens/sec on cuda:0
```



Após o treinamento, nós usamos o modelo do transformador para traduzir algumas frases em inglês para o francês e calcular suas pontuações BLEU.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
```

(continues on next page)

(continued from previous page)

```
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 0.687
he's calm . => il est calme .,  bleu 1.000
i'm home . => je suis chez moi .,  bleu 1.000
```

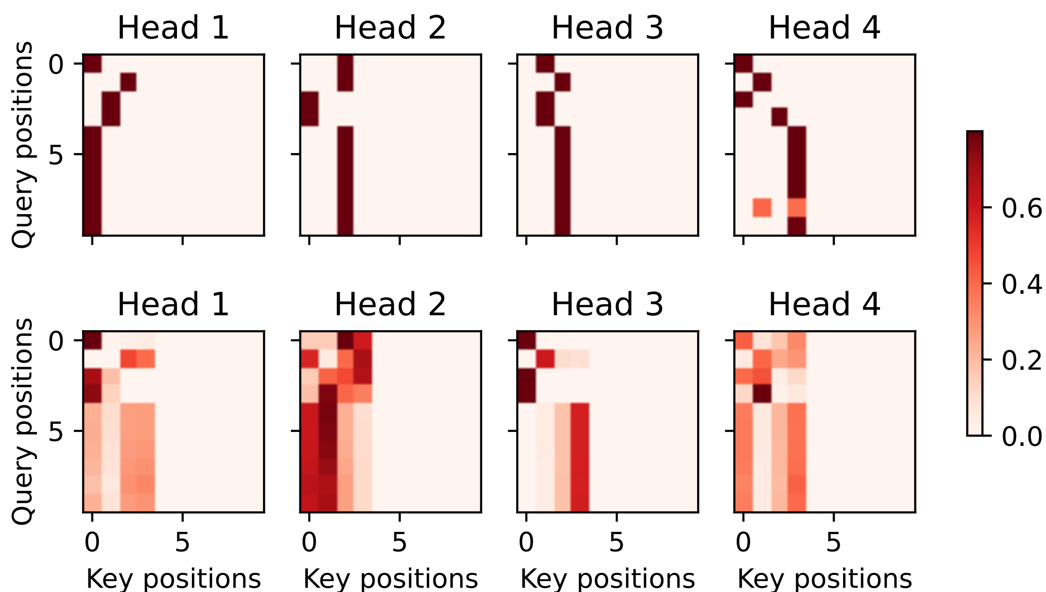
Vamos visualizar os pesos de atenção do transformador ao traduzir a última frase em inglês para o francês. A forma dos pesos de autoatenção do *encoder* é (número de camadas do codificador, número de *heads* de atenção, `num_steps` ou número de consultas,`num_steps` ou número de pares de valores-chave).

```
enc_attention_weights = torch.cat(net.encoder.attention_weights, 0).reshape((num_layers, num_
↪heads,
    -1, num_steps))
enc_attention_weights.shape
```

```
torch.Size([2, 4, 10, 10])
```

Na autoatenção do *encoder*, tanto as consultas quanto as chaves vêm da mesma sequência de entrada. Como os tokens de preenchimento não têm significado, com o comprimento válido especificado da sequência de entrada, nenhuma consulta atende às posições dos tokens de preenchimento. A seguir, duas camadas de pesos de atenção de várias cabeças são apresentadas linha por linha. Cada *head* participa independentemente com base em subespaços de representação separados de consultas, chaves e valores.

```
d2l.show_heatmaps(
    enc_attention_weights.cpu(), xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



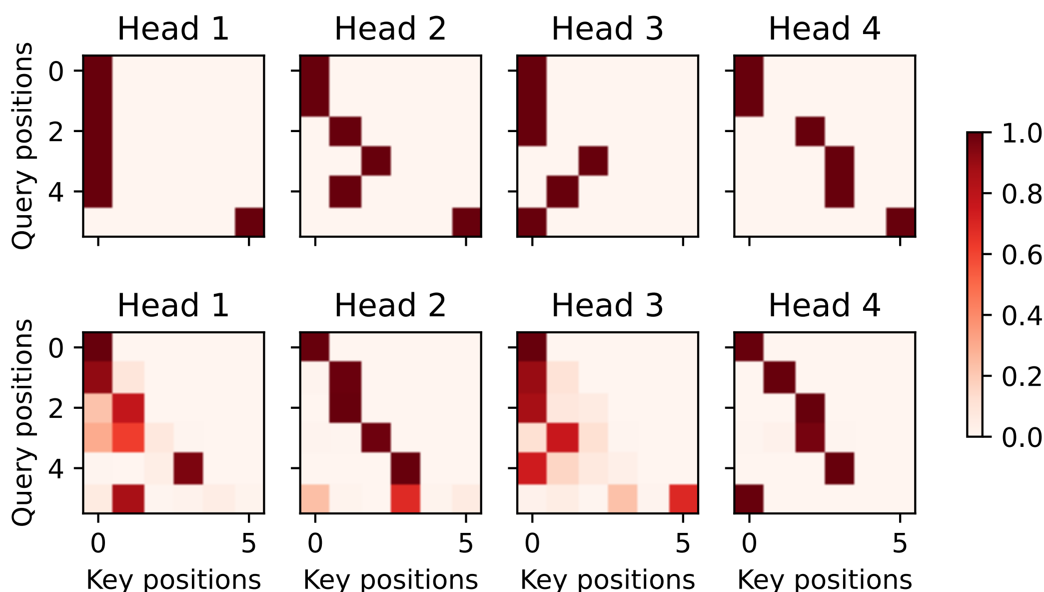
Para visualizar os pesos de autoatenção do *decoder* e os pesos de atenção do *encoder-decoder*, precisamos de mais manipulações de dados. Por exemplo, preenchemos os pesos de atenção mascarados com zero. Observe que os pesos de atenção do *decoder* e os pesos de atenção do *encoder-decoder* têm as mesmas consultas: o token de início de sequência seguido pelos tokens de saída.

```
dec_attention_weights_2d = [head[0].tolist()
                            for step in dec_attention_weight_seq
                            for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = torch.tensor(
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
dec_attention_weights = dec_attention_weights_filled.reshape((-1, 2, num_layers, num_heads,␣
↪num_steps))
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.permute(1, 2, 3, 0, 4)
dec_self_attention_weights.shape, dec_inter_attention_weights.shape
```

```
(torch.Size([2, 4, 6, 10]), torch.Size([2, 4, 6, 10]))
```

Devido à propriedade auto-regressiva da autoatenção do *decoder* nenhuma consulta atende aos pares de valores-chave após a posição da consulta.

```
# Plus one to include the beginning-of-sequence token
d2l.show_heatmaps(
    dec_self_attention_weights[:, :, :, :len(translation.split()) + 1],
    xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```



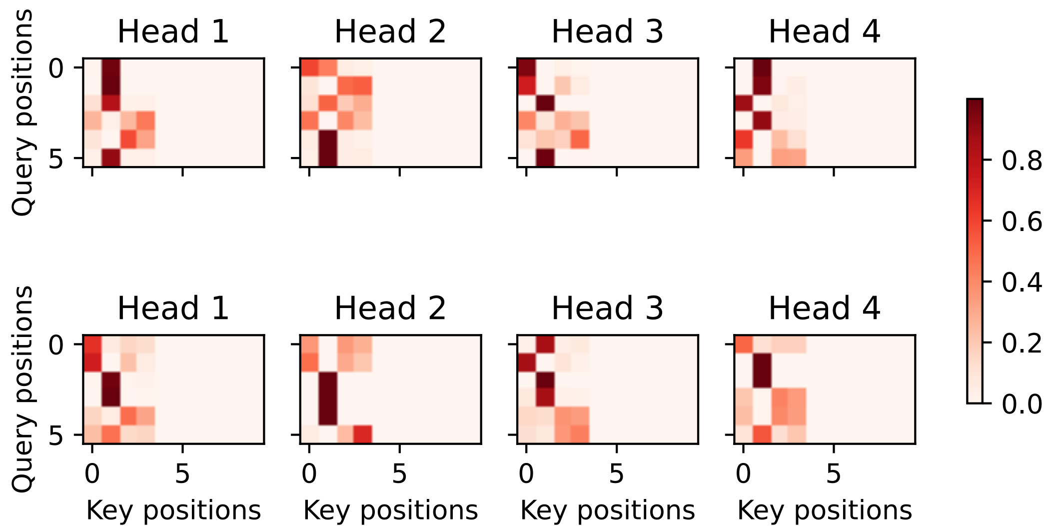
Semelhante ao caso da autoatenção do *encoder*, por meio do comprimento válido especificado da sequência de entrada, nenhuma consulta da sequência de saída atende a esses tokens de preenchimento da sequência de entrada.

```
d2l.show_heatmaps(
    dec_inter_attention_weights, xlabel='Key positions',
```

(continues on next page)

(continued from previous page)

```
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



Embora a arquitetura do transformador tenha sido proposta originalmente para o aprendizado de sequência a sequência, como descobriremos mais tarde neste livro, ou o *encoder* do transformador ou o *decoder* do transformador geralmente é usado individualmente para diferentes tarefas de *deep learning*.

10.7.7 Resumo

- O transformador é uma instância da arquitetura do *encoder-decoder*, embora o *encoder* ou *decoder* possam ser usados individualmente na prática.
- No transformador, a autoatenção com várias *heads* é usada para representar a sequência de entrada e a sequência de saída, embora o *decoder* tenha que preservar a propriedade auto-regressiva por meio de uma versão mascarada.
- Ambas as conexões residuais e a normalização da camada no transformador são importantes para treinar um modelo muito profundo.
- A rede *feed-forward* posicional no modelo do transformador transforma a representação em todas as posições de sequência usando o mesmo MLP.

10.7.8 Exercícios

1. Treine um transformador mais profundo nos experimentos. Como isso afeta a velocidade de treinamento e o desempenho da tradução?
2. É uma boa ideia substituir a atenção do produto escalonado com atenção aditiva no transformador? Por quê?
3. Para modelagem de linguagem, devemos usar o *encoder* do transformador, o *decoder* ou ambos? Como projetar este método?

4. Quais podem ser os desafios para os transformadores se as sequências de entrada forem muito longas? Por quê?
5. Como melhorar a eficiência computacional e de memória de transformadores? Dica: você pode consultar o artigo de pesquisa de Tay et al. (Tay et al., 2020).
6. Como podemos projetar modelos baseados em transformadores para tarefas de classificação de imagens sem usar CNNs? Dica: você pode consultar o transformador de visão (Dosovitskiy et al., 2021).

Discussions[107]

[107] https://discuss.d2l.ai/t/1066

11 | Algoritmos de Otimização

Se você leu o livro em sequência até agora, já usou vários algoritmos de otimização para treinar modelos de aprendizado profundo. Foram as ferramentas que nos permitiram continuar atualizando os parâmetros do modelo e minimizar o valor da função perda, conforme avaliado no conjunto de treinamento. Na verdade, qualquer pessoa que se contentar em tratar a otimização como um dispositivo de caixa preta para minimizar as funções objetivas em um ambiente simples pode muito bem se contentar com o conhecimento de que existe uma série de encantamentos de tal procedimento (com nomes como "SGD" e "Adam")

Para se sair bem, entretanto, é necessário algum conhecimento mais profundo. Os algoritmos de otimização são importantes para o aprendizado profundo. Por um lado, treinar um modelo complexo de aprendizado profundo pode levar horas, dias ou até semanas. O desempenho do algoritmo de otimização afeta diretamente a eficiência de treinamento do modelo. Por outro lado, compreender os princípios de diferentes algoritmos de otimização e a função de seus hiperparâmetros nos permitirá ajustar os hiperparâmetros de maneira direcionada para melhorar o desempenho dos modelos de aprendizado profundo.

Neste capítulo, exploramos algoritmos comuns de otimização de aprendizagem profunda em profundidade. Quase todos os problemas de otimização que surgem no aprendizado profundo são * não convexos . *No entanto, o projeto e a análise de algoritmos no contexto de problemas* convexos * provaram ser muito instrutivos. É por essa razão que este capítulo inclui uma cartilha sobre otimização convexa e a prova para um algoritmo de descida gradiente estocástico muito simples em uma função objetivo convexa.

11.1 Optimização e Deep Learning

Nesta seção, discutiremos a relação entre a otimização e o aprendizado profundo, bem como os desafios de usar a otimização no aprendizado profundo. Para um problema de aprendizado profundo, geralmente definiremos uma *função de perda* primeiro. Uma vez que temos a função de perda, podemos usar um algoritmo de otimização na tentativa de minimizar a perda. Na otimização, uma função de perda é frequentemente referida como a *função objetivo* do problema de otimização. Por tradição e convenção, a maioria dos algoritmos de otimização se preocupa com a *minimização*. Se alguma vez precisarmos maximizar um objetivo, há uma solução simples: basta virar o sinal no objetivo.

11.1.1 Objetivos da Optimização

Embora a otimização forneça uma maneira de minimizar a função de perda para profundas aprendizagem, em essência, as metas de otimização e aprendizagem profunda são fundamentalmente diferente. O primeiro se preocupa principalmente em minimizar um objetivo, enquanto o último está preocupado em encontrar um modelo adequado, dado um quantidade finita de dados. Em :numref: `sec_model_selection`, discutimos a diferença entre esses dois objetivos em detalhes. Por exemplo, erro de treinamento e erro de generalização geralmente diferem: uma vez que o objetivo função do algoritmo de otimização é geralmente uma função de perda com base no conjunto de dados de treinamento, o objetivo da otimização é reduzir o erro de treinamento. No entanto, o objetivo do aprendizado profundo (ou mais amplamente, inferência estatística) é reduzir o erro de generalização. Para realizar o último, precisamos pagar atenção ao *overfitting*, além de usar o algoritmo de otimização para reduzir o erro de treinamento.

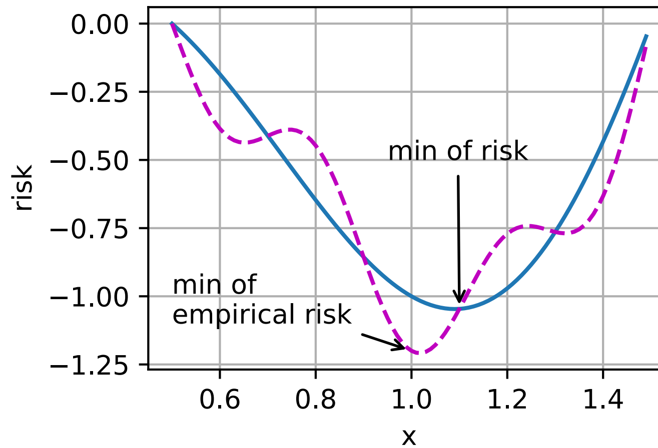
```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

Para ilustrar os diferentes objetivos mencionados acima, deixe-nos considerar o risco empírico e o risco. Conforme descrito in :numref: `subsec_empirical-risk-and-risk`, o risco empírico é uma perda média no conjunto de dados de treinamento enquanto o risco é a perda esperada em toda a população de dados. Abaixo, definimos duas funções: a função de risco `f` e a função de risco empírica `g`. Suponha que tenhamos apenas uma quantidade finita de dados de treinamento. Como resultado, aqui `g` é menos suave do que `f`.

```
def f(x):
    return x * torch.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * torch.cos(5 * np.pi * x)
```

O gráfico abaixo ilustra que o mínimo do risco empírico em um conjunto de dados de treinamento pode estar em um local diferente do mínimo do risco (erro de generalização).

```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = torch.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```



11.1.2 Desafios de otimização em Deep Learning

Neste capítulo, vamos nos concentrar especificamente no desempenho dos algoritmos de otimização para minimizar a função objetivo, ao invés de um erro de generalização do modelo. Em :numref: `sec_linear_regression` distinguimos entre soluções analíticas e soluções numéricas em problemas de otimização. No aprendizado profundo, a maioria das funções objetivas são complicadas e não possuem soluções analíticas. Em vez disso, devemos usar algoritmos de otimização. Os algoritmos de otimização neste capítulo todos caem nisso categoria.

Existem muitos desafios na otimização do aprendizado profundo. Alguns dos mais irritantes são mínimos locais, pontos de sela e gradientes de desaparecimento. Vamos dar uma olhada neles.

Minimo Local

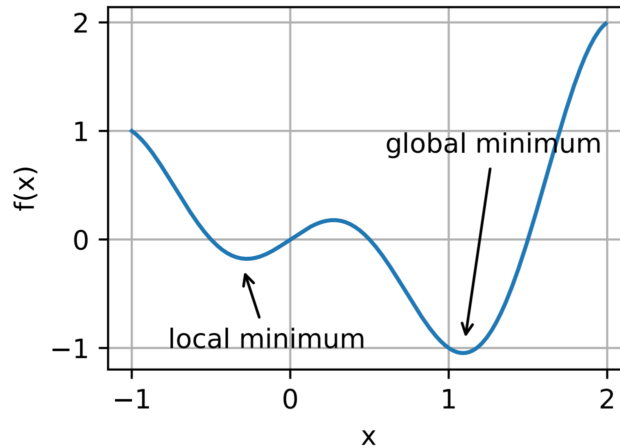
Para qualquer função objetivo $f(x)$, se o valor de $f(x)$ em x for menor do que os valores de $f(x)$ em quaisquer outros pontos nas proximidades de x, então $f(x)$ poderia ser um mínimo local . Se o valor de $f(x)$ em x é o mínimo da função objetivo em todo o domínio, então $f(x)$ é o mínimo global.

Por exemplo, dada a função

$$f(x) = x \cdot \text{cos}(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \tag{11.1.1}$$

podemos aproximar o mínimo local e o mínimo global desta função.

```
x = torch.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

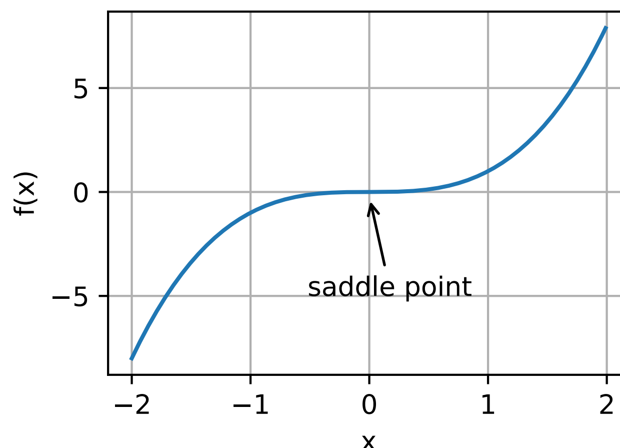


A função objetivo dos modelos de aprendizado profundo geralmente tem muitos ótimos locais. Quando a solução numérica de um problema de otimização está próxima do ótimo local, a solução numérica obtida pela iteração final pode apenas minimizar a função objetivo *localmente*, ao invés de *globalmente*, conforme o gradiente das soluções da função objetivo se aproxima ou torna-se zero . Apenas algum grau de ruído pode derrubar o parâmetro do mínimo local. Na verdade, esta é uma das propriedades benéficas de descida gradiente estocástica do minibatch onde a variação natural dos gradientes sobre os minibatches é capaz de deslocar os parâmetros dos mínimos locais.

Pontos de Sela

Além dos mínimos locais, os pontos de sela são outra razão para o desaparecimento dos gradientes. Um * ponto de sela * é qualquer local onde todos os gradientes de uma função desaparecem, mas que não é um mínimo global nem local. Considere a função $f(x) = x^3$. Sua primeira e segunda derivadas desaparecem para $x = 0$. A otimização pode parar neste ponto, embora não seja o mínimo.

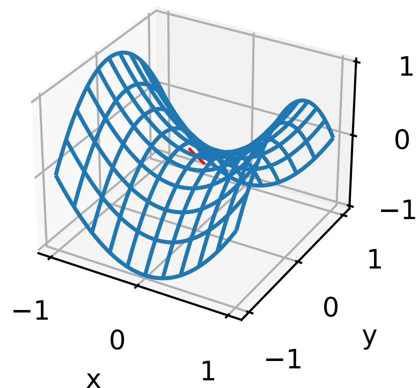
```
x = torch.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```



Os pontos de sela em dimensões mais altas são ainda mais insidiosos, como mostra o exemplo

abaixo. Considere a função $f(x, y) = x^2 - y^2$. Ele tem seu ponto de sela em $(0, 0)$. Este é um máximo em relação a y e um mínimo em relação a x. Além disso, *se parece* com uma sela, que é onde essa propriedade matemática recebeu seu nome.

```
x, y = torch.meshgrid(
    torch.linspace(-1.0, 1.0, 101), torch.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```



Assumimos que a entrada de uma função é um vetor k -dimensional e seu a saída é um escalar, então sua matriz Hessiana terá k autovalores (consulte o apêndice online sobre eigendecompositions[108]). A solução do função pode ser um mínimo local, um máximo local ou um ponto de sela em um posição onde o gradiente da função é zero:

- Quando os autovalores da matriz Hessiana da função na posição do gradiente zero são todos positivos, temos um mínimo local para a função.
- Quando os valores próprios da matriz Hessiana da função na posição do gradiente zero são todos negativos, temos um máximo local para a função.
- Quando os valores próprios da matriz Hessiana da função na posição do gradiente zero são negativos e positivos, temos um ponto de sela para a função.

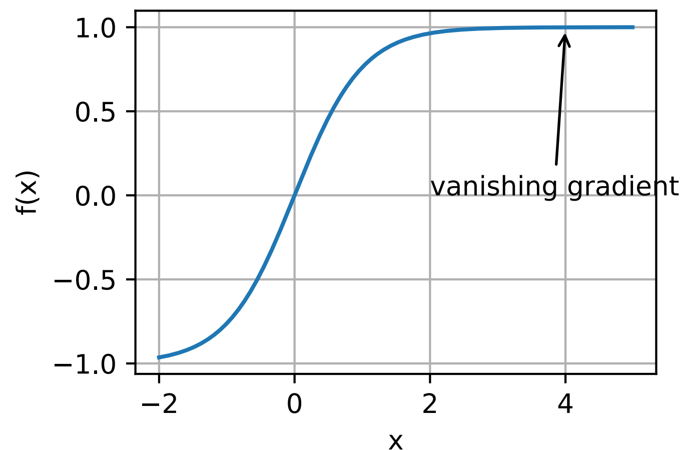
Para problemas de alta dimensão, a probabilidade de que pelo menos * alguns * dos autovalores sejam negativos é bastante alta. Isso torna os pontos de sela mais prováveis do que os mínimos locais. Discutiremos algumas exceções a essa situação na próxima seção, ao introduzir a convexidade. Em suma, funções convexas são aquelas em que os autovalores do Hessiano nunca são negativos. Infelizmente, porém, a maioria dos problemas de aprendizado profundo não se enquadra nessa categoria. No entanto, é uma ótima ferramenta para estudar algoritmos de otimização.

[108] https://d2l.ai/chapter_apencha-mathematics-for-deep-learning/eigendecomposition.html

Gradiente de Desaparecimento

Provavelmente, o problema mais insidioso a ser encontrado é o gradiente de desaparecimento. Lembre-se de nossas funções de ativação comumente usadas e seus derivados em: numref: `subsec_activation-functions`. Por exemplo, suponha que queremos minimizar a função $f(x) = \tanh(x)$ e começamos em $x = 4$. Como podemos ver, o gradiente de f é quase nulo. Mais especificamente, $f'(x) = 1 - \tanh^2(x)$ e portanto $f'(4) = 0.0013$. Conseqüentemente, a otimização ficará parada por um longo tempo antes de progredirmos. Isso acabou sendo um dos motivos pelos quais treinar modelos de aprendizado profundo era bastante complicado antes da introdução da função de ativação ReLU.

```
x = torch.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [torch.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```



Como vimos, a otimização para aprendizado profundo está cheia de desafios. Felizmente, existe uma gama robusta de algoritmos que funcionam bem e são fáceis de usar, mesmo para iniciantes. Além disso, não é realmente necessário encontrar a melhor solução. Ótimos locais ou mesmo soluções aproximadas deles ainda são muito úteis.

11.1.3 Sumário

- Minimizar o erro de treinamento *não* garante que encontraremos o melhor conjunto de parâmetros para minimizar o erro de generalização.
- Os problemas de otimização podem ter muitos mínimos locais.
- O problema pode ter ainda mais pontos de sela, pois geralmente os problemas não são convexos.
- O desaparecimento de gradientes pode causar o travamento da otimização. Frequentemente, uma reparametrização do problema ajuda. Uma boa inicialização dos parâmetros também pode ser benéfica.

11.1.4 Exercícios

1. Considere um MLP simples com uma única camada oculta de, digamos, d dimensões na camada oculta e uma única saída. Mostre que, para qualquer mínimo local, existem pelo menos $d!$ Soluções equivalentes que se comportam de forma idêntica.
2. Suponha que temos uma matriz aleatória simétrica **M** onde as entradas $M_{ij} = M_{ji}$ são, cada um, extraídos de alguma distribuição de probabilidade p_{ij}. Além disso, assuma que $p_{ij}(x) = p_{ij}(-x)$, ou seja, que o a distribuição é simétrica (consulte, por exemplo (Wigner, 1958) para obter detalhes).
    1. Prove que a distribuição sobre os autovalores também é simétrica. Ou seja, para qualquer autovetor **v** a probabilidade de que o autovalor associado λ satisfaça $P(\lambda > 0) = P(\lambda < 0)$.
    2. Por que o acima *não* implica $P(\lambda > 0) = 0.5$?
3. Em que outros desafios envolvidos na otimização do aprendizado profundo você consegue pensar?
4. Suponha que você deseja equilibrar uma bola (real) em uma sela (real).
    1. Por que isso é difícil?
    2. Você pode explorar esse efeito também para algoritmos de otimização?

Discussão[109]

11.2 Convexidade

A convexidade desempenha um papel vital no projeto de algoritmos de otimização. Em grande parte, isso se deve ao fato de que é muito mais fácil analisar e testar algoritmos em tal contexto. Em outras palavras, se o algoritmo funcionar mal, mesmo na configuração convexa, normalmente, não devemos esperar ver grandes resultados de outra forma. Além disso, embora os problemas de otimização no aprendizado profundo sejam geralmente não convexos, eles costumam exibir algumas propriedades dos convexos próximos aos mínimos locais. Isso pode levar a novas variantes de otimização interessantes, como (Izmailov et al., 2018).

```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

[109] https://discuss.d2l.ai/t/487

11.2.1 Definições

Antes da análise convexa, precisamos definir *conjuntos convexos* e *funções convexas*. Eles levam a ferramentas matemáticas que são comumente aplicadas ao aprendizado de máquina.

Convex Sets

Os conjuntos são a base da convexidade. Simplificando, um conjunto $\mathcal{X}$ em um espaço vetorial é *convexo* se para qualquer $a, b \in \mathcal{X}$ o segmento de linha conectando a e b também estiver em $\mathcal{X}$. Em termos matemáticos, isso significa que para todos $\lambda \in [0, 1]$ temos

$$\lambda a + (1 - \lambda) b \in \mathcal{X} \text{ whenever } a, b \in \mathcal{X}. \quad (11.2.1)$$

Isso soa um pouco abstrato. Considere Fig. 11.2.1. O primeiro conjunto não é convexo, pois existem segmentos de linha que não estão contidos nele. Os outros dois conjuntos não sofrem esse problema.

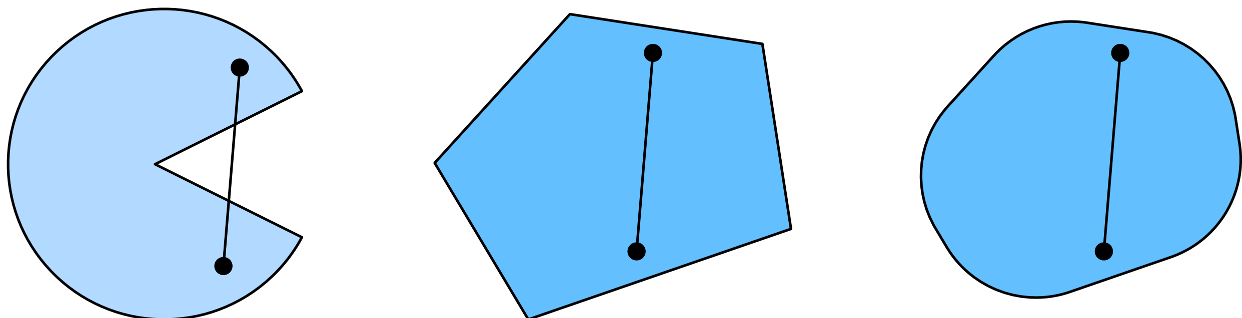


Fig. 11.2.1: O primeiro conjunto é não convexo e os outros dois são convexos.

As definições por si só não são particularmente úteis, a menos que você possa fazer algo com elas. Neste caso, podemos olhar as interseções como mostrado em Fig. 11.2.2. Suponha que $\mathcal{X}$ e $\mathcal{Y}$ são conjuntos convexos. Então $\mathcal{X} \cap \mathcal{Y}$ também é convexo. Para ver isso, considere qualquer $a, b \in \mathcal{X} \cap \mathcal{Y}$. Como $\mathcal{X}$ e $\mathcal{Y}$ são convexos, os segmentos de linha que conectam a e b estão contidos em $\mathcal{X}$ e $\mathcal{Y}$. Dado isso, eles também precisam estar contidos em $\mathcal{X} \cap \mathcal{Y}$, provando assim nosso teorema.

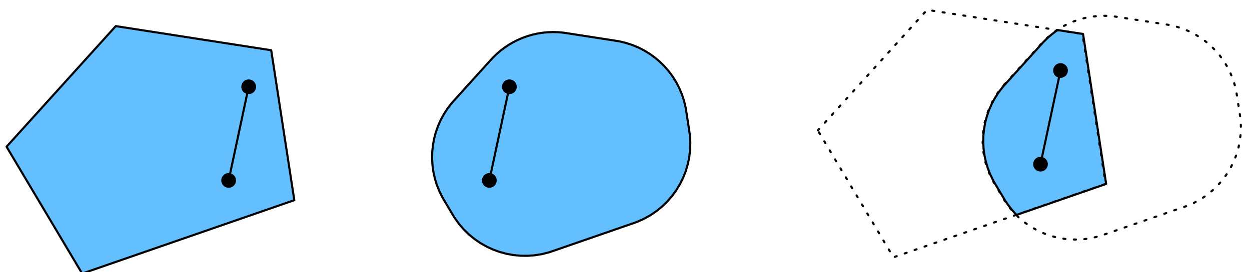


Fig. 11.2.2: A interseção entre dois conjuntos convexos é convexa.

Podemos reforçar este resultado com pouco esforço: dados os conjuntos convexos $\mathcal{X}_i$, sua interseção $\cap_i \mathcal{X}_i$ é convexa. Para ver que o inverso não é verdadeiro, considere dois conjuntos disjuntos $\mathcal{X} \cap \mathcal{Y} = \emptyset$. Agora escolha $a \in \mathcal{X}$ e $b \in \mathcal{Y}$. O segmento de linha em Fig. 11.2.3 conectando a e b precisa conter alguma parte que não está em $\mathcal{X}$ nem em $\mathcal{Y}$, uma vez que assumimos que $\mathcal{X} \cap \mathcal{Y} = \emptyset$.

Consequentemente, o segmento de linha também não está em $\mathcal{X} \cup \mathcal{Y}$, provando assim que, em geral, as uniões de conjuntos convexos não precisam ser convexas.

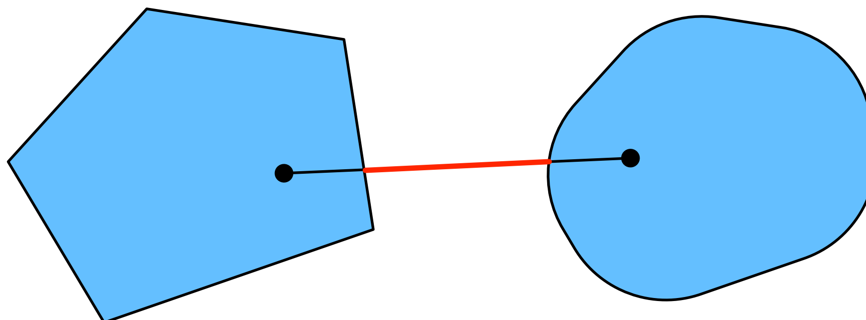


Fig. 11.2.3: A união de dois conjuntos convexos não precisa ser convexa.

Normalmente, os problemas de aprendizado profundo são definidos em conjuntos convexos. Por exemplo, $\mathbb{R}^d$, o conjunto de vetores d-dimensionais de números reais, é um conjunto convexo (afinal, a linha entre quaisquer dois pontos em $\mathbb{R}^d$ permanece em $\mathbb{R}^d$). Em alguns casos, trabalhamos com variáveis de comprimento limitado, como bolas de raio r conforme definido por $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ e } \|\mathbf{x}\| \leq r\}$.

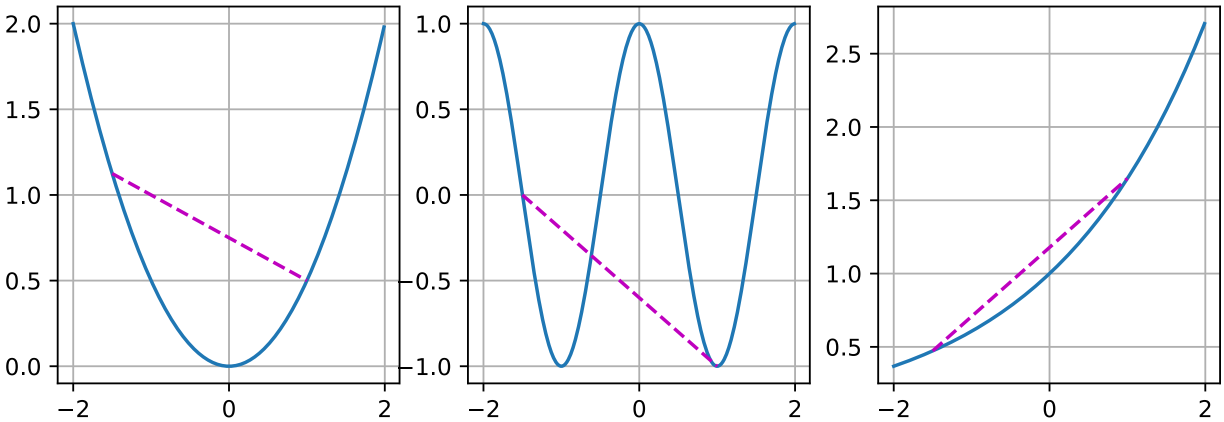
Função Convexa

Agora que temos conjuntos convexos, podemos introduzir *funções convexas* f. Dado um conjunto convexo $\mathcal{X}$, uma função $f : \mathcal{X} \to \mathbb{R}$ é *convexa* se para todos $x, x' \in \mathcal{X}$ e para todos $\lambda \in [0, 1]$ temos

$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x'). \tag{11.2.2}$$

Para ilustrar isso, vamos representar graficamente algumas funções e verificar quais satisfazem o requisito. Abaixo definimos algumas funções, convexas e não convexas.

```
f = lambda x: 0.5 * x**2  # Convex
g = lambda x: torch.cos(np.pi * x)  # Nonconvex
h = lambda x: torch.exp(0.5 * x)  # Convex

x, segment = torch.arange(-2, 2, 0.01), torch.tensor([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```



Como esperado, a função cosseno é *não convexa*, enquanto a parábola e a função exponencial são. Observe que o requisito de que $\mathcal{X}$ seja um conjunto convexo é necessário para que a condição faça sentido. Caso contrário, o resultado de $f(\lambda x + (1 - \lambda)x')$ pode não ser bem definido.

Desigualdades de Jensen

Dada uma função convexa f, uma das ferramentas matemáticas mais úteis é a *desigualdade de Jensen*. Isso equivale a uma generalização da definição de convexidade:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_X[f(X)] \geq f\left(E_X[X]\right), \tag{11.2.3}$$

onde α_i são números reais não negativos tais que $\sum_i \alpha_i = 1$ e X é uma variável aleatória. Em outras palavras, a expectativa de uma função convexa não é menos do que a função convexa de uma expectativa, onde a última é geralmente uma expressão mais simples. Para provar a primeira desigualdade, aplicamos repetidamente a definição de convexidade a um termo da soma de cada vez.

Uma das aplicações comuns da desigualdade de Jensen é para ligar uma expressão mais complicada por uma mais simples. Por exemplo, sua aplicação pode ser no que diz respeito ao log-verossimilhança de variáveis aleatórias parcialmente observadas. Ou seja, usamos

$$E_{Y \sim P(Y)}[-\log P(X \mid Y)] \geq -\log P(X), \tag{11.2.4}$$

uma vez que $\int P(Y)P(X \mid Y)dY = P(X)$. Isso pode ser usado em métodos variacionais. Aqui Y é normalmente a variável aleatória não observada, $P(Y)$ é a melhor estimativa de como ela pode ser distribuída e $P(X)$ é a distribuição com Y integrada. Por exemplo, no agrupamento Y podem ser os rótulos de cluster e $P(X \mid Y)$ é o modelo gerador ao aplicar rótulos de cluster.

11.2.2 Propriedades

As funções convexas têm algumas propriedades úteis. Nós os descrevemos abaixo.

Mínimos locais são mínimos globais

Em primeiro lugar, os mínimos locais das funções convexas também são os mínimos globais. Podemos provar isso por contradição como segue.

Considere uma função convexa f definida em um conjunto convexo $\mathcal{X}$. Suponha que $x^* \in \mathcal{X}$ seja um mínimo local: existe um pequeno valor positivo p de forma que para $x \in \mathcal{X}$ que satisfaz $0 < |x - x^*| \leq p$ temos $f(x^*) < f(x)$.

Suponha que o mínimo local x^* não é o mínimo global de f: existe $x' \in \mathcal{X}$ para o qual $f(x') < f(x^*)$. Também existe $\lambda \in [0, 1)$ como $\lambda = 1 - \frac{p}{|x^* - x'|}$ de modo a $0 < |\lambda x^* + (1 - \lambda)x' - x^*| \leq p$.

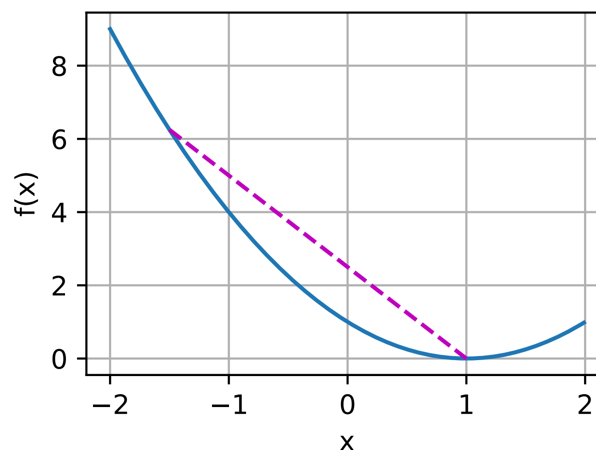
Contudo, de acordo com a definição de funções convexas, temos

$$\begin{aligned}
f(\lambda x^* + (1-\lambda) x') &\leq \lambda f(x^*) + (1-\lambda) f(x') \\
&< \lambda f(x^*) + (1-\lambda) f(x^*) \\
&= f(x^*),
\end{aligned} \tag{11.2.5}$$

o que contradiz nossa afirmação de que x^* é um mínimo local. Portanto, não existe $x' \in \mathcal{X}$ para o qual $f(x') < f(x^*)$. O mínimo local x^* também é o mínimo global.

Por exemplo, a função convexa $f(x) = (x - 1)^2$ tem um mínimo local em $x = 1$, que também é o mínimo global.

```
f = lambda x: (x - 1) ** 2
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



O fato de que os mínimos locais para funções convexas também são os mínimos globais é muito conveniente. Isso significa que, se minimizarmos as funções, não podemos "ficar presos". Observe, porém, que isso não significa que não possa haver mais de um mínimo global ou que possa mesmo existir um. Por exemplo, a função $f(x) = \mathrm{max}(|x| - 1, 0)$ atinge seu valor mínimo no intervalo $[-1, 1]$. Por outro lado, a função $f(x) = \exp(x)$ não atinge um valor mínimo em $\mathbb{R}$: para $x \to -\infty$ ele assíntotas para 0, mas não há x para o qual $f(x) = 0$.

Os conjuntos de funções convexas abaixo são convexos

Nós podemos convenientemente definir conjuntos convexos via *conjuntos abaixo* de funções convexas. Concretamente, dada uma função convexa f definida em um conjunto convexo $\mathcal{X}$, qualquer conjunto abaixo

$$\mathcal{S}_b := \{x | x \in \mathcal{X} \text{ and } f(x) \leq b\} \tag{11.2.6}$$

é convexo.

Vamos provar isso rapidamente. Lembre-se de que para qualquer $x, x' \in \mathcal{S}_b$ precisamos mostrar que $\lambda x + (1 - \lambda) x' \in \mathcal{S}_b$ enquanto $\lambda \in [0, 1]$. Como $f(x) \leq b$ e $f(x') \leq b$, pela definição de convexidade, temos

$$f(\lambda x + (1 - \lambda) x') \leq \lambda f(x) + (1 - \lambda) f(x') \leq b. \tag{11.2.7}$$

Convexidade e derivados secundários

Sempre que a segunda derivada de uma função $f : \mathbb{R}^n \to \mathbb{R}$ existe, é muito fácil verificar se f é convexa. Tudo o que precisamos fazer é verificar se o hessiano de f é semidefinido positivo: $\nabla^2 f \succeq 0$, ou seja, denotando a matriz Hessiana $\nabla^2 f$ por $\mathbf{H}$, $\mathbf{x}^\top \mathbf{H} \mathbf{x} \geq 0$ para todo $\mathbf{x} \in \mathbb{R}^n$. Por exemplo, a função $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{x}\|^2$ é convexa, pois $\nabla^2 f = \mathbf{1}$, ou seja, seu Hessian é uma matriz de identidade.

Formalmente, qualquer função unidimensional duas vezes diferenciável $f : \mathbb{R} \to \mathbb{R}$ é convexa se e somente se sua segunda derivada $f'' \geq 0$. Para qualquer função multidimensional duas vezes diferenciável $f : \mathbb{R}^n \to \mathbb{R}$, é convexo se e somente se for Hessiano $\nabla^2 f \succeq 0$.

Primeiro, precisamos provar o caso unidimensional. Para ver isso convexidade de f implica $f'' \geq 0$ usamos o fato de que

$$\frac{1}{2} f(x + \epsilon) + \frac{1}{2} f(x - \epsilon) \geq f\left(\frac{x + \epsilon}{2} + \frac{x - \epsilon}{2}\right) = f(x). \tag{11.2.8}$$

Uma vez que a segunda derivada é dada pelo limite sobre diferenças finitas, segue-se que

$$f''(x) = \lim_{\epsilon \to 0} \frac{f(x + \epsilon) + f(x - \epsilon) - 2f(x)}{\epsilon^2} \geq 0. \tag{11.2.9}$$

Para ver isso $f'' \geq 0$ implica que f é convexo usamos o fato de que $f'' \geq 0$ implica que f' é uma função monotonicamente não decrescente. Sejam $a < x < b$ três pontos em $\mathbb{R}$, onde $x = (1 - \lambda)a + \lambda b$ e $\lambda \in (0, 1)$. De acordo com o teorema do valor médio, existem $\alpha \in [a, x]$ e $\beta \in [x, b]$ de tal modo que

$$f'(\alpha) = \frac{f(x) - f(a)}{x - a} \text{ and } f'(\beta) = \frac{f(b) - f(x)}{b - x}. \tag{11.2.10}$$

Por monotonicidade $f'(\beta) \geq f'(\alpha)$, por isso

$$\frac{x - a}{b - a} f(b) + \frac{b - x}{b - a} f(a) \geq f(x). \tag{11.2.11}$$

De $x = (1 - \lambda)a + \lambda b$, temos

$$\lambda f(b) + (1 - \lambda) f(a) \geq f((1 - \lambda)a + \lambda b), \tag{11.2.12}$$

provando assim convexidade.

Em segundo lugar, precisamos de um lema antes comprovando o caso multidimensional: $f : \mathbb{R}^n \to \mathbb{R}$ é convexo se e somente se para todos $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1-z)\mathbf{y}) \text{ where } z \in [0,1] \tag{11.2.13}$$

é convexo.

Para provar que a convexidade de f implica que g é convexo, podemos mostrar que para todos $a, b, \lambda \in [0,1]$ (portanto $0 \leq \lambda a + (1-\lambda)b \leq 1$)

$$\begin{aligned}
&g(\lambda a + (1-\lambda)b)\\
=&f\left(\left(\lambda a + (1-\lambda)b\right)\mathbf{x} + \left(1 - \lambda a - (1-\lambda)b\right)\mathbf{y}\right)\\
=&f\left(\lambda\left(a\mathbf{x} + (1-a)\mathbf{y}\right) + (1-\lambda)\left(b\mathbf{x} + (1-b)\mathbf{y}\right)\right)\\
\leq&\lambda f\left(a\mathbf{x} + (1-a)\mathbf{y}\right) + (1-\lambda)f\left(b\mathbf{x} + (1-b)\mathbf{y}\right)\\
=&\lambda g(a) + (1-\lambda)g(b).
\end{aligned} \tag{11.2.14}$$

Para provar o contrário, nós podemos mostrar isso para todo$\lambda \in [0,1]$

$$\begin{aligned}
&f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y})\\
=&g(\lambda \cdot 1 + (1-\lambda)\cdot 0)\\
\leq&\lambda g(1) + (1-\lambda)g(0)\\
=&\lambda f(\mathbf{x}) + (1-\lambda)g(\mathbf{y}).
\end{aligned} \tag{11.2.15}$$

Finalmente, usando o lema acima e o resultado do caso unidimensional, o caso multi-dimensional pode ser comprovado da seguinte forma. Uma função multidimensional $f : \mathbb{R}^n \to \mathbb{R}$ é convexa se e somente se para todos $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ $g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1-z)\mathbf{y})$, onde $z \in [0,1]$, é convexo. De acordo com o caso unidimensional, isso vale se e somente se $g'' = (\mathbf{x} - \mathbf{y})^\top \mathbf{H}(\mathbf{x} - \mathbf{y}) \geq 0$ ($\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f$) para todo $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, que é equivalente a $\mathbf{H} \succeq 0$ de acordo com a definição de matrizes semidefinidas positivas.

11.2.3 Restrições

Uma das boas propriedades da otimização convexa é que ela nos permite lidar com as restrições de maneira eficiente. Ou seja, permite-nos resolver problemas da forma:

$$\begin{aligned}
&\underset{\mathbf{x}}{\text{minimize}} \ f(\mathbf{x})\\
&\text{subject to } c_i(\mathbf{x}) \leq 0 \text{ for all } i \in \{1, \ldots, N\}.
\end{aligned} \tag{11.2.16}$$

Aqui f é o objetivo e as funções c_i são funções de restrição. Para ver o que isso faz, considere o caso em que $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$. Neste caso, os parâmetros $\mathbf{x}$ são restritos à bola unitária. Se uma segunda restrição é $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$, então isso corresponde a todos $\mathbf{x}$ em um meio-espaço. Satisfazer as duas restrições simultaneamente significa selecionar uma fatia de uma bola como o conjunto de restrições.

Função de Lagrange

Em geral, resolver um problema de otimização restrito é difícil. Uma maneira de lidar com isso vem da física com uma intuição bastante simples. Imagine uma bola dentro de uma caixa. A bola rolará para o local mais baixo e as forças da gravidade serão equilibradas com as forças que os lados da caixa podem impor à bola. Em suma, o gradiente da função objetivo (ou seja, a gravidade) será compensado pelo gradiente da função de restrição (necessidade de permanecer dentro da caixa em virtude das paredes "empurrando para trás"). Observe que qualquer restrição que não esteja ativa (ou seja, a bola não toca a parede) não será capaz de exercer qualquer força na bola.

Ignorando a derivação da função de Lagrange L (consulte, por exemplo, o livro de Boyd e Vandenberghe para obter detalhes (Boyd & Vandenberghe, 2004)), o raciocínio acima pode ser expresso através do seguinte problema de otimização do ponto de sela:

$$L(\mathbf{x}, \alpha) = f(\mathbf{x}) + \sum_i \alpha_i c_i(\mathbf{x}) \text{ where } \alpha_i \geq 0. \tag{11.2.17}$$

Aqui, as variáveis α_i são os chamados *Multiplicadores de Lagrange* que garantem que uma restrição seja devidamente aplicada. Eles são escolhidos apenas grandes o suficiente para garantir que $c_i(\mathbf{x}) \leq 0$ para todos os i. Por exemplo, para qualquer $\mathbf{x}$ para o qual $c_i(\mathbf{x}) < 0$ naturalmente, acabaríamos escolhendo $\alpha_i = 0$. Além disso, este é um problema de otimização *ponto de sela* em que se deseja *maximizar* L em relação a α e simultaneamente *minimizá-lo* em relação a $\mathbf{x}$. Existe uma vasta literatura explicando como chegar à função $L(\mathbf{x}, \alpha)$. Para nossos propósitos, é suficiente saber que o ponto de sela de L é onde o problema de otimização restrito original é resolvido de forma otimizada.

Penalidades

Uma maneira de satisfazer problemas de otimização restrita pelo menos aproximadamente é adaptar a função de Lagrange L. Em vez de satisfazer $c_i(\mathbf{x}) \leq 0$, simplesmente adicionamos $\alpha_i c_i(\mathbf{x})$ à função objetivo $f(x)$. Isso garante que as restrições não sejam violadas demais.

Na verdade, temos usado esse truque o tempo todo. Considere a diminuição do peso em Section 4.5. Nele adicionamos $\frac{\lambda}{2}\|\mathbf{w}\|^2$ à função objetivo para garantir que $\mathbf{w}$ não cresça muito. Usando o ponto de vista da otimização restrita, podemos ver que isso garantirá que $\|\mathbf{w}\|^2 - r^2 \leq 0$ para algum raio r. Ajustar o valor de λ nos permite variar o tamanho de $\mathbf{w}$.

Em geral, adicionar penalidades é uma boa maneira de garantir a satisfação aproximada da restrição. Na prática, isso acaba sendo muito mais robusto do que a satisfação exata. Além disso, para problemas não convexos, muitas das propriedades que tornam a abordagem exata tão atraente no caso convexo (por exemplo, otimização) não são mais válidas.

Projeções

Uma estratégia alternativa para satisfazer as restrições são as projeções. Novamente, nós os encontramos antes, por exemplo, ao lidar com recorte de gradiente em Section 8.5. Lá, garantimos que um gradiente tem comprimento limitado por c via

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \text{min}(1, c/\|\mathbf{g}\|). \tag{11.2.18}$$

Isso acaba sendo uma *projeção* de g na bola de raio c. Mais geralmente, uma projeção em um conjunto (convexo) $\mathcal{X}$ é definida como

$$\text{Proj}_\mathcal{X}(\mathbf{x}) = \mathop{\text{argmin}}_{\mathbf{x}' \in \mathcal{X}} \|\mathbf{x} - \mathbf{x}'\|_2. \tag{11.2.19}$$

Portanto, é o ponto mais próximo em $\mathcal{X}$ para $\mathbf{x}$. Isso soa um pouco abstrato. Fig. 11.2.4 explica um pouco mais claramente. Nele temos dois conjuntos convexos, um círculo e um diamante. Os pontos dentro do conjunto (amarelo) permanecem inalterados. Pontos fora do conjunto (preto) são mapeados para o ponto mais próximo dentro do conjunto (vermelho). Enquanto para bolas de L_2 isso não altera a direção, este não precisa ser o caso em geral, como pode ser visto no caso do diamante.

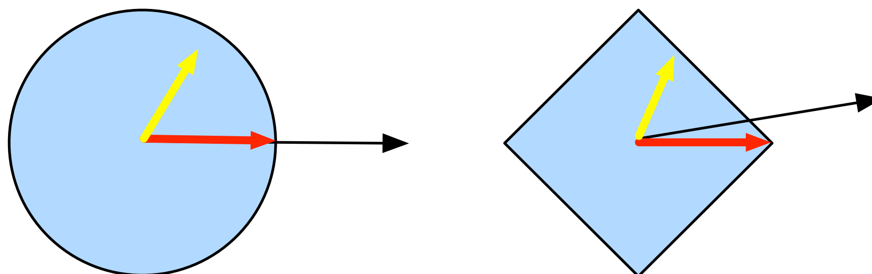


Fig. 11.2.4: Projeções convexas.

Um dos usos para projeções convexas é calcular vetores de peso esparsos. Neste caso, projetamos $\mathbf{w}$ em uma bola L_1 (a última é uma versão generalizada do diamante na imagem acima).

11.2.4 Sumário

No contexto de aprendizagem profunda, o principal objetivo das funções convexas é motivar algoritmos de otimização e nos ajudar a entendê-los em detalhes. A seguir, veremos como a descida gradiente e a descida gradiente estocástica podem ser derivadas de acordo.

- As interseções de conjuntos convexos são convexas. Os sindicatos não.
- A expectativa de uma função convexa é maior do que a função convexa de uma expectativa (desigualdade de Jensen).
- Uma função duas vezes diferenciável é convexa se, e somente se, sua segunda derivada tem apenas autovalores não negativos em toda a extensão.
- Restrições convexas podem ser adicionadas por meio da função Lagrange. Na prática, basta adicioná-los com uma penalidade à função objetivo.
- As projeções são mapeadas para pontos no conjunto (convexo) mais próximo do ponto original.

11.2.5 Exercícios

1. Suponha que queremos verificar a convexidade de um conjunto desenhando todas as linhas entre os pontos dentro do conjunto e verificando se as linhas estão contidas.
    1. Prove que é suficiente verificar apenas os pontos no limite.
    2. Prove que é suficiente verificar apenas os vértices do conjunto.
2. Denote por $\mathcal{B}_p[r] := \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_p \leq r\}$ a bola de raio r usando a norma p. Prove que $\mathcal{B}_p[r]$ é convexo para todos $p \geq 1$.

3. Dadas as funções convexas f e g mostram que $\max(f, g)$ também é convexo. Prove que $\min(f, g)$ não é convexo.
4. Prove que a normalização da função softmax é convexa. Mais especificamente, provar a convexidade de $f(x) = \log \sum_i \exp(x_i)$.
5. Prove que os subespaços lineares são conjuntos convexos, ou seja, $\mathcal{X} = \{\mathbf{x} | \mathbf{W}\mathbf{x} = \mathbf{b}\}$.
6. Prove que no caso de subespaços lineares com $\mathbf{b} = 0$ a projeção $\mathrm{Proj}_\mathcal{X}$ pode ser escrita como $\mathbf{M}\mathbf{x}$ para algumas matrizes $\mathbf{M}$.
7. Mostre que para funções convexas duas vezes diferenciáveis f podemos escrever $f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x + \xi)$ para algum $\xi \in [0, \epsilon]$.
8. Dado um vetor $\mathbf{w} \in \mathbb{R}^d$ com $\|\mathbf{w}\|_1 > 1$, calcule a projeção na bola unitária ℓ_1.
   1. Como etapa intermediária, escreva o objetivo penalizado $\|\mathbf{w} - \mathbf{w}'\|_2^2 + \lambda \|\mathbf{w}'\|_1$ e calcule a solução para um dado $\lambda > 0$.
   2. Você consegue encontrar o valor 'certo' de λ sem muitas tentativas e erros?
9. Dado um conjunto convexo $\mathcal{X}$ e dois vetores $\mathbf{x}$ e $\mathbf{y}$ provam que as projeções nunca aumentam as distâncias, ou seja, $\|\mathbf{x} - \mathbf{y}\| \geq \|\mathrm{Proj}_\mathcal{X}(\mathbf{x}) - \mathrm{Proj}_\mathcal{X}(\mathbf{y})\|$.

Discussions[110]

11.3 Gradiente descendente

Nesta seção, apresentaremos os conceitos básicos da descida gradiente. Isso é breve por necessidade. Veja, por exemplo, (Boyd & Vandenberghe, 2004) para uma introdução detalhada à otimização convexa. Embora o último raramente seja usado diretamente no aprendizado profundo, uma compreensão da descida gradiente é a chave para entender os algoritmos de descida gradiente estocásticos. Por exemplo, o problema de otimização pode divergir devido a uma taxa de aprendizado excessivamente grande. Este fenômeno já pode ser visto na descida do gradiente. Da mesma forma, o pré-condicionamento é uma técnica comum na descida de gradiente e é transportado para algoritmos mais avançados. Comecemos com um caso especial simples.

11.3.1 Gradiente descendente em uma dimensão

A descida gradiente em uma dimensão é um excelente exemplo para explicar por que o algoritmo de descida gradiente pode reduzir o valor da função objetivo. Considere alguma função de valor real continuamente diferenciável $f : \mathbb{R} \rightarrow \mathbb{R}$. Usando uma expansão de Taylor (Section 18.3), obtemos que

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \tag{11.3.1}$$

Ou seja, na primeira aproximação $f(x + \epsilon)$ é dado pelo valor da função $f(x)$ e a primeira derivada $f'(x)$ em x. Não é irracional supor que, para pequenos ϵ movendo-se na direção do gradiente negativo, f diminuirá. Para manter as coisas simples, escolhemos um tamanho de passo fixo $\eta > 0$ e escolhemos $\epsilon = -\eta f'(x)$. Conectando isso à expansão Taylor acima, obtemos

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \tag{11.3.2}$$

[110] https://discuss.d2l.ai/t/350

Se a derivada $f'(x) \neq 0$ não desaparecer, fazemos progresso, pois $\eta f'^2(x) > 0$. Além disso, podemos sempre escolher η pequeno o suficiente para que os termos de ordem superior se tornem irrelevantes. Daí chegamos a

$$f(x - \eta f'(x)) \lessapprox f(x). \tag{11.3.3}$$

Isso significa que, se usarmos

$$x \leftarrow x - \eta f'(x) \tag{11.3.4}$$

para iterar x, o valor da função $f(x)$ pode diminuir. Portanto, na descida do gradiente, primeiro escolhemos um valor inicial x e uma constante $\eta > 0$ e, em seguida, os usamos para iterar continuamente x até que a condição de parada seja alcançada, por exemplo, quando a magnitude do gradiente $|f'(x)|$ é pequeno o suficiente ou o número de iterações atingiu um determinado valor.

Para simplificar, escolhemos a função objetivo $f(x) = x^2$ para ilustrar como implementar a descida gradiente. Embora saibamos que $x = 0$ é a solução para minimizar $f(x)$, ainda usamos esta função simples para observar como x muda. Como sempre, começamos importando todos os módulos necessários.

```
%matplotlib inline
import numpy as np
import torch
from d2l import torch as d2l
```

```
f = lambda x: x**2  # Objective function
gradf = lambda x: 2 * x  # Its derivative
```

Em seguida, usamos $x = 10$ como o valor inicial e assumimos $\eta = 0,2$. Usando a descida gradiente para iterar x por 10 vezes, podemos ver que, eventualmente, o valor de x se aproxima da solução ótima.

```
def gd(eta):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * gradf(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

res = gd(0.2)
```

```
epoch 10, x: 0.06046617599999997
```

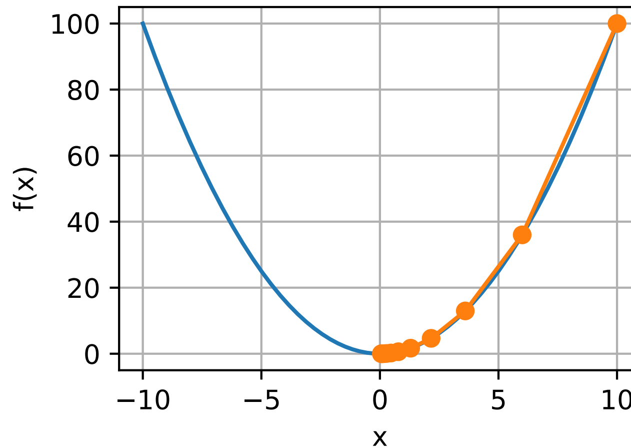
O progresso da otimização em x pode ser traçado da seguinte maneira.

```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, res], [[f(x) for x in f_line], [f(x) for x in res]],
```

(continues on next page)

(continued from previous page)

```
            'x', 'f(x)', fmts=['-', '-o'])

show_trace(res)
```

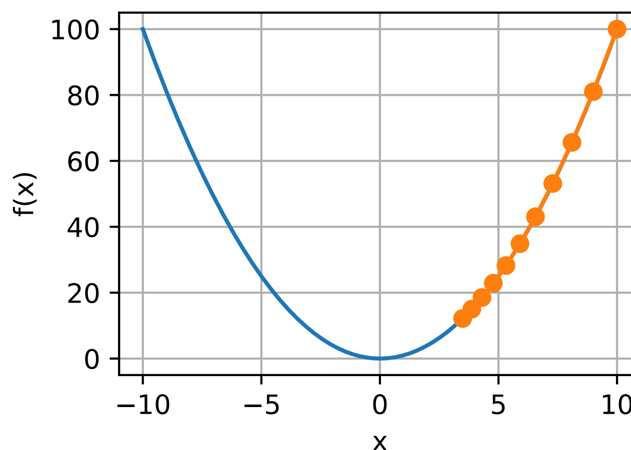


Taxa de Aprendizagem

A taxa de aprendizagem η pode ser definida pelo designer do algoritmo. Se usarmos uma taxa de aprendizado muito pequena, a atualização de x será muito lenta, exigindo mais iterações para obter uma solução melhor. Para mostrar o que acontece nesse caso, considere o progresso no mesmo problema de otimização para $\eta = 0,05$. Como podemos ver, mesmo após 10 passos ainda estamos muito longe da solução ótima.

```
show_trace(gd(0.05))
```

```
epoch 10, x: 3.4867844009999995
```

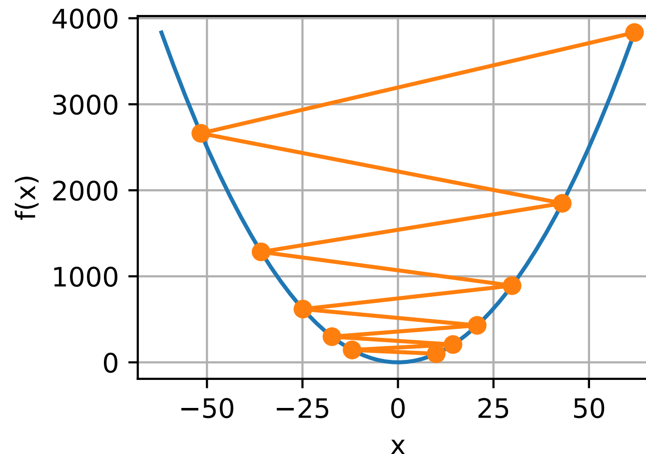


Por outro lado, se usarmos uma taxa de aprendizado excessivamente alta, $|\eta f'(x)|$ pode ser muito grande para a fórmula de expansão de Taylor de primeira ordem. Ou seja, o termo $\mathcal{O}(\eta^2 f'^2(x))$ in (11.3.1) pode se tornar significativo. Nesse caso, não podemos garantir que a iteração de x será

capaz de diminuir o valor de $f(x)$. Por exemplo, quando definimos a taxa de aprendizagem para $\eta = 1.1$, x ultrapassa a solução ótima $x = 0$ e diverge gradualmente.

```
show_trace(gd(1.1))
```

```
epoch 10, x: 61.917364224000096
```

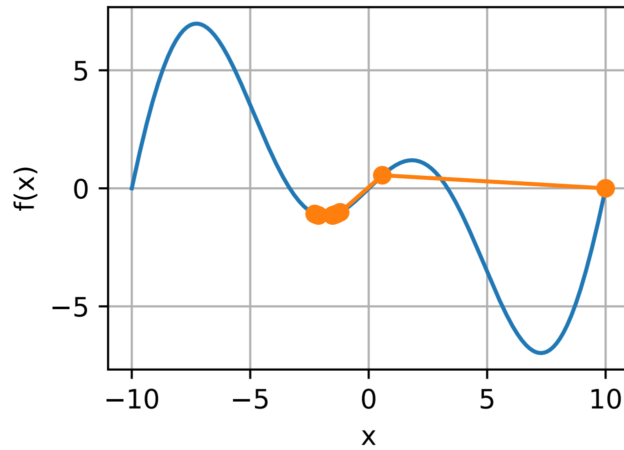


Minimo Local

Para ilustrar o que acontece com funções não convexas, considere o caso de $f(x) = x \cdot \cos cx$. Esta função possui infinitos mínimos locais. Dependendo de nossa escolha da taxa de aprendizado e de quão bem condicionado o problema está, podemos chegar a uma de muitas soluções. O exemplo abaixo ilustra como uma alta taxa de aprendizado (irrealista) levará a um mínimo local insatisfatório.

```
c = torch.tensor(0.15 * np.pi)
f = lambda x: x * torch.cos(c * x)
gradf = lambda x: torch.cos(c * x) - c * x * torch.sin(c * x)
show_trace(gd(2))
```

```
epoch 10, x: tensor(-1.5282)
```



11.3.2 Gradiente descendente multivariado

Agora que temos uma melhor intuição do caso univariado, consideremos a situação em que $\mathbf{x} \in \mathbb{R}^d$. Ou seja, a função objetivo $f : \mathbb{R}^d \to \mathbb{R}$ mapeia vetores em escalares. Correspondentemente, seu gradiente também é multivariado. É um vetor que consiste em d derivadas parciais:

$$\nabla f(\mathbf{x}) = \bigg[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\bigg]^\top. \tag{11.3.5}$$

Cada elemento derivado parcial $\partial f(\mathbf{x})/\partial x_i$ no gradiente indica a taxa de variação de f em $\mathbf{x}$ em relação à entrada x_i. Como antes, no caso univariado, podemos usar a aproximação de Taylor correspondente para funções multivariadas para ter uma ideia do que devemos fazer. Em particular, temos que

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\epsilon\|^2). \tag{11.3.6}$$

Em outras palavras, até os termos de segunda ordem em ϵ, a direção da descida mais acentuada é dada pelo gradiente negativo $-\nabla f(\mathbf{x})$. A escolha de uma taxa de aprendizagem adequada $\eta > 0$ produz o algoritmo de descida gradiente prototípico:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \tag{11.3.7}$$

Para ver como o algoritmo se comporta na prática, vamos construir uma função objetivo $f(\mathbf{x}) = x_1^2 + 2x_2^2$ com um vetor bidimensional $\mathbf{x} = [x_1, x_2]^\top$ como entrada e um escalar como saída. O gradiente é dado por $ nabla f (mathbf {x}) = [2x_1, 4x_2] ^ top $. Observaremos a trajetória de $\mathbf{x}$ por gradiente descendente a partir da posição inicial $[-5, -2]$. Precisamos de mais duas funções auxiliares. O primeiro usa uma função de atualização e aplica $ 20 $ vezes ao valor inicial. O segundo auxiliar visualiza a trajetória de $\mathbf{x}$.

```
def train_2d(trainer, steps=20):  #@save
    """Optimize a 2-dim objective function with a customized trainer."""
    # s1 and s2 are internal state variables and will
    # be used later in the chapter
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
```

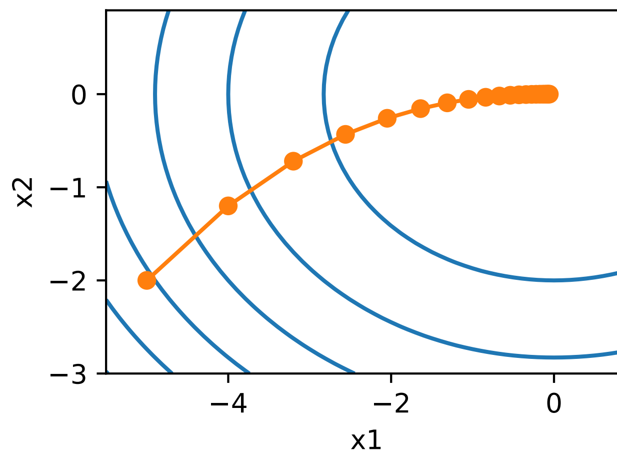
(continues on next page)

(continued from previous page)

```
        results.append((x1, x2))
    return results

def show_trace_2d(f, results):  #@save
    """Show the trace of 2D variables during optimization."""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = torch.meshgrid(torch.arange(-5.5, 1.0, 0.1),
                          torch.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

A seguir, observamos a trajetória da variável de otimização $\mathbf{x}$ para a taxa de aprendizagem $\eta = 0.1$. Podemos ver que após 20 passos o valor de $\mathbf{x}$ se aproxima de seu mínimo em $[0, 0]$. O progresso é bastante bem comportado, embora bastante lento.

```
f = lambda x1, x2: x1 ** 2 + 2 * x2 ** 2  # Objective
gradf = lambda x1, x2: (2 * x1, 4 * x2)  # Gradient

def gd(x1, x2, s1, s2):
    (g1, g2) = gradf(x1, x2)  # Compute gradient
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)  # Update variables

eta = 0.1
show_trace_2d(f, train_2d(gd))
```



11.3.3 Métodos Adaptativos

Como pudemos ver em Section 11.3.1, obter a taxa de aprendizado η "na medida certa" é complicado. Se o pegarmos muito pequeno, não faremos progresso. Se o escolhermos muito grande, a solução oscila e, no pior dos casos, pode até divergir. E se pudéssemos determinar η automaticamente ou nos livrarmos da necessidade de selecionar um tamanho de passo? Métodos de segunda ordem que olham não apenas para o valor e gradiente da objetiva, mas também para sua *curvatura* podem ajudar neste caso. Embora esses métodos não possam ser aplicados ao aprendizado profundo diretamente devido ao custo computacional, eles fornecem intuição útil sobre

como projetar algoritmos de otimização avançados que imitam muitas das propriedades desejáveis dos algoritmos descritos abaixo.

Método de Newton

Revendo a expansão de Taylor de f, não há necessidade de parar após o primeiro mandato. Na verdade, podemos escrever como

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + \mathcal{O}(\|\epsilon\|^3). \tag{11.3.8}$$

Para evitar notações complicadas, definimos $H_f := \nabla\nabla^\top f(\mathbf{x})$ como o *Hessiano* de f. Esta é uma matriz $d \times d$. Para d pequenos e problemas simples, H_f é fácil de calcular. Para redes profundas, por outro lado, H_f pode ser proibitivamente grande, devido ao custo de armazenamento de $\mathcal{O}(d^2)$ entradas. Além disso, pode ser muito caro computar por retropropagação, pois precisaríamos aplicar a retropropagação ao gráfico de chamada de retropropagação. Por enquanto, vamos ignorar essas considerações e ver que algoritmo obteríamos.

Afinal, o mínimo de f satisfaz $\nabla f(\mathbf{x}) = 0$. Pegando derivados de (11.3.8) em relação a ϵ e ignorando termos de ordem superior, chegamos a

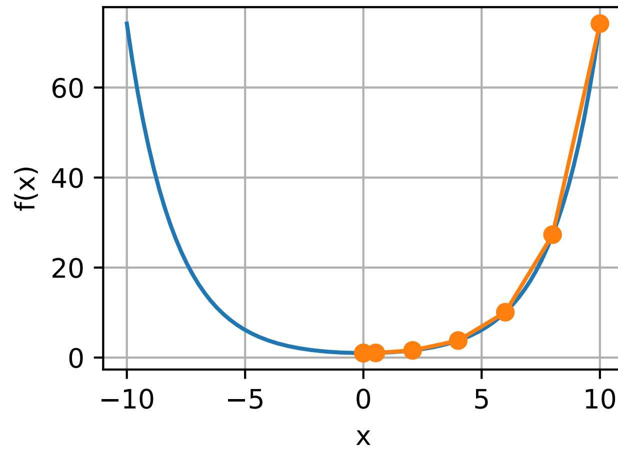
$$\nabla f(\mathbf{x}) + H_f\epsilon = 0 \text{ and hence } \epsilon = -H_f^{-1}\nabla f(\mathbf{x}). \tag{11.3.9}$$

Ou seja, precisamos inverter o Hessian H_f como parte do problema de otimização.

Para $f(x) = \frac{1}{2}x^2$ temos $\nabla f(x) = x$ e $H_f = 1$. Portanto, para qualquer x obtemos $\epsilon = -x$. Em outras palavras, um único passo é suficiente para convergir perfeitamente sem a necessidade de nenhum ajuste! Infelizmente, tivemos um pouco de sorte aqui, já que a expansão de Taylor foi exata. Vamos ver o que acontece em outros problemas.

```
c = torch.tensor(0.5)
f = lambda x: torch.cosh(c * x)  # Objective
gradf = lambda x: c * torch.sinh(c * x)  # Derivative
hessf = lambda x: c**2 * torch.cosh(c * x)  # Hessian

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * gradf(x) / hessf(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton())
```

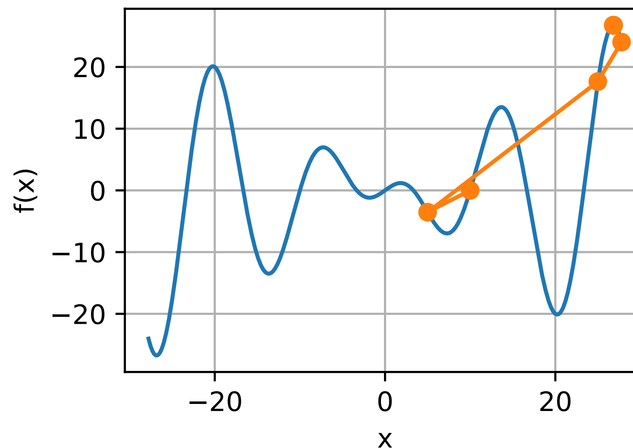
```
epoch 10, x: tensor(0.)
```



Agora vamos ver o que acontece quando temos uma função *não convexa*, como $f(x) = x\cos(cx)$. Afinal, observe que no método de Newton acabamos nos dividindo pelo Hessiano. Isso significa que se a segunda derivada for *negativa*, caminharíamos na direção de *aumentar* f. Essa é uma falha fatal do algoritmo. Vamos ver o que acontece na prática.

```
c = torch.tensor(0.15 * np.pi)
f = lambda x: x * torch.cos(c * x)
gradf = lambda x: torch.cos(c * x) - c * x * torch.sin(c * x)
hessf = lambda x: - 2 * c * torch.sin(c * x) - x * c**2 * torch.cos(c * x)

show_trace(newton())
```

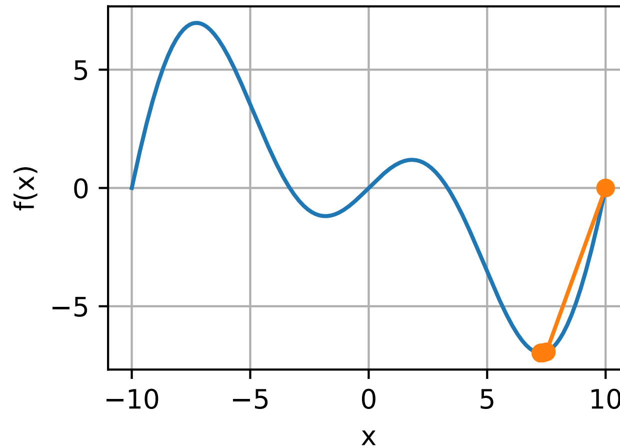
```
epoch 10, x: tensor(26.8341)
```



Isso deu espetacularmente errado. Como podemos arranjá-lo? Uma maneira seria "consertar" o Hessian tomando seu valor absoluto. Outra estratégia é trazer de volta a taxa de aprendizado. Isso parece anular o propósito, mas não totalmente. Ter informações de segunda ordem permite-nos ser cautelosos sempre que a curvatura for grande e dar passos mais longos sempre que a objetiva for plana. Vamos ver como isso funciona com uma taxa de aprendizado um pouco menor, digamos $\eta = 0.5$. Como podemos ver, temos um algoritmo bastante eficiente.

```
show_trace(newton(0.5))
```

```
epoch 10, x: tensor(7.2699)
```



Análise de Convergência

Analisamos apenas a taxa de convergência para f convexa e três vezes diferenciável, onde em seu mínimo x^* a segunda derivada é diferente de zero, ou seja, onde $f''(x^*) > 0$. A prova multivariada é uma extensão direta do argumento abaixo e omitida por não nos ajudar muito em termos de intuição.

Denote por x_k o valor de x na k-ésima iteração e seja $e_k := x_k - x^*$ a distância da otimização. Pela expansão da série de Taylor, temos que a condição $f'(x^*) = 0$ pode ser escrita como

$$0 = f'(x_k - e_k) = f'(x_k) - e_k f''(x_k) + \frac{1}{2} e_k^2 f'''(\xi_k). \tag{11.3.10}$$

Isso vale para alguns $\xi_k \in [x_k - e_k, x_k]$. Lembre-se de que temos a atualização $x_{k+1} = x_k - f'(x_k)/f''(x_k)$. Dividindo a expansão acima por $f''(x_k)$, obtemos

$$e_k - f'(x_k)/f''(x_k) = \frac{1}{2} e_k^2 f'''(\xi_k)/f''(x_k). \tag{11.3.11}$$

Conectar as equações de atualização leva ao seguinte limite $e_{k+1} \leq e_k^2 f'''(\xi_k)/f'(x_k)$. Consequentemente, sempre que estamos em uma região de $f'''(\xi_k)/f''(x_k) \leq c$ limitada, temos um erro quadraticamente decrescente $e_{k+1} \leq c e_k^2$.

À parte, os pesquisadores de otimização chamam isso de convergência *linear*, enquanto uma condição como $e_{k+1} \leq \alpha e_k$ seria chamada de taxa de convergência *constante*. Observe que esta análise vem com uma série de advertências: Não temos muita garantia de quando alcançaremos a região de convergência rápida. Em vez disso, sabemos apenas que, uma vez que o alcance, a convergência será muito rápida. Em segundo lugar, isso requer que f seja bem comportado com os derivados de ordem superior. Tudo se resume a garantir que f não tenha nenhuma propriedade "surpreendente" em termos de como ele pode alterar seus valores.

Precondicionamento

Sem surpresa, computar e armazenar o Hessian completo é muito caro. Portanto, é desejável encontrar alternativas. Uma maneira de melhorar as coisas é evitar calcular o Hessian em sua totalidade, mas apenas calcular as entradas * diagonais *. Embora isso não seja tão bom quanto o método de Newton completo, ainda é muito melhor do que não usá-lo. Além disso, as estimativas para os principais elementos diagonais são o que impulsiona algumas das inovações em algoritmos de otimização de descida de gradiente estocástico. Isso leva à atualização de algoritmos do formulário

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathrm{diag}(H_f)^{-1} \nabla f(\mathbf{x}). \tag{11.3.12}$$

Para ver por que isso pode ser uma boa ideia, considere uma situação em que uma variável denota a altura em milímetros e a outra denota a altura em quilômetros. Assumindo que para ambas a escala natural está em metros, temos um péssimo desencontro nas parametrizações. Usar o pré-condicionamento remove isso. O pré-condicionamento eficaz com gradiente descendente equivale a selecionar uma taxa de aprendizagem diferente para cada coordenada.

Gradiente descendente com pesquisa de linha

Um dos principais problemas na descida gradiente era que poderíamos ultrapassar a meta ou fazer um progresso insuficiente. Uma solução simples para o problema é usar a pesquisa de linha em conjunto com a descida gradiente. Ou seja, usamos a direção dada por $\nabla f(\mathbf{x})$ e, em seguida, realizamos a pesquisa binária para saber qual o comprimento do passo η minimiza $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$.

Este algoritmo converge rapidamente (para uma análise e prova veja, por exemplo, (Boyd & Vandenberghe, 2004)). No entanto, para fins de aprendizado profundo, isso não é tão viável, uma vez que cada etapa da pesquisa de linha exigiria que avaliássemos a função objetivo em todo o conjunto de dados. Isso é muito caro de se realizar.

11.3.4 Sumário

- As taxas de aprendizagem são importantes. Muito grande e divergimos, muito pequeno e não avançamos.
- A descida do gradiente pode ficar presa em mínimos locais.
- Em dimensões altas, ajustar a taxa de aprendizagem é complicado.
- O pré-condicionamento pode ajudar no ajuste da escala.
- O método de Newton é muito mais rápido *uma vez* que começou a funcionar corretamente em problemas convexos.
- Cuidado ao usar o método de Newton sem quaisquer ajustes para problemas não convexos.

11.3.5 Exercícios

1. Experimente diferentes taxas de aprendizagem e funções objetivas para a descida do gradiente.
2. Implemente a pesquisa de linha para minimizar uma função convexa no intervalo $[a, b]$.
   - Você precisa de derivadas para pesquisa binária, ou seja, para decidir se deve escolher $[a, (a + b)/2]$ ou $[(a + b)/2, b]$.
   - Quão rápida é a taxa de convergência do algoritmo?
   - Implemente o algoritmo e aplique-o para minimizar $\log(\exp(x) + \exp(-2 * x - 3))$.
3. Projete uma função objetivo definida em $\mathbb{R}^2$ onde a descida do gradiente é excessivamente lenta. Dica: dimensione coordenadas diferentes de forma diferente.
4. Implemente a versão leve do método de Newton usando o pré-condicionamento:
   - Use diagonal Hessian como pré-condicionador.
   - Use os valores absolutos disso em vez dos valores reais (possivelmente com sinal).
   - Aplique isso ao problema acima.
5. Aplique o algoritmo acima a uma série de funções objetivo (convexas ou não). O que acontece se você girar as coordenadas em 45 graus?

Discussão[111]

11.4 Gradiente Descendente Estocástico

Nesta seção, apresentaremos os princípios básicos da gradiente descendente estocástico.

```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

11.4.1 Atualizações de gradiente estocástico

No aprendizado profundo, a função objetivo é geralmente a média das funções de perda para cada exemplo no conjunto de dados de treinamento. Assumimos que $f_i(\mathbf{x})$ é a função de perda do conjunto de dados de treinamento com n exemplos, um índice de i e um vetor de parâmetro de $\mathbf{x}$, então temos o função objetiva

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{x}). \tag{11.4.1}$$

O gradiente da função objetivo em $\mathbf{x}$ é calculado como

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}). \tag{11.4.2}$$

[111] https://discuss.d2l.ai/t/351

Se o gradiente descendente for usado, o custo de computação para cada iteração de variável independente é $\mathcal{O}(n)$, que cresce linearmente com n. Portanto, quando o conjunto de dados de treinamento do modelo é grande, o custo da descida do gradiente para cada iteração será muito alto.

A descida gradiente estocástica (SGD) reduz o custo computacional a cada iteração. A cada iteração de descida do gradiente estocástico, amostramos uniformemente um índice $i \in \{1, \ldots, n\}$ para exemplos de dados aleatoriamente e calculamos o gradiente $\nabla f_i(\mathbf{x})$ para atualizar **x**:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}). \tag{11.4.3}$$

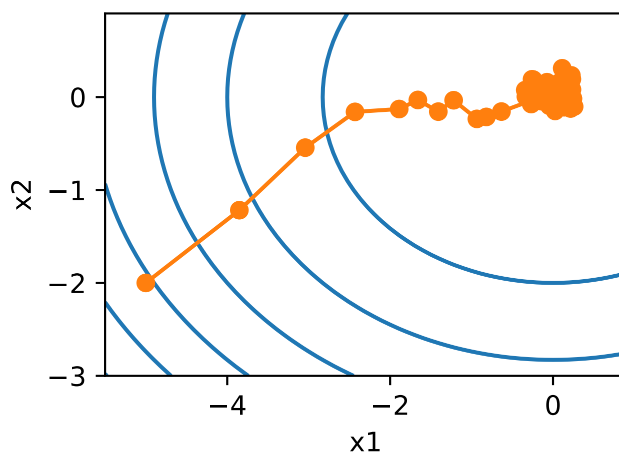
Aqui, η é a taxa de aprendizado. Podemos ver que o custo de computação para cada iteração cai de $\mathcal{O}(n)$ da descida do gradiente para a constante $\mathcal{O}(1)$. Devemos mencionar que o gradiente estocástico $\nabla f_i(\mathbf{x})$ é a estimativa imparcial do gradiente $\nabla f(\mathbf{x})$.

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{11.4.4}$$

Isso significa que, em média, o gradiente estocástico é uma boa estimativa do gradiente.

Agora, vamos compará-lo com a descida do gradiente adicionando ruído aleatório com uma média de 0 e uma variância de 1 ao gradiente para simular um SGD.

```
f = lambda x1, x2: x1 ** 2 + 2 * x2 ** 2  # Objective
gradf = lambda x1, x2: (2 * x1, 4 * x2)  # Gradient

def sgd(x1, x2, s1, s2):
    global lr  # Learning rate scheduler
    (g1, g2) = gradf(x1, x2)
    # Simulate noisy gradient
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))
    eta_t = eta * lr()  # Learning rate at time t
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)  # Update variables

eta = 0.1
lr = (lambda: 1)  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```



Como podemos ver, a trajetória das variáveis no SGD é muito mais ruidosa do que a que observamos na descida do gradiente na seção anterior. Isso se deve à natureza estocástica do gradiente. Ou seja, mesmo quando chegamos perto do mínimo, ainda estamos sujeitos à incerteza injetada pelo gradiente instantâneo via $\eta \nabla f_i(\mathbf{x})$. Mesmo após 50 passos, a qualidade ainda não é tão boa. Pior ainda, não melhorará após etapas adicionais (encorajamos o leitor a experimentar um número maior de etapas para confirmar isso por conta própria). Isso nos deixa com a única alternativa — alterar a taxa de aprendizagem η. No entanto, se escolhermos isso muito pequeno, não faremos nenhum progresso significativo inicialmente. Por outro lado, se o escolhermos muito grande, não obteremos uma boa solução, como visto acima. A única maneira de resolver esses objetivos conflitantes é reduzir a taxa de aprendizado *dinamicamente* à medida que a otimização avança.

Esta também é a razão para adicionar uma função de taxa de aprendizagem `lr` na função de passo`sgd`. No exemplo acima, qualquer funcionalidade para o agendamento da taxa de aprendizagem permanece latente, pois definimos a função `lr` associada como constante, ou seja,`lr = (lambda: 1)`.

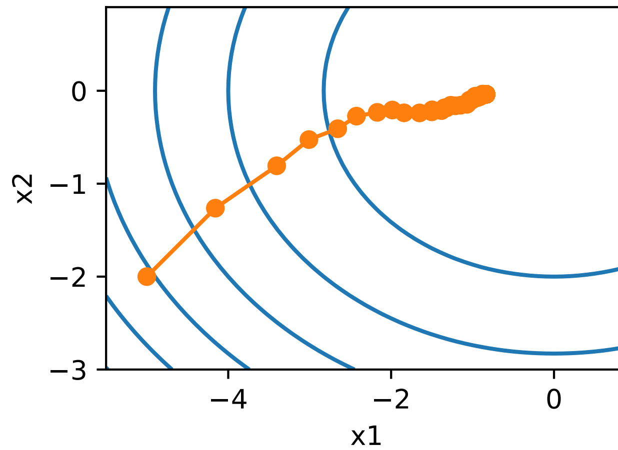
11.4.2 Taxa de aprendizagem dinâmica

Substituir η por uma taxa de aprendizado dependente do tempo $\eta(t)$ aumenta a complexidade de controlar a convergência de um algoritmo de otimização. Em particular, é preciso descobrir com que rapidez η deve decair. Se for muito rápido, pararemos de otimizar prematuramente. Se diminuirmos muito lentamente, perderemos muito tempo com a otimização. Existem algumas estratégias básicas que são usadas no ajuste de η ao longo do tempo (discutiremos estratégias mais avançadas em um capítulo posterior):

$$\begin{aligned}
\eta(t) & = \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{piecewise constant} \\
\eta(t) & = \eta_0 \cdot e^{-\lambda t} && \text{exponential} \\
\eta(t) & = \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{polynomial}
\end{aligned} \tag{11.4.5}$$

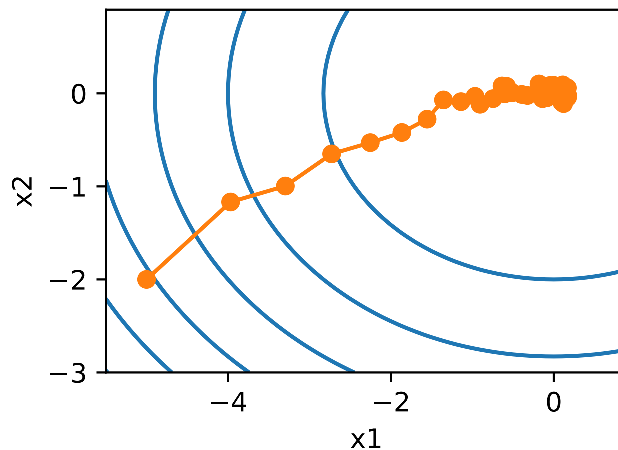
No primeiro cenário, diminuímos a taxa de aprendizado, por exemplo, sempre que o progresso na otimização para. Esta é uma estratégia comum para treinar redes profundas. Alternativamente, poderíamos diminuí-lo de forma muito mais agressiva por uma redução exponencial. Infelizmente, isso leva a uma parada prematura antes que o algoritmo tenha convergido. Uma escolha popular é o decaimento polinomial com $\alpha = 0.5$. No caso da otimização convexa, há uma série de provas que mostram que essa taxa é bem comportada. Vamos ver como isso se parece na prática.

```
def exponential():
    global ctr
    ctr += 1
    return math.exp(-0.1 * ctr)

ctr = 1
lr = exponential  # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000))
```



Como esperado, a variância nos parâmetros é reduzida significativamente. No entanto, isso ocorre às custas de não convergir para a solução ótima $\mathbf{x} = (0, 0)$. Mesmo depois de 1000 passos, ainda estamos muito longe da solução ideal. Na verdade, o algoritmo não consegue convergir. Por outro lado, se usarmos um decaimento polinomial onde a taxa de aprendizagem decai com a raiz quadrada inversa do número de passos, a convergência é boa.

```
def polynomial():
    global ctr
    ctr += 1
    return (1 + 0.1 * ctr)**(-0.5)

ctr = 1
lr = polynomial  # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```



Existem muitas outras opções de como definir a taxa de aprendizagem. Por exemplo, poderíamos começar com uma taxa pequena, aumentar rapidamente e diminuí-la novamente, embora mais lentamente. Poderíamos até alternar entre taxas de aprendizagem menores e maiores. Existe uma grande variedade de tais horários. Por enquanto, vamos nos concentrar em tabelas de taxas de aprendizagem para as quais uma análise teórica abrangente é possível, ou seja, em taxas de aprendizagem em um cenário convexo. Para problemas não-convexos gerais é muito difícil obter garantias de convergência significativas, uma vez que, em geral, minimizar problemas não-convexos

não lineares é NP difícil. Para uma pesquisa, consulte, por exemplo, as excelentes notas de aula[112] de Tibshirani 2015.

11.4.3 Análise de convergência para objetivos convexos

O que segue é opcional e serve principalmente para transmitir mais intuição sobre o problema. Nos limitamos a uma das provas mais simples, conforme descrito por (Nesterov & Vial, 2000). Existem técnicas de prova significativamente mais avançadas, por exemplo, sempre que a função objetiva é particularmente bem comportada. (Hazan et al., 2008) mostra que para funções fortemente convexas, ou seja, para funções que podem ser limitadas de baixo por $\mathbf{x}^\top \mathbf{Q} \mathbf{x}$, é possível minimizá-los em um pequeno número de etapas enquanto diminui a taxa de aprendizagem como $\eta(t) = \eta_0/(\beta t + 1)$. Infelizmente, esse caso nunca ocorre realmente no aprendizado profundo e ficamos com uma taxa de diminuição muito mais lenta na prática.

Considere o caso onde

$$\mathbf{w}_{t+1} = \mathbf{w}_{t} - \eta_t \partial_\mathbf{w} l(\mathbf{x}_t, \mathbf{w}). \tag{11.4.6}$$

Em particular, assuma que $\mathbf{x}_t$ é retirado de alguma distribuição $P(\mathbf{x})$ e que $l(\mathbf{x}, \mathbf{w})$ é uma função convexa em $\mathbf{w}$ para todos os $\mathbf{x}$. Última denotada por

$$R(\mathbf{w}) = E_{\mathbf{x} \sim P}[l(\mathbf{x}, \mathbf{w})] \tag{11.4.7}$$

o risco esperado e por R^* seu mínimo em relação a $\mathbf{w}$. Por último, seja $\mathbf{w}^*$ o minimizador (assumimos que ele existe dentro do domínio que $\mathbf{w}$ está definido). Neste caso, podemos rastrear a distância entre o parâmetro atual $\mathbf{w}_t$ e o minimizador de risco $\mathbf{w}^*$ e ver se melhora com o tempo:

$$\begin{aligned}\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 & = \|\mathbf{w}_{t} - \eta_t \partial_\mathbf{w} l(\mathbf{x}_t, \mathbf{w}) - \mathbf{w}^*\|^2 \\ & = \|\mathbf{w}_{t} - \mathbf{w}^*\|^2 + \eta_t^2 \|\partial_\mathbf{w} l(\mathbf{x}_t, \mathbf{w})\|^2 - 2 \eta_t \left\langle \mathbf{w}_t - \mathbf{w}^*, \partial_\mathbf{w} l(\mathbf{x}_t, \mathbf{w})\right\rangle. \end{aligned} \tag{11.4.8}$$

O gradiente $\partial_\mathbf{w} l(\mathbf{x}_t, \mathbf{w})$ pode ser limitado de cima por alguma constante de Lipschitz L, portanto, temos que

$$\eta_t^2 \|\partial_\mathbf{w} l(\mathbf{x}_t, \mathbf{w})\|^2 \leq \eta_t^2 L^2. \tag{11.4.9}$$

Estamos principalmente interessados em como a distância entre $\mathbf{w}_t$ e $\mathbf{w}^*$ muda *na expectativa*. Na verdade, para qualquer sequência específica de passos, a distância pode muito bem aumentar, dependendo de qualquer $\mathbf{x}_t$ que encontrarmos. Portanto, precisamos limitar o produto interno. Por convexidade temos que

$$l(\mathbf{x}_t, \mathbf{w}^*) \geq l(\mathbf{x}_t, \mathbf{w}_t) + \left\langle \mathbf{w}^* - \mathbf{w}_t, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}_t) \right\rangle. \tag{11.4.10}$$

Usando ambas as desigualdades e conectando-as ao acima, obtemos um limite para a distância entre os parâmetros no tempo $t+1$ da seguinte forma:

$$\|\mathbf{w}_{t} - \mathbf{w}^*\|^2 - \|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 \geq 2 \eta_t (l(\mathbf{x}_t, \mathbf{w}_t) - l(\mathbf{x}_t, \mathbf{w}^*)) - \eta_t^2 L^2. \tag{11.4.11}$$

Isso significa que progredimos enquanto a diferença esperada entre a perda atual e a perda ótima supera $\eta_t L^2$. Como o primeiro tende a convergir para 0, segue-se que a taxa de aprendizado η_t também precisa desaparecer.

[112] https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf

Em seguida, consideramos as expectativas sobre essa expressão. Isso produz

$$E_{\mathbf{w}_t}\left[\|\mathbf{w}_t-\mathbf{w}^*\|^2\right]-E_{\mathbf{w}_{t+1}|\mathbf{w}_t}\left[\|\mathbf{w}_{t+1}-\mathbf{w}^*\|^2\right]\geq 2\eta_t[E[R[\mathbf{w}_t]]-R^*]-\eta_t^2L^2. \tag{11.4.12}$$

A última etapa envolve a soma das desigualdades para $t\in\{t,\ldots,T\}$. Uma vez que a soma dos telescópios e diminuindo o termo inferior, obtemos

$$\|\mathbf{w}_0-\mathbf{w}^*\|^2\geq 2\sum_{t=1}^{T}\eta_t[E[R[\mathbf{w}_t]]-R^*]-L^2\sum_{t=1}^{T}\eta_t^2. \tag{11.4.13}$$

Observe que exploramos que $\mathbf{w}_0$ é dado e, portanto, a expectativa pode ser descartada. Última definição

$$\bar{\mathbf{w}}:=\frac{\sum_{t=1}^{T}\eta_t\mathbf{w}_t}{\sum_{t=1}^{T}\eta_t}. \tag{11.4.14}$$

Então, por convexidade, segue-se que

$$\sum_t\eta_tE[R[\mathbf{w}_t]]\geq\sum\eta_t\cdot\left[E[\bar{\mathbf{w}}]\right]. \tag{11.4.15}$$

Conectar isso à desigualdade acima produz o limite

$$\left[E[\bar{\mathbf{w}}]\right]-R^*\leq\frac{r^2+L^2\sum_{t=1}^{T}\eta_t^2}{2\sum_{t=1}^{T}\eta_t}. \tag{11.4.16}$$

Aqui $r^2:=\|\mathbf{w}_0-\mathbf{w}^*\|^2$ é um limite na distância entre a escolha inicial dos parâmetros e o resultado final. Em suma, a velocidade de convergência depende de quão rapidamente a função de perda muda por meio da constante de Lipschitz L e quão longe da otimização o valor inicial está r. Observe que o limite é em termos de $\bar{\mathbf{w}}$ em vez de $\mathbf{w}_T$. Este é o caso, pois $\bar{\mathbf{w}}$ é uma versão suavizada do caminho de otimização. Agora vamos analisar algumas opções para η_t.

- **Horizonte de tempo conhecido**. Sempre que r, L e T são conhecidos, podemos escolher \$$\eta=r/L\sqrt{T}$. Isso resulta no limite superior $rL(1+1/T)/2\sqrt{T}<rL/\sqrt{T}$. Ou seja, convergimos com a taxa $\mathcal{O}(1/\sqrt{T})$ para a solução ótima.
- **Horizonte de tempo desconhecido**. Sempre que quisermos ter uma boa solução para *a qualquer* momento T, podemos escolher $\eta=\mathcal{O}(1/\sqrt{T})$. Isso nos custa um fator logarítmico extra e leva a um limite superior da forma $\mathcal{O}(\log T/\sqrt{T})$.

Observe que para perdas fortemente convexas $l(\mathbf{x},\mathbf{w}')\geq l(\mathbf{x},\mathbf{w})+\langle\mathbf{w}'-\mathbf{w},\partial_{\mathbf{w}}l(\mathbf{x},\mathbf{w})\rangle+\frac{\lambda}{2}\|\mathbf{w}-\mathbf{w}'\|^2$ podemos projetar agendas de otimização convergentes ainda mais rapidamente. Na verdade, um declínio exponencial em η leva a um limite da forma $\mathcal{O}(\log T/T)$.

11.4.4 Gradientes estocásticos e amostras finitas

Até agora, fomos um pouco rápidos e soltos quando se trata de falar sobre a descida gradiente estocástica. Postulamos que desenhamos instâncias x_i, normalmente com rótulos y_i de alguma distribuição $p(x,y)$ e que usamos isso para atualizar os pesos w de alguma maneira. Em particular, para um tamanho de amostra finito, simplesmente argumentamos que a distribuição discreta $p(x,y)=\frac{1}{n}\sum_{i=1}^{n}\delta_{x_i}(x)\delta_{y_i}(y)$ nos permite realizar SGD sobre ele.

No entanto, não foi exatamente isso que fizemos. Nos exemplos de brinquedos na seção atual, simplesmente adicionamos ruído a um gradiente não estocástico, ou seja, fingimos ter pares

(x_i, y_i). Acontece que isso se justifica aqui (veja os exercícios para uma discussão detalhada). Mais preocupante é que em todas as discussões anteriores claramente não fizemos isso. Em vez disso, iteramos em todas as instâncias exatamente uma vez. Para ver por que isso é preferível, considere o inverso, ou seja, estamos amostrando n observações da distribuição discreta com substituição. A probabilidade de escolher um elemento i aleatoriamente é N^{-1}. Portanto, escolher pelo menos uma vez é

$$P(\text{choose } i) = 1 - P(\text{omit } i) = 1 - (1 - N^{-1})^N \approx 1 - e^{-1} \approx 0.63. \tag{11.4.17}$$

Um raciocínio semelhante mostra que a probabilidade de escolher uma amostra exatamente uma vez é dada por $\binom{N}{1} N^{-1}(1 - N^{-1})^{N-1} = \frac{N-1}{N}(1 - N^{-1})^N \approx e^{-1} \approx 0.37$. Isso leva a um aumento da variância e diminuição da eficiência dos dados em relação à amostragem sem reposição. Portanto, na prática, fazemos o último (e esta é a escolha padrão em todo este livro). Por último, observe que passagens repetidas pelo conjunto de dados percorrem-no em uma ordem aleatória *diferente*.

11.4.5 Sumário

- Para problemas convexos, podemos provar que, para uma ampla escolha de taxas de aprendizado, a Descida de Gradiente Estocástico convergirá para a solução ótima.
- Geralmente, esse não é o caso para aprendizagem profunda. No entanto, a análise de problemas convexos nos dá uma visão útil sobre como abordar a otimização, nomeadamente para reduzir a taxa de aprendizagem progressivamente, embora não muito rapidamente.
- Os problemas ocorrem quando a taxa de aprendizagem é muito pequena ou muito grande. Na prática, uma taxa de aprendizado adequada geralmente é encontrada somente após vários experimentos.
- Quando há mais exemplos no conjunto de dados de treinamento, custa mais calcular cada iteração para a descida do gradiente, portanto, SGD é preferível nesses casos.
- As garantias de otimização para SGD em geral não estão disponíveis em casos não convexos, pois o número de mínimos locais que requerem verificação pode ser exponencial.

11.4.6 Exercícios

1. Experimente diferentes programações de taxa de aprendizagem para SGD e com diferentes números de iterações. Em particular, plote a distância da solução ótima $(0, 0)$ como uma função do número de iterações.
2. Prove que para a função $f(x_1, x_2) = x_1^2 + 2x_2^2$ adicionar ruído normal ao gradiente é equivalente a minimizar uma função de perda $l(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ onde x é extraído de uma distribuição normal.
    - Derive a média e a variância da distribuição de $\mathbf{x}$.
    - Mostre que esta propriedade é geralmente válida para funções objetivo $f(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mu)^\top Q(\mathbf{x} - \mu)$ para $Q \succeq 0$.
3. Compare a convergência de SGD quando você faz a amostra de $\{(x_1, y_1), \ldots, (x_m, y_m)\}$ com substituição e quando você faz a amostra sem substituição.
4. Como você mudaria o solucionador SGD se algum gradiente (ou melhor, alguma coordenada associada a ele) fosse consistentemente maior do que todos os outros gradientes?

5. Suponha que $f(x) = x^2(1 + \sin x)$. Quantos mínimos locais f tem? Você pode alterar f de forma que, para minimizá-lo, seja necessário avaliar todos os mínimos locais?

Discussão[113]

11.5 Gradiente Estocástico Descendente Minibatch

Até agora, encontramos dois extremos na abordagem de aprendizagem baseada em gradiente: Section 11.3 usa o conjunto de dados completo para calcular gradientes e atualizar parâmetros, uma passagem de cada vez. Inversamente Section 11.4 processa uma observação por vez para fazer progresso. Cada um deles tem suas próprias desvantagens. O Gradient Descent não é particularmente *eficiente em dados* sempre que os dados são muito semelhantes. Stochastic Gradient Descent não é particularmente *computacionalmente eficiente*, uma vez que CPUs e GPUs não podem explorar todo o poder da vetorização. Isso sugere que pode haver um meio-termo feliz e, de fato, é isso que temos usado até agora nos exemplos que discutimos.

11.5.1 Vetorização e caches

No centro da decisão de usar minibatches está a eficiência computacional. Isso é mais facilmente compreendido quando se considera a paralelização para várias GPUs e vários servidores. Nesse caso, precisamos enviar pelo menos uma imagem para cada GPU. Com 8 GPUs por servidor e 16 servidores, já chegamos a um tamanho de minibatch de 128.

As coisas são um pouco mais sutis quando se trata de GPUs individuais ou até CPUs. Esses dispositivos têm vários tipos de memória, geralmente vários tipos de unidades de computação e diferentes restrições de largura de banda entre eles. Por exemplo, uma CPU tem um pequeno número de registradores e, em seguida, L1, L2 e, em alguns casos, até mesmo cache L3 (que é compartilhado entre os diferentes núcleos do processador). Esses caches têm tamanho e latência crescentes (e, ao mesmo tempo, largura de banda decrescente). Basta dizer que o processador é capaz de realizar muito mais operações do que a interface de memória principal é capaz de fornecer.

- Uma CPU de 2 GHz com 16 núcleos e vetorização AVX-512 pode processar até $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ bytes por segundo. A capacidade das GPUs facilmente excede esse número por um fator de 100. Por outro lado, um processador de servidor de médio porte pode não ter muito mais do que 100 GB/s de largura de banda, ou seja, menos de um décimo do que seria necessário para manter o processador alimentado. Para piorar a situação, nem todo acesso à memória é criado da mesma forma: primeiro, as interfaces de memória são normalmente de 64 bits ou mais largas (por exemplo, em GPUs de até 384 bits), portanto, a leitura de um único byte incorre no custo de um acesso muito mais amplo.
- Há uma sobrecarga significativa para o primeiro acesso, enquanto o acesso sequencial é relativamente barato (geralmente chamado de leitura intermitente). Há muito mais coisas para se manter em mente, como armazenamento em cache quando temos vários sockets, chips e outras estruturas. Uma discussão detalhada sobre isso está além do escopo desta seção. Veja, por exemplo, este artigo da Wikipedia[114] para uma discussão mais aprofundada.

A maneira de aliviar essas restrições é usar uma hierarquia de caches de CPU que são realmente rápidos o suficiente para fornecer dados ao processador. Esta é *a* força motriz por trás dos lotes no

[113] https://discuss.d2l.ai/t/497
[114] https://en.wikipedia.org/wiki/Cache_hierarchy

aprendizado profundo. Para manter as coisas simples, considere a multiplicação matriz-matriz, digamos $\mathbf{A} = \mathbf{B}\mathbf{C}$. Temos várias opções para calcular $\mathbf{A}$. Por exemplo, podemos tentar o seguinte:

1. Poderíamos calcular $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}^\top$, ou seja, poderíamos calculá-lo elemento a elemento por meio de produtos escalares.
2. Poderíamos calcular $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}_{:,j}^\top$, ou seja, poderíamos calcular uma coluna de cada vez . Da mesma forma, poderíamos calcular $\mathbf{A}$ uma linha $\mathbf{A}_{i,:}$ de cada vez.
3. Poderíamos simplesmente calcular $\mathbf{A} = \mathbf{B}\mathbf{C}$.
4. Poderíamos quebrar $\mathbf{B}$ e $\mathbf{C}$ em matrizes de blocos menores e calcular $\mathbf{A}$ um bloco de cada vez.

Se seguirmos a primeira opção, precisaremos copiar um vetor linha e uma coluna para a CPU cada vez que quisermos calcular um elemento $\mathbf{A}_{ij}$. Pior ainda, devido ao fato de que os elementos da matriz estão alinhados sequencialmente, somos obrigados a acessar muitas localizações disjuntas para um dos dois vetores à medida que os lemos da memória. A segunda opção é muito mais favorável. Nele, podemos manter o vetor coluna $\mathbf{C}_{:,j}$ no cache da CPU enquanto continuamos percorrendo B. Isso reduz pela metade o requisito de largura de banda de memória com acesso correspondentemente mais rápido. Claro, a opção 3 é a mais desejável. Infelizmente, a maioria das matrizes pode não caber inteiramente no cache (é isso que estamos discutindo, afinal). No entanto, a opção 4 oferece uma alternativa prática útil: podemos mover blocos da matriz para o cache e multiplicá-los localmente. Bibliotecas otimizadas cuidam disso para nós. Vejamos como essas operações são eficientes na prática.

Além da eficiência computacional, a sobrecarga introduzida pelo Python e pela própria estrutura de aprendizado profundo é considerável. Lembre-se de que cada vez que executamos um comando, o interpretador Python envia um comando para o mecanismo MXNet que precisa inseri-lo no gráfico computacional e lidar com ele durante o agendamento. Essa sobrecarga pode ser bastante prejudicial. Em suma, é altamente recomendável usar vetorização (e matrizes) sempre que possível.

```
%matplotlib inline
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l

timer = d2l.Timer()
A = torch.zeros(256, 256)
B = torch.randn(256, 256)
C = torch.randn(256, 256)
```

A atribuição elementar simplesmente itera sobre todas as linhas e colunas de $\mathbf{B}$ e $\mathbf{C}$respectivamente para atribuir o valor a $\mathbf{A}$.

```
# Compute A = BC one element at a time
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j] = torch.dot(B[i, :], C[:, j])
timer.stop()
```

```
1.0866479873657227
```

Uma estratégia mais rápida é realizar a atribuição em colunas.

```
# Compute A = BC one column at a time
timer.start()
for j in range(256):
    A[:, j] = torch.mv(B, C[:, j])
timer.stop()
```

```
0.009199380874633789
```

Por último, a maneira mais eficaz é realizar toda a operação em um bloco. Vejamos qual é a respectiva velocidade das operações.

```
# Compute A = BC in one go
timer.start()
A = torch.mm(B, C)
timer.stop()

# Multiply and add count as separate operations (fused in practice)
gigaflops = [2/i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

```
performance in Gigaflops: element 1.841, column 217.406, full 767.415
```

11.5.2 Minibatches

No passado, tínhamos como certo que leríamos *minibatches* de dados em vez de observações únicas para atualizar os parâmetros. Agora fornecemos uma breve justificativa para isso. O processamento de observações únicas exige que realizemos muitas multiplicações de vetor-matriz única (ou mesmo vetor-vetor), o que é bastante caro e incorre em uma sobrecarga significativa em nome da estrutura de aprendizado profundo subjacente. Isso se aplica tanto à avaliação de uma rede quando aplicada aos dados (geralmente chamada de inferência) quanto ao calcular gradientes para atualizar parâmetros. Ou seja, isso se aplica sempre que executamos$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ onde

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \tag{11.5.1}$$

Podemos aumentar a eficiência *computacional* dessa operação aplicando-a a um minibatch de observações por vez. Ou seja, substituímos o gradiente $\mathbf{g}_t$ em uma única observação por um em um pequeno lote

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \tag{11.5.2}$$

Vamos ver o que isso faz com as propriedades estatísticas de $\mathbf{g}_t$: uma vez que tanto $\mathbf{x}_t$ e também todos os elementos do minibatch $\mathcal{B}_t$ são desenhados uniformemente e aleatoriamente do conjunto de treinamento, a expectativa do gradiente permanece inalterada. A variância, por outro lado, é

reduzida significativamente. Como o gradiente de minibatch é composto de $b := |\mathcal{B}_t|$ gradientes independentes que estão sendo calculados, seu desvio padrão é reduzido por um fator de $b^{-\frac{1}{2}}$. Isso, por si só, é uma coisa boa, pois significa que as atualizações estão alinhadas de forma mais confiável com o gradiente total.

Ingenuamente, isso indicaria que escolher um grande minibatch $\mathcal{B}_t$ seria universalmente desejável. Infelizmente, depois de algum ponto, a redução adicional no desvio padrão é mínima quando comparada ao aumento linear no custo computacional. Na prática, escolhemos um minibatch que é grande o suficiente para oferecer boa eficiência computacional e ainda caber na memória de uma GPU. Para ilustrar a economia, vamos dar uma olhada em alguns códigos. Nele realizamos a mesma multiplicação matriz-matriz, mas desta vez dividida em "minibatches" de 64 colunas por vez.

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = torch.mm(B, C[:, j:j+64])
timer.stop()
print(f'performance in Gigaflops: block {2 / timer.times[3]:.3f}')
```

```
performance in Gigaflops: block 2711.250
```

Como podemos ver, o cálculo no minibatch é essencialmente tão eficiente quanto na matriz completa. Uma palavra de cautela é necessária. Em Section 7.5 usamos um tipo de regularização que era fortemente dependente da quantidade de variância em um minibatch. À medida que aumentamos o último, a variância diminui e com ela o benefício da injeção de ruído devido à normalização do lote. Consulte, por exemplo, (Ioffe, 2017) para obter detalhes sobre como redimensionar e calcular os termos apropriados.

11.5.3 Lendo o conjunto de dados

Vamos dar uma olhada em como os minibatches são gerados com eficiência a partir de dados. A seguir, usamos um conjunto de dados desenvolvido pela NASA para testar a asa ruído de aeronaves diferentes[115] para comparar esses algoritmos de otimização. Por conveniência, usamos apenas os primeiros 1.500 exemplos. Os dados são clareados para pré-processamento, ou seja, removemos a média e redimensionamos a variação para 1 por coordenada.

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),
                               batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

[115] https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

11.5.4 Implementação do zero

Lembre-se da implementação SGD do minibatch de Section 3.2. A seguir, fornecemos uma implementação um pouco mais geral. Por conveniência, ele tem a mesma assinatura de chamada que os outros algoritmos de otimização introduzidos posteriormente neste capítulo. Especificamente, adicionamos o status insira os `estados` e coloque o hiperparâmetro nos `hiperparâmetros` do dicionário. Dentro Além disso, calcularemos a média da perda de cada exemplo de minibatch no treinamento função, então o gradiente no algoritmo de otimização não precisa ser dividido pelo tamanho do lote.

```
def sgd(params, states, hyperparams):
    for p in params:
        p.data.sub_(hyperparams['lr'] * p.grad)
        p.grad.data.zero_()
```

A seguir, implementamos uma função de treinamento genérica para facilitar o uso de outros algoritmos de otimização introduzidos posteriormente neste capítulo. Ele inicializa um modelo de regressão linear e pode ser usado para treinar o modelo com minibatch SGD e outros algoritmos introduzidos posteriormente.

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # Initialization
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),
                     requires_grad=True)
    b = torch.zeros((1), requires_grad=True)
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # Train
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
    return timer.cumsum(), animator.Y[0]
```

Vamos ver como a otimização procede para a descida do gradiente do lote. Isso pode ser alcançado definindo o tamanho do minibatch para 1500 (ou seja, para o número total de exemplos). Como resultado, os parâmetros do modelo são atualizados apenas uma vez por época. Há pouco progresso. Na verdade, após 6 etapas, o progresso é interrompido.

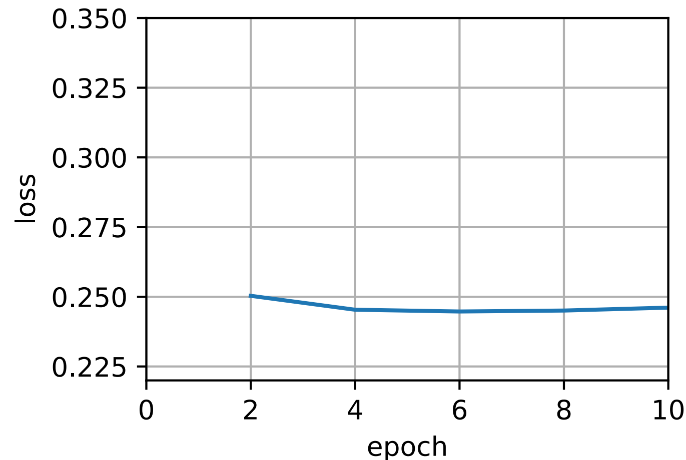
```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
```

(continues on next page)

(continued from previous page)

```
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

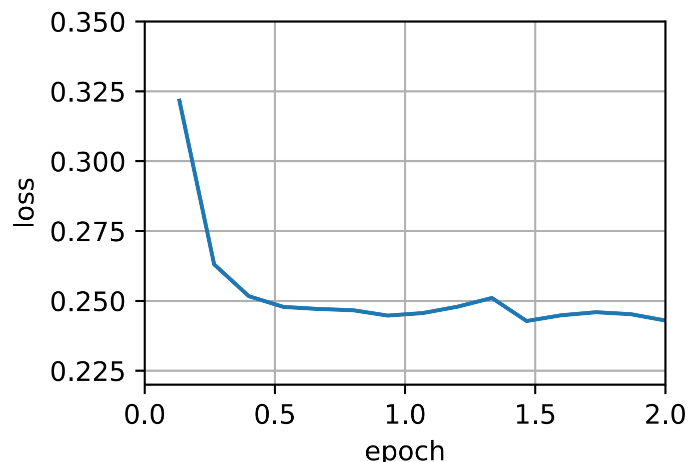
```
loss: 0.246, 0.021 sec/epoch
```



Quando o tamanho do lote é igual a 1, usamos SGD para otimização. Para simplificar a implementação, escolhemos uma taxa de aprendizado constante (embora pequena). No SGD, os parâmetros do modelo são atualizados sempre que um exemplo é processado. Em nosso caso, isso equivale a 1.500 atualizações por época. Como podemos ver, o declínio no valor da função objetivo diminui após uma época. Embora ambos os procedimentos tenham processado 1.500 exemplos em uma época, o SGD consome mais tempo do que a descida de gradiente em nosso experimento. Isso ocorre porque o SGD atualizou os parâmetros com mais frequência e porque é menos eficiente processar observações únicas uma de cada vez.

```
sgd_res = train_sgd(0.005, 1)
```

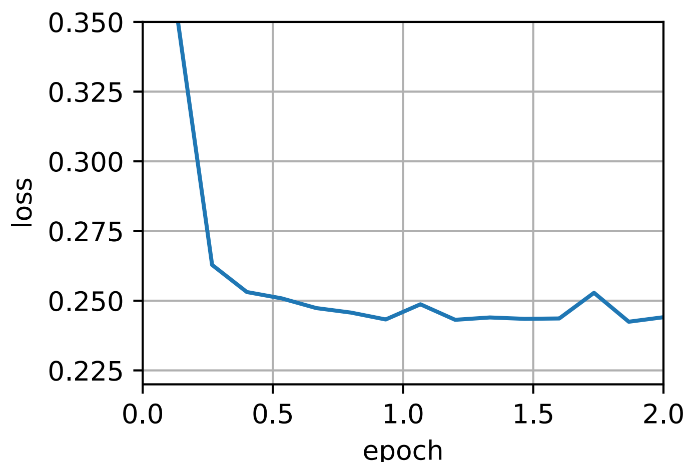
```
loss: 0.243, 0.073 sec/epoch
```



Finalmente, quando o tamanho do lote é igual a 100, usamos minibatch SGD para otimização. O tempo necessário por época é menor do que o tempo necessário para SGD e o tempo para a descida do gradiente do lote.

```
mini1_res = train_sgd(.4, 100)
```

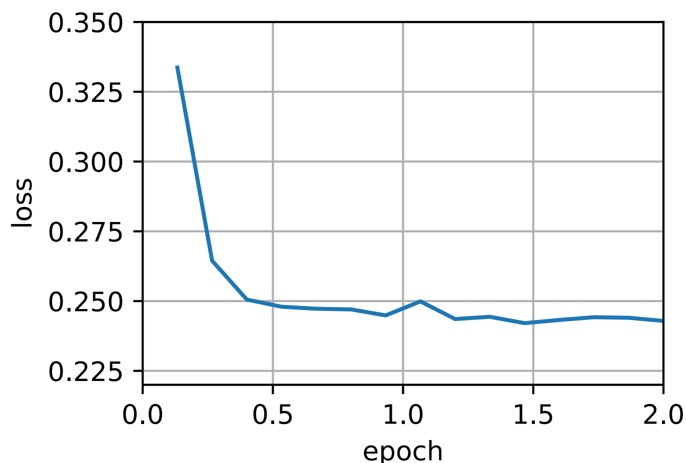
```
loss: 0.244, 0.003 sec/epoch
```



Reduzindo o tamanho do lote para 10, o tempo de cada época aumenta porque a carga de trabalho de cada lote é menos eficiente de executar.

```
mini2_res = train_sgd(.05, 10)
```

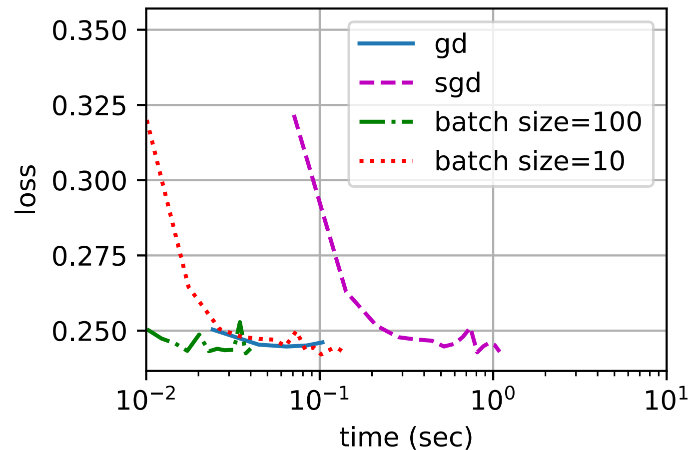
```
loss: 0.243, 0.009 sec/epoch
```



Agora podemos comparar o tempo versus a perda dos quatro experimentos anteriores. Como pode ser visto, embora SGD convirja mais rápido do que GD em termos de número de exemplos processados, ele usa mais tempo para atingir a mesma perda do que GD porque calcular o gradiente exemplo por exemplo não é tão eficiente. O Minibatch SGD é capaz de compensar a velocidade de convergência e a eficiência de computação. Um tamanho de minibatch de 10 é mais

eficiente do que SGD; um tamanho de minibatch de 100 supera até mesmo o GD em termos de tempo de execução.

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```



11.5.5 Implementação concisa

No Gluon, podemos usar a classe `Trainer` para chamar algoritmos de otimização. Isso é usado para implementar uma função de treinamento genérica. Usaremos isso em todo o capítulo atual.

```
#@save
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=4):
    # Initialization
    net = nn.Sequential(nn.Linear(5, 1))
    def init_weights(m):
        if type(m) == nn.Linear:
            torch.nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)

    optimizer = trainer_fn(net.parameters(), **hyperparams)

    loss = nn.MSELoss()
    # Note: L2 Loss = 1/2 * MSE Loss. PyTorch has MSE Loss which is slightly
    # different from MXNet's L2Loss by a factor of 2. Hence we halve the loss
    # value to get L2Loss in PyTorch
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            optimizer.zero_grad()
            out = net(X)
            y = y.reshape(out.shape)
            l = loss(out, y)/2
```

(continues on next page)

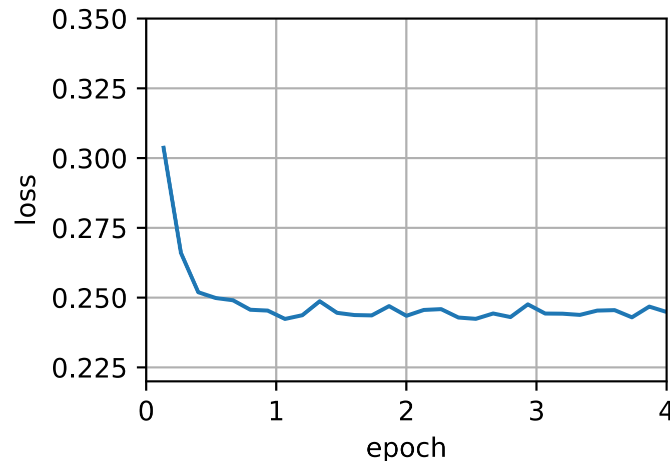
(continued from previous page)

```
            l.backward()
            optimizer.step()
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss)/2,))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
```

Usar o Gluon para repetir o último experimento mostra um comportamento idêntico.

```
data_iter, _ = get_data_ch11(10)
trainer = torch.optim.SGD
train_concise_ch11(trainer, {'lr': 0.05}, data_iter)
```

```
loss: 0.245, 0.011 sec/epoch
```



11.5.6 Sumário

- A vetorização torna o código mais eficiente devido à sobrecarga reduzida decorrente da estrutura de aprendizado profundo e devido à melhor localização da memória e armazenamento em cache em CPUs e GPUs.
- Há uma compensação entre a eficiência estatística decorrente do SGD e a eficiência computacional decorrente do processamento de grandes lotes de dados por vez.
- A descida gradiente estocástica do Minibatch oferece o melhor dos dois mundos: eficiência computacional e estatística.
- No minibatch SGD, processamos lotes de dados obtidos por uma permutação aleatória dos dados de treinamento (ou seja, cada observação é processada apenas uma vez por época, embora em ordem aleatória).
- É aconselhável diminuir as taxas de aprendizagem durante o treinamento.

- Em geral, minibatch SGD é mais rápido do que SGD e gradiente descendente para convergência para um risco menor, quando medido em termos de tempo de clock.

11.5.7 Exercícios

1. Modifique o tamanho do lote e a taxa de aprendizado e observe a taxa de declínio para o valor da função objetivo e o tempo consumido em cada época.
2. Leia a documentação do MXNet e use a função da classe `Trainer` `set_learning_rate` para reduzir a taxa de aprendizagem do minibatch SGD para 1/10 de seu valor anterior após cada época.
3. Compare o minibatch SGD com uma variante que, na verdade, *obtém amostras com substituição* do conjunto de treinamento. O que acontece?
4. Um gênio do mal replica seu conjunto de dados sem avisar você (ou seja, cada observação ocorre duas vezes e seu conjunto de dados cresce para o dobro do tamanho original, mas ninguém lhe disse). Como o comportamento do SGD, do minibatch SGD e do gradiente de descida muda?

Discussão[116]

11.6 Momentum

Em Section 11.4, revisamos o que acontece ao realizar a descida do gradiente estocástico, ou seja, ao realizar a otimização onde apenas uma variante barulhenta do gradiente está disponível. Em particular, notamos que, para gradientes ruidosos, precisamos ser extremamente cautelosos ao escolher a taxa de aprendizado em face do ruído. Se diminuirmos muito rapidamente, a convergência para. Se formos tolerantes demais, não conseguiremos convergir para uma solução boa o suficiente, pois o ruído continua nos afastando da otimização.

11.6.1 Fundamentos

Nesta seção, exploraremos algoritmos de otimização mais eficazes, especialmente para certos tipos de problemas de otimização que são comuns na prática.

Médias com vazamento

A seção anterior nos viu discutindo o minibatch SGD como um meio de acelerar a computação. Também teve o bom efeito colateral de que a média dos gradientes reduziu a quantidade de variância. O minibatch SGD pode ser calculado por:

$$mathbf{g}_{t,t-1} = partial_{mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(mathbf{x}_{i}, mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}. \tag{11.6.1}$$

Para manter a notação simples, aqui usamos $\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$ como o SGD para a amostra i usando os pesos atualizados no tempo $ t-1 $. Seria bom se pudéssemos nos beneficiar do efeito

[116] https://discuss.d2l.ai/t/1068

da redução da variância, mesmo além da média dos gradientes em um minibatch. Uma opção para realizar esta tarefa é substituir o cálculo do gradiente por uma "média com vazamento":

$$\mathbf{v}_t = \beta\mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \tag{11.6.2}$$

por algum $\beta \in (0, 1)$. Isso substitui efetivamente o gradiente instantâneo por um que foi calculado em vários gradientes *anteriores*. $\mathbf{v}$ é chamado *momentum*. Ele acumula gradientes anteriores semelhantes a como uma bola pesada rolando pela paisagem da função objetivo se integra às forças passadas. Para ver o que está acontecendo com mais detalhes, vamos expandir $\mathbf{v}_t$ recursivamente em

$$\mathbf{v}_t = \beta^2\mathbf{v}_{t-2} + \beta\mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \ldots, = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}. \tag{11.6.3}$$

β grande equivale a uma média de longo alcance, enquanto β pequeno equivale a apenas uma ligeira correção em relação a um método de gradiente. A nova substituição de gradiente não aponta mais para a direção da descida mais íngreme em uma instância particular, mas sim na direção de uma média ponderada de gradientes anteriores. Isso nos permite obter a maioria dos benefícios da média de um lote sem o custo de realmente calcular os gradientes nele. Iremos revisitar este procedimento de média com mais detalhes posteriormente.

O raciocínio acima formou a base para o que agora é conhecido como métodos de gradiente *acelerado*, como gradientes com momentum. Eles têm o benefício adicional de serem muito mais eficazes nos casos em que o problema de otimização é mal condicionado (ou seja, onde há algumas direções onde o progresso é muito mais lento do que em outras, parecendo um desfiladeiro estreito). Além disso, eles nos permitem calcular a média dos gradientes subsequentes para obter direções de descida mais estáveis. Na verdade, o aspecto da aceleração, mesmo para problemas convexos sem ruído, é uma das principais razões pelas quais o momentum funciona e por que funciona tão bem.

Como seria de esperar, devido ao seu momentum de eficácia, é um assunto bem estudado em otimização para aprendizado profundo e além. Veja, por exemplo, o belo artigo expositivo[117] por (Goh, 2017) para uma análise aprofundada e animação interativa. Foi proposto por (Polyak, 1964). (Nesterov, 2018) tem uma discussão teórica detalhada no contexto da otimização convexa. O momentum no aprendizado profundo é conhecido por ser benéfico há muito tempo. Veja, por exemplo, a discussão de (Sutskever et al., 2013) para obter detalhes.

Um problema mal condicionado

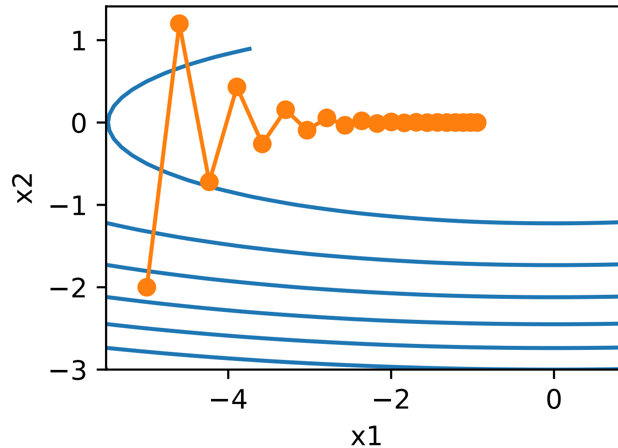
Para obter uma melhor compreensão das propriedades geométricas do método do momento, revisitamos a descida do gradiente, embora com uma função objetivo significativamente menos agradável. Lembre-se de que em Section 11.3 usamos $f(\mathbf{x}) = x_1^2 + 2x_2^2$, ou seja, um objetivo elipsóide moderadamente distorcido. Distorcemos esta função ainda mais estendendo-a na direção x_1 por meio de

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{11.6.4}$$

Como antes, f tem seu mínimo em $(0, 0)$. Esta função é *muito* plana na direção de x_1. Vamos ver o que acontece quando executamos a descida gradiente como antes nesta nova função. Escolhemos uma taxa de aprendizagem de $0, 4$.

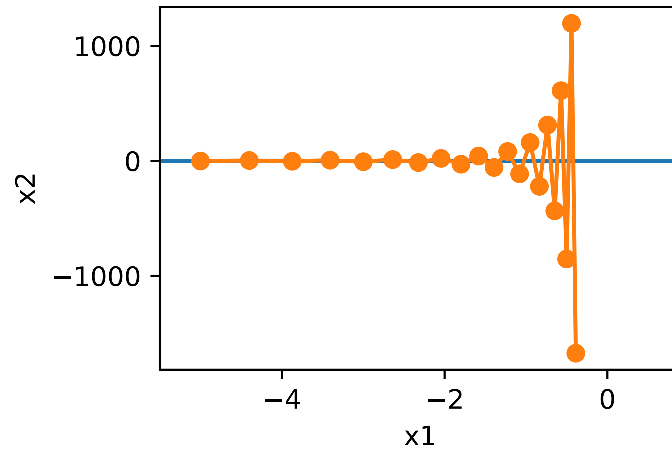
[117] https://distill.pub/2017/momentum/

```
%matplotlib inline
import torch
from d2l import torch as d2l

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```



Por construção, o gradiente na direção x_2 é *muito* maior e muda muito mais rapidamente do que na direção x_1 horizontal. Portanto, estamos presos entre duas escolhas indesejáveis: se escolhermos uma pequena taxa de aprendizado, garantimos que a solução não diverge na direção x_2, mas estamos sobrecarregados com uma convergência lenta na direção x_1. Por outro lado, com uma grande taxa de aprendizado, progredimos rapidamente na direção x_1, mas divergimos em x_2. O exemplo abaixo ilustra o que acontece mesmo após um ligeiro aumento na taxa de aprendizagem de $0,4$ para $0,6$. A convergência na direção x_1 melhora, mas a qualidade geral da solução é muito pior.

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```



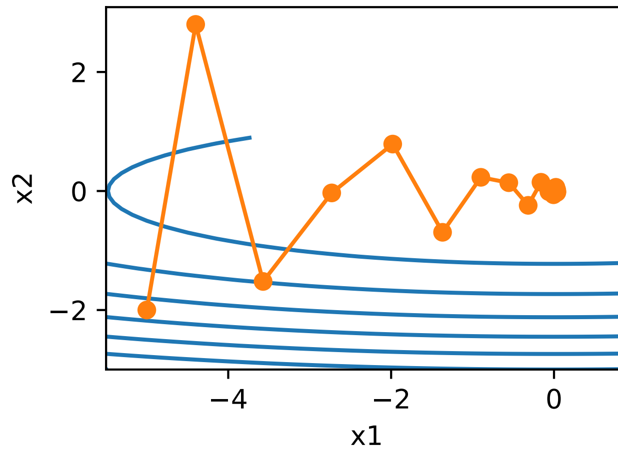
Método Momentum

O método do momento nos permite resolver o problema de descida gradiente descrito acima de. Olhando para o traço de otimização acima, podemos intuir que calcular a média de gradientes em relação ao passado funcionaria bem. Afinal, na direção x_1, isso agregará gradientes bem alinhados, aumentando assim a distância que percorremos a cada passo. Por outro lado, na direção x_2 onde os gradientes oscilam, um gradiente agregado reduzirá o tamanho do passo devido às oscilações que se cancelam. Usar $\mathbf{v}_t$ em vez do gradiente $\mathbf{g}_t$ produz as seguintes equações de atualização:

$$\begin{aligned}
\mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t, t-1}, \\
\mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.
\end{aligned} \tag{11.6.5}$$

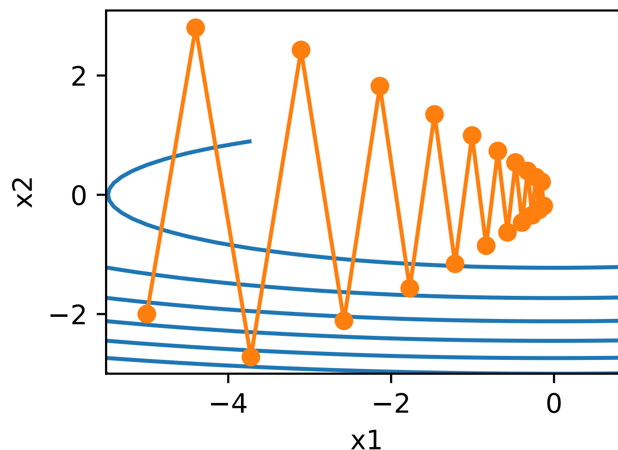
Observe que para $\beta = 0$ recuperamos a descida gradiente regular. Antes de nos aprofundarmos nas propriedades matemáticas, vamos dar uma olhada rápida em como o algoritmo se comporta na prática.

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```



Como podemos ver, mesmo com a mesma taxa de aprendizado que usamos antes, o momentum ainda converge bem. Vamos ver o que acontece quando diminuímos o parâmetro momentum. Reduzi-lo para $\beta = 0,25$ leva a uma trajetória que quase não converge. No entanto, é muito melhor do que sem momentum (quando a solução diverge).

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

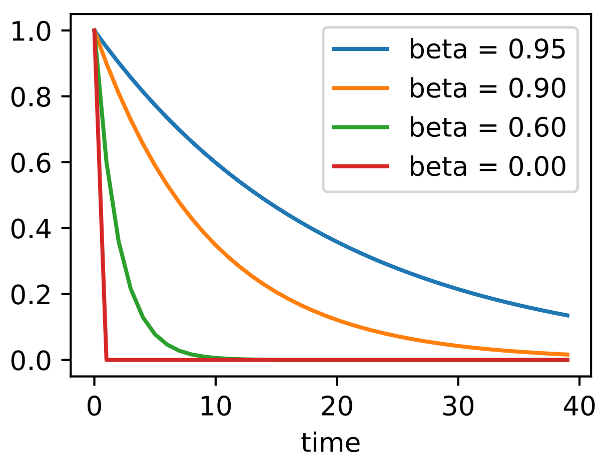


Observe que podemos combinar momentum com SGD e, em particular, minibatch-SGD. A única mudança é que, nesse caso, substituímos os gradientes $\mathbf{g}_{t,t-1}$ por $\mathbf{g}_t$. Por último, por conveniência, inicializamos $\mathbf{v}_0 = 0$ no momento $t = 0$. Vejamos o que a média de vazamento realmente faz com as atualizações.

Peso Efetivo da Amostra

Lembre-se de que $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^{\tau} \mathbf{g}_{t-\tau, t-\tau-1}$. No limite, os termos somam $\sum_{\tau=0}^{\infty} \beta^{\tau} = \frac{1}{1-\beta}$. Em outras palavras, em vez de dar um passo de tamanho η em GD ou SGD, damos um passo de tamanho $\frac{\eta}{1-\beta}$ enquanto, ao mesmo tempo, lidamos com um potencial muito direção de descida melhor comportada. Esses são dois benefícios em um. Para ilustrar como a ponderação se comporta para diferentes escolhas de β, considere o diagrama abaixo.

```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



11.6.2 Experimentos Práticos

Vamos ver como o momentum funciona na prática, ou seja, quando usado no contexto de um otimizador adequado. Para isso, precisamos de uma implementação um pouco mais escalonável.

Implementação do zero

Em comparação com (minibatch) SGD, o método de momentum precisa manter um conjunto de variáveis auxiliares, ou seja, a velocidade. Tem a mesma forma dos gradientes (e variáveis do problema de otimização). Na implementação abaixo, chamamos essas variáveis de `estados`.

```
def init_momentum_states(feature_dim):
    v_w = torch.zeros((feature_dim, 1))
    v_b = torch.zeros(1)
    return (v_w, v_b)
```

```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
```

(continues on next page)

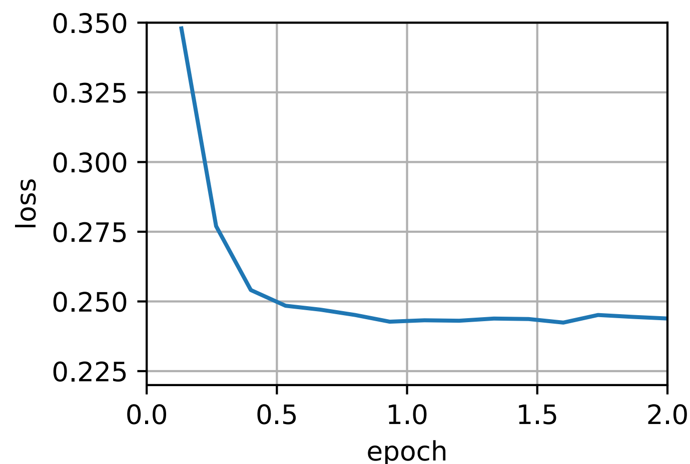
(continued from previous page)

```
        with torch.no_grad():
            v[:] = hyperparams['momentum'] * v + p.grad
            p[:] -= hyperparams['lr'] * v
        p.grad.data.zero_()
```

Vamos ver como isso funciona na prática.

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

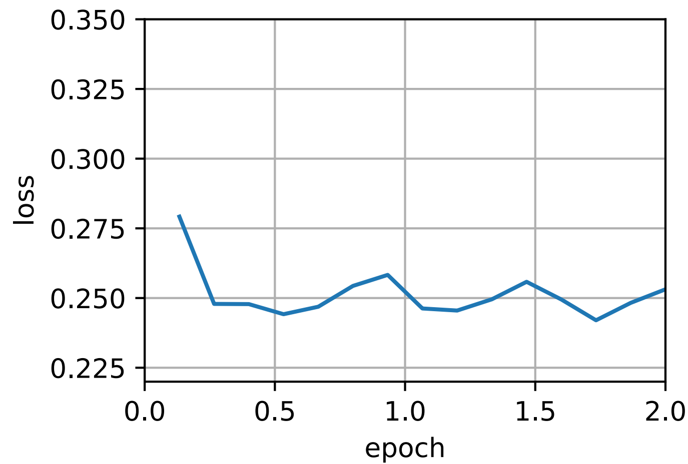
```
loss: 0.244, 0.011 sec/epoch
```



Quando aumentamos o hiperparâmetro de momento `momentum` para 0,9, resulta em um tamanho de amostra efetivo significativamente maior de $\frac{1}{1-0.9} = 10$. Reduzimos ligeiramente a taxa de aprendizagem para $0,01$ para manter os assuntos sob controle.

```
train_momentum(0.01, 0.9)
```

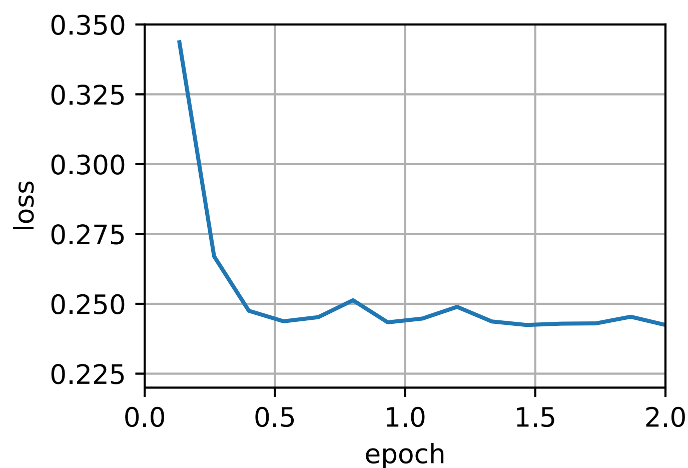
```
loss: 0.253, 0.011 sec/epoch
```



A redução da taxa de aprendizagem resolve ainda mais qualquer questão de problemas de otimização não suave. Configurá-lo como $0,005$ produz boas propriedades de convergência.

```
train_momentum(0.005, 0.9)
```

```
loss: 0.242, 0.012 sec/epoch
```

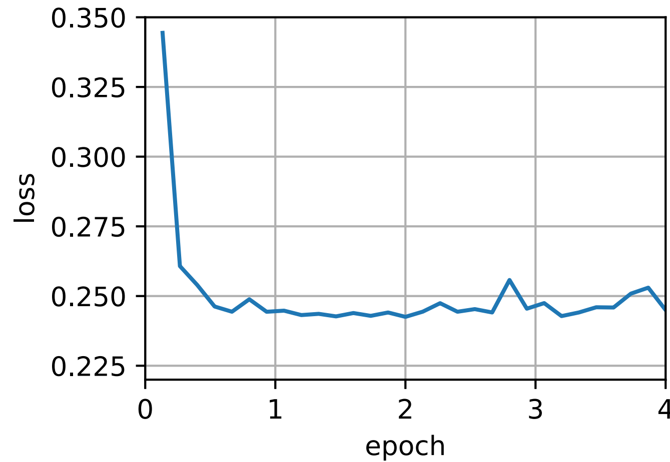


Implementação concisa

Há muito pouco a fazer no Gluon, uma vez que o solucionador sgd padrão já tem o momentum embutido. A configuração dos parâmetros correspondentes produz uma trajetória muito semelhante.

```
trainer = torch.optim.SGD
d2l.train_concise_ch11(trainer, {'lr': 0.005, 'momentum': 0.9}, data_iter)
```

```
loss: 0.245, 0.011 sec/epoch
```



11.6.3 Análise teórica

Até agora, o exemplo 2D de $f(x) = 0.1x_1^2 + 2x_2^2$ parecia bastante artificial. Veremos agora que isso é na verdade bastante representativo dos tipos de problemas que podemos encontrar, pelo menos no caso de minimizar funções objetivas quadráticas convexas.

Funções quadráticas convexas

Considere a função

$$h(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \tag{11.6.6}$$

Esta é uma função quadrática geral. Para matrizes definidas positivas $\mathbf{Q} \succ 0$, ou seja, para matrizes com autovalores positivos, tem um minimizador em $\mathbf{x}^* = -\mathbf{Q}^{-1} \mathbf{c}$ com valor mínimo $b - \frac{1}{2} \mathbf{c}^\top \mathbf{Q}^{-1} \mathbf{c}$. Portanto, podemos reescrever h como

$$h(\mathbf{x}) = \frac{1}{2} (\mathbf{x} - \mathbf{Q}^{-1} \mathbf{c})^\top \mathbf{Q} (\mathbf{x} - \mathbf{Q}^{-1} \mathbf{c}) + b - \frac{1}{2} \mathbf{c}^\top \mathbf{Q}^{-1} \mathbf{c}. \tag{11.6.7}$$

O gradiente é dado por $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q} (\mathbf{x} - \mathbf{Q}^{-1} \mathbf{c})$. Ou seja, é dada pela distância entre $\mathbf{x}$ e o minimizador, multiplicada por $\mathbf{Q}$. Consequentemente, também o momento é uma combinação linear de termos $\mathbf{Q} (\mathbf{x}_t - \mathbf{Q}^{-1} \mathbf{c})$.

Uma vez que $\mathbf{Q}$ é definido positivo, pode ser decomposto em seu auto-sistema via $\mathbf{Q} = \mathbf{O}^\top \boldsymbol{\Lambda} \mathbf{O}$ para um ortogonal (rotação) matriz $\mathbf{O}$ e uma matriz diagonal $\boldsymbol{\Lambda}$ de autovalores positivos. Isso nos permite realizar uma mudança de variáveis de $\mathbf{x}$ para $\mathbf{z} := \mathbf{O} (\mathbf{x} - \mathbf{Q}^{-1} \mathbf{c})$ para obter uma expressão muito simplificada:

$$h(\mathbf{z}) = \frac{1}{2} \mathbf{z}^\top \boldsymbol{\Lambda} \mathbf{z} + b'. \tag{11.6.8}$$

Aqui $c' = b - \frac{1}{2} \mathbf{c}^\top \mathbf{Q}^{-1} \mathbf{c}$. Uma vez que $\mathbf{O}$ é apenas uma matriz ortogonal, isso não perturba os gradientes de uma forma significativa. Expresso em termos de $\mathbf{z}$ gradiente, a descida torna-se

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \boldsymbol{\Lambda} \mathbf{z}_{t-1} = (\mathbf{I} - \boldsymbol{\Lambda}) \mathbf{z}_{t-1}. \tag{11.6.9}$$

O fato importante nesta expressão é que a descida gradiente *não se mistura* entre diferentes espaços auto. Ou seja, quando expresso em termos do autossistema de $\mathbf{Q}$, o problema de otimização ocorre de maneira coordenada. Isso também vale para o momento.

$$\begin{aligned}
\mathbf{v}_t & = \beta \mathbf{v}_{t-1} + \boldsymbol{\Lambda} \mathbf{z}_{t-1} \\
\mathbf{z}_t & = \mathbf{z}_{t-1} - \eta \left(\beta \mathbf{v}_{t-1} + \boldsymbol{\Lambda} \mathbf{z}_{t-1}\right) \\
& = (\mathbf{I} - \eta \boldsymbol{\Lambda}) \mathbf{z}_{t-1} - \eta \beta \mathbf{v}_{t-1}.
\end{aligned} \tag{11.6.10}$$

Ao fazer isso, acabamos de provar o seguinte teorema: Gradiente descendente com e sem momento para uma função quadrática convexa se decompõe em otimização coordenada na direção dos vetores próprios da matriz quadrática.

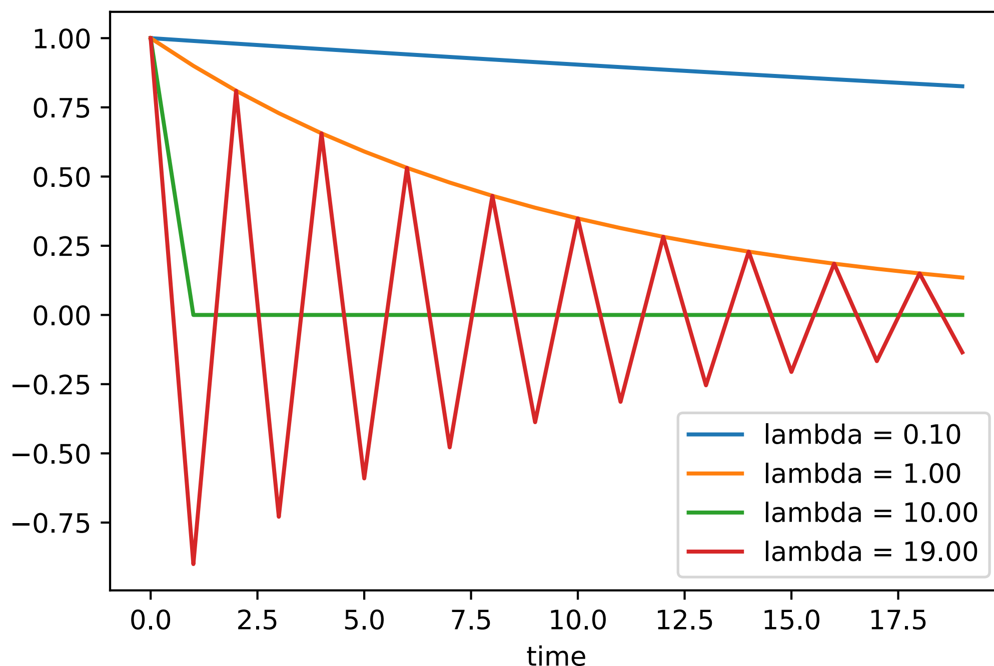
Funções Escalares

Dado o resultado acima, vamos ver o que acontece quando minimizamos a função $f(x) = \frac{\lambda}{2} x^2$. Para descida gradiente, temos

$$x_{t+1} = x_t - \eta \lambda x_t = (1 - \eta \lambda) x_t. \tag{11.6.11}$$

Sempre que $|1 - \eta \lambda| < 1$ esta otimização converge a uma taxa exponencial, pois após t passos temos $x_t = (1 - \eta \lambda)^t x_0$. Isso mostra como a taxa de convergência melhora inicialmente à medida que aumentamos a taxa de aprendizado η até $\eta \lambda = 1$. Além disso, as coisas divergem e para $\eta \lambda > 2$ o problema de otimização diverge.

```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = torch.arange(20).detach().numpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



Para analisar a convergência no caso de momentum, começamos reescrevendo as equações de atualização em termos de dois escalares: um para x e outro para o momentum v. Isso produz:

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1-\eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \tag{11.6.12}$$

Usamos $\mathbf{R}$ para denotar o comportamento de convergência que rege 2×2. Após t passos, a escolha inicial $[v_0, x_0]$ torna-se $\mathbf{R}(\beta, \eta, \lambda)^t [v_0, x_0]$. Consequentemente, cabe aos autovalores de $\mathbf{R}$ determinar a velocidade de convergência. Veja o Post do Destill[118] de (Goh, 2017) para uma ótima animação e (Flammarion & Bach, 2015) para uma análise detalhada. Pode-se mostrar que $0 < \eta\lambda < 2 + 2\beta$ momentum converge. Este é um intervalo maior de parâmetros viáveis quando comparado a $0 < \eta\lambda < 2$ para descida de gradiente. Também sugere que, em geral, grandes valores de β são desejáveis. Mais detalhes requerem uma boa quantidade de detalhes técnicos e sugerimos que o leitor interessado consulte as publicações originais.

11.6.4 Sumário

- Momentum substitui gradientes por uma média com vazamento em relação aos gradientes anteriores. Isso acelera a convergência significativamente.
- É desejável tanto para descida gradiente sem ruído quanto para descida gradiente estocástica (ruidosa).
- O momentum evita a paralisação do processo de otimização, que é muito mais provável de ocorrer na descida do gradiente estocástico.
- O número efetivo de gradientes é dado por $\frac{1}{1-\beta}$ devido à redução exponenciada de dados anteriores.
- No caso de problemas quadráticos convexos, isso pode ser analisado explicitamente em detalhes.
- A implementação é bastante direta, mas exige que armazenemos um vetor de estado adicional (momentum $\mathbf{v}$).

11.6.5 Exercícios

1. Use outras combinações de hiperparâmetros de momentum e taxas de aprendizagem e observe e analise os diferentes resultados experimentais.
2. Experimente GD e momentum para um problema quadrático onde você tem vários autovalores, ou seja, $f(x) = \frac{1}{2} \sum_i \lambda_i x_i^2$ ou seja $\lambda_i = 2^{-i}$. Trace como os valores de x diminuem para a inicialização $x_i = 1$.
3. Derive o valor mínimo e minimizador para $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b$.
4. O que muda quando executamos SGD com momentum? O que acontece quando usamos minibatch SGD com momentum? Experimentar com os parâmetros?

Discussão[119]

[118] https://distill.pub/2017/momentum/
[119] https://discuss.d2l.ai/t/1070

11.7 Adagrad

Vamos começar considerando os problemas de aprendizado com recursos que ocorrem com pouca frequência.

11.7.1 Recursos esparsos e taxas de aprendizado

Imagine que estamos treinando um modelo de linguagem. Para obter uma boa precisão, normalmente queremos diminuir a taxa de aprendizado à medida que continuamos treinando, normalmente a uma taxa de $\mathcal{O}(t^{-\frac{1}{2}})$ ou mais lenta. Agora, considere um treinamento de modelo em recursos esparsos, ou seja, recursos que ocorrem raramente. Isso é comum para a linguagem natural, por exemplo, é muito menos provável que vejamos a palavra *précondicionamento* do que *aprendizagem*. No entanto, também é comum em outras áreas, como publicidade computacional e filtragem colaborativa personalizada. Afinal, existem muitas coisas que interessam apenas a um pequeno número de pessoas.

Os parâmetros associados a recursos pouco frequentes recebem apenas atualizações significativas sempre que esses recursos ocorrem. Dada uma taxa de aprendizado decrescente, podemos acabar em uma situação em que os parâmetros para características comuns convergem rapidamente para seus valores ideais, enquanto para características raras ainda não podemos observá-los com frequência suficiente antes que seus valores ideais possam ser determinados. Em outras palavras, a taxa de aprendizado diminui muito lentamente para recursos freqüentes ou muito rapidamente para recursos pouco frequentes.

Um possível hack para corrigir esse problema seria contar o número de vezes que vemos um determinado recurso e usar isso como um relógio para ajustar as taxas de aprendizagem. Ou seja, em vez de escolher uma taxa de aprendizagem da forma $\eta = \frac{\eta_0}{\sqrt{t + c}}$ poderiamos usar $\eta_i = \frac{\eta_0}{\sqrt{s(i, t) + c}}$ conta o número de valores diferentes de zero para o recurso i que observamos até o momento t. Na verdade, isso é muito fácil de implementar sem sobrecarga significativa. No entanto, ele falha sempre que não temos esparsidade, mas apenas dados em que os gradientes são frequentemente muito pequenos e raramente grandes. Afinal, não está claro onde se traçaria a linha entre algo que se qualifica como uma característica observada ou não.

Adagrad by (Duchi et al., 2011) aborda isso substituindo o contador bastante bruto $s(i, t)$ por um agregado de quadrados de gradientes previamente observados. Em particular, ele usa $s(i, t+1) = s(i, t) + \left(\partial_i f(\mathbf{x})\right)^2$ como um meio de ajustar a taxa de aprendizagem. Isso tem dois benefícios: primeiro, não precisamos mais decidir apenas quando um gradiente é grande o suficiente. Em segundo lugar, ele é dimensionado automaticamente com a magnitude dos gradientes. As coordenadas que normalmente correspondem a grandes gradientes são reduzidas significativamente, enquanto outras com pequenos gradientes recebem um tratamento muito mais suave. Na prática, isso leva a um procedimento de otimização muito eficaz para publicidade computacional e problemas relacionados. Mas isso oculta alguns dos benefícios adicionais inerentes ao Adagrad que são mais bem compreendidos no contexto do pré-condicionamento.

11.7.2 Precondicionamento

Problemas de otimização convexa são bons para analisar as características dos algoritmos. Afinal, para a maioria dos problemas não-convexos, é difícil derivar garantias teóricas significativas, mas a *intuição* e o *insight* geralmente são transmitidos. Vejamos o problema de minimizar $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$.

Como vimos em Section 11.6, é possível reescrever este problema em termos de sua composição automática $\mathbf{Q} = \mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U}$ para chegar a um problema muito simplificado onde cada coordenada pode ser resolvida individualmente:

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \boldsymbol{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \tag{11.7.1}$$

Aqui usamos $\mathbf{x} = \mathbf{U}\mathbf{x}$ e consequentemente $\mathbf{c} = \mathbf{U}\mathbf{c}$. O problema modificado tem como minimizador $\bar{\mathbf{x}} = -\boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}}$ e valor mínimo $-\frac{1}{2}\bar{\mathbf{c}}^\top \boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}} + b$. Isso é muito mais fácil de calcular, pois $\boldsymbol{\Lambda}$ é uma matriz diagonal contendo os autovalores de $\mathbf{Q}$.

Se perturbarmos $\mathbf{c}$ ligeiramente, esperaríamos encontrar apenas pequenas mudanças no minimizador de $ f $. Infelizmente, esse não é o caso. Embora pequenas mudanças em $\mathbf{c}$ levem a mudanças igualmente pequenas em $\bar{\mathbf{c}}$, este não é o caso para o minimizador de f (e de $\bar{f}$ respectivamente). Sempre que os autovalores $\boldsymbol{\Lambda}_i$ forem grandes, veremos apenas pequenas mudanças em $\bar{x}_i$ e no mínimo de $\bar{f}$. Por outro lado, para pequenas $\boldsymbol{\Lambda}_i$, as mudanças em $\bar{x}_i$ podem ser dramáticas. A razão entre o maior e o menor autovalor é chamada de número de condição de um problema de otimização.

$$\kappa = \frac{\boldsymbol{\Lambda}_1}{\boldsymbol{\Lambda}_d}. \tag{11.7.2}$$

Se o número de condição κ for grande, será difícil resolver o problema de otimização com precisão. Precisamos garantir que somos cuidadosos ao acertar uma ampla faixa dinâmica de valores. Nossa análise leva a uma questão óbvia, embora um tanto ingênua: não poderíamos simplesmente "consertar" o problema distorcendo o espaço de forma que todos os autovalores sejam 1. Em teoria, isso é muito fácil: precisamos apenas dos autovalores e autovetores de $\mathbf{Q}$ para redimensionar o problema de $\mathbf{x}$ para um em $\mathbf{z} := \boldsymbol{\Lambda}^{\frac{1}{2}}\mathbf{U}\mathbf{x}$. No novo sistema de coordenadas $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ poderia ser simplificado para $\|\mathbf{z}\|^2$. Infelizmente, esta é uma sugestão pouco prática. O cálculo de autovalores e autovetores é em geral muito mais caro do que resolver o problema real.

Embora o cálculo exato dos autovalores possa ser caro, adivinhá-los e computá-los de forma aproximada já pode ser muito melhor do que não fazer nada. Em particular, poderíamos usar as entradas diagonais de $\mathbf{Q}$ e redimensioná-las de acordo. Isso é *muito* mais barato do que calcular valores próprios.

$$\tilde{\mathbf{Q}} = \mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q})\mathbf{Q}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q}). \tag{11.7.3}$$

Neste caso, temos $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij}/\sqrt{\mathbf{Q}_{ii}\mathbf{Q}_{jj}}$ e especificamente $\tilde{\mathbf{Q}}_{ii} = 1$ para todo i. Na maioria dos casos, isso simplifica consideravelmente o número da condição. Por exemplo, nos casos que discutimos anteriormente, isso eliminaria totalmente o problema em questão, uma vez que o problema está alinhado ao eixo.

Infelizmente, enfrentamos ainda outro problema: no aprendizado profundo, normalmente nem mesmo temos acesso à segunda derivada da função objetivo: para $\mathbf{x} \in \mathbb{R}^d$ a segunda derivada, mesmo em um minibatch pode exigir $\mathcal{O}(d^2)$ espaço e trabalho para computar, tornando-o praticamente inviável. A ideia engenhosa do Adagrad é usar um proxy para aquela diagonal indescritível do Hessian que é relativamente barato para calcular e eficaz— a magnitude do gradiente em si.

Para ver por que isso funciona, vamos dar uma olhada em $\bar{f}(\bar{\mathbf{x}})$. Nós temos isso

$$\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}}) = \boldsymbol{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}} = \boldsymbol{\Lambda}\left(\bar{\mathbf{x}} - \bar{\mathbf{x}}_0\right), \tag{11.7.4}$$

onde $\bar{\mathbf{x}}_0$ é o minimizador de $\bar{f}$. Portanto, a magnitude do gradiente depende tanto de $\boldsymbol{\Lambda}$ quanto da distância da otimalidade. Se $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ não mudou, isso seria tudo o que é necessário. Afinal, neste caso, a magnitude do gradiente $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$ é suficiente. Como o AdaGrad é um algoritmo descendente de gradiente estocástico, veremos gradientes com variância diferente de zero mesmo em otimização. Como resultado, podemos usar com segurança a variância dos gradientes como um proxy barato para a escala de Hessian. Uma análise completa está além do escopo desta seção (seriam várias páginas). Recomendamos ao leitor (Duchi et al., 2011) para detalhes.

11.7.3 O Algoritmo

Vamos formalizar a discussão de cima. Usamos a variável $\mathbf{s}_t$ para acumular a variância do gradiente anterior como segue.

$$\begin{aligned}
\mathbf{g}_t & = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\
\mathbf{s}_t & = \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\
\mathbf{w}_t & = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.
\end{aligned} \tag{11.7.5}$$

Aqui, as operações são aplicadas de acordo com as coordenadas. Ou seja, $\mathbf{v}^2$ tem entradas v_i^2. Da mesma forma, $\frac{1}{\sqrt{v}}$ tem entradas $\frac{1}{\sqrt{v_i}}$ e $\mathbf{u} \cdot \mathbf{v}$ tem entradas $u_i v_i$. Como antes, η é a taxa de aprendizagem e ϵ é uma constante aditiva que garante que não dividamos por 0. Por último, inicializamos $\mathbf{s}_0 = \mathbf{0}$.

Assim como no caso do momentum, precisamos manter o controle de uma variável auxiliar, neste caso para permitir uma taxa de aprendizagem individual por coordenada. Isso não aumenta o custo do Adagrad significativamente em relação ao SGD, simplesmente porque o custo principal normalmente é calcular $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ e sua derivada.

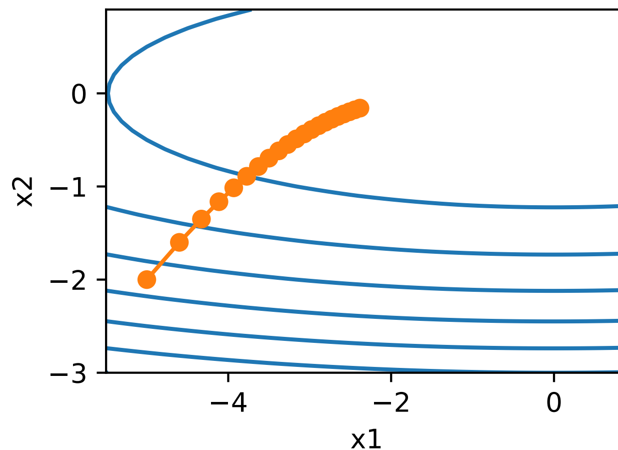
Observe que o acúmulo de gradientes quadrados em $\mathbf{s}_t$ significa que $\mathbf{s}_t$ cresce essencialmente a uma taxa linear (um pouco mais lento do que linearmente na prática, uma vez que os gradientes inicialmente diminuem). Isso leva a uma taxa de aprendizado $\mathcal{O}(t^{-\frac{1}{2}})$, embora ajustada por coordenada. Para problemas convexos, isso é perfeitamente adequado. No aprendizado profundo, porém, podemos querer diminuir a taxa de aprendizado um pouco mais lentamente. Isso levou a uma série de variantes do Adagrad que discutiremos nos capítulos subsequentes. Por enquanto, vamos ver como ele se comporta em um problema convexo quadrático. Usamos o mesmo problema de antes:

$$f(\mathbf{x}) = 0.1 x_1^2 + 2 x_2^2. \tag{11.7.6}$$

Vamos implementar o Adagrad usando a mesma taxa de aprendizado anterior, ou seja, $\eta = 0.4$. Como podemos ver, a trajetória iterativa da variável independente é mais suave. No entanto, devido ao efeito cumulativo de s_t, a taxa de aprendizado diminui continuamente, de modo que a variável independente não se move tanto durante os estágios posteriores da iteração.

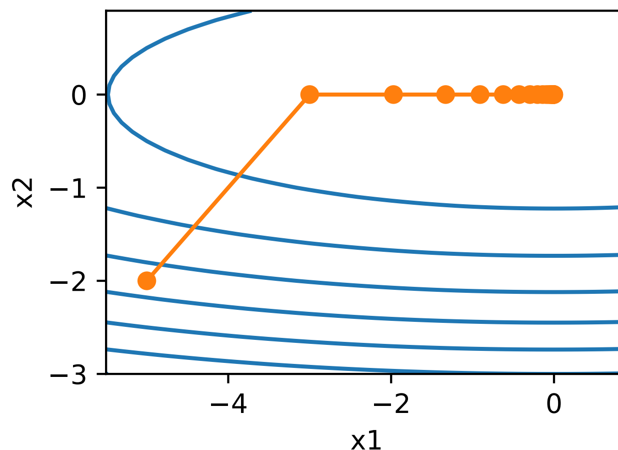
```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```



Conforme aumentamos a taxa de aprendizado para 2, vemos um comportamento muito melhor. Isso já indica que a diminuição na taxa de aprendizagem pode ser bastante agressiva, mesmo no caso sem ruído e precisamos garantir que os parâmetros convirjam de forma adequada.

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```



11.7.4 Implementação do zero

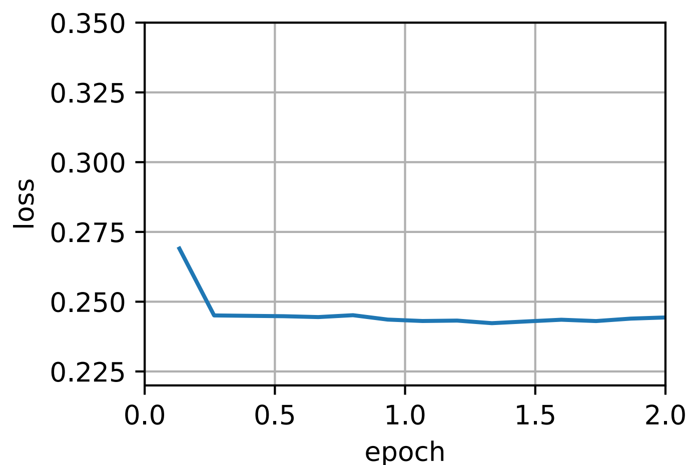
Assim como o método momentum, o Adagrad precisa manter uma variável de estado da mesma forma que os parâmetros.

```
def init_adagrad_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] += torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

Comparado com o experimento em Section 11.5, usamos um maior taxa de aprendizagem para treinar o modelo.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.012 sec/epoch
```

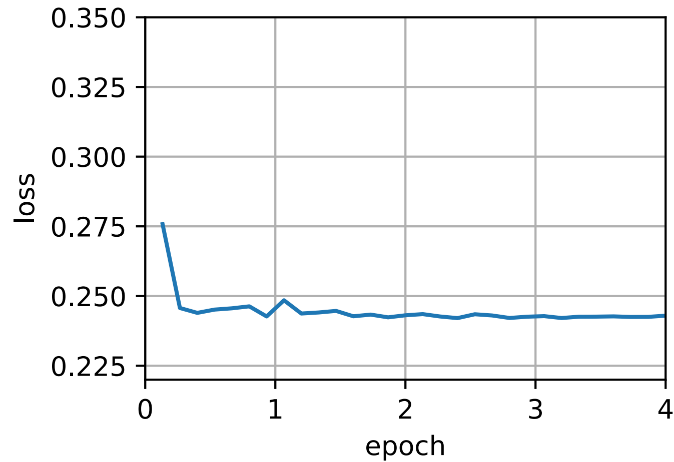


11.7.5 Implementação concisa

Usando a instância `Trainer` do algoritmo `adagrad`, podemos invocar o algoritmo Adagrad no Gluon.

```
trainer = torch.optim.Adagrad
d2l.train_concise_ch11(trainer, {'lr': 0.1}, data_iter)
```

```
loss: 0.243, 0.013 sec/epoch
```



11.7.6 Sumário

- O Adagrad diminui a taxa de aprendizagem dinamicamente por coordenada.
- Ele usa a magnitude do gradiente como um meio de ajustar a rapidez com que o progresso é alcançado - as coordenadas com gradientes grandes são compensadas com uma taxa de aprendizado menor.
- Calcular a segunda derivada exata é tipicamente inviável em problemas de aprendizado profundo devido a limitações de memória e computacionais. O gradiente pode ser um proxy útil.
- Se o problema de otimização tiver uma estrutura bastante irregular, o Adagrad pode ajudar a mitigar a distorção.
- O Adagrad é particularmente eficaz para recursos esparsos em que a taxa de aprendizado precisa diminuir mais lentamente para termos que ocorrem com pouca frequência.
- Em problemas de aprendizado profundo, o Adagrad às vezes pode ser muito agressivo na redução das taxas de aprendizado. Discutiremos estratégias para mitigar isso no contexto de Section 11.10.

11.7.7 Exercícios

1. Prove que para uma matriz ortogonal $\mathbf{U}$ e um vetor $\mathbf{c}$ o seguinte é válido: $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$. Por que isso significa que a magnitude das perturbações não muda após uma mudança ortogonal das variáveis?
2. Experimente o Adagrad para $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ e também para a função objetivo foi girada em 45 graus, ou seja, $f(\mathbf{x}) = 0.1(x_1+x_2)^2+2(x_1-x_2)^2$. Ele se comporta de maneira diferente?
3. Prove teorema do círculo de Gerschgorin[120] que afirma que os valores próprios λ_i de uma matriz $\mathbf{M}$ satisfazer $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k\neq j} |\mathbf{M}_{jk}|$ para pelo menos uma escolha de j.
4. O que o teorema de Gerschgorin nos diz sobre os autovalores da matriz pré-condicionada diagonalmente $\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})$?

[120] https://en.wikipedia.org/wiki/Gershgorin_circle_theorem

5. Experimente o Adagrad para uma rede profunda adequada, como Section 6.6 quando aplicado ao Fashion MNIST.
6. Como você precisaria modificar o Adagrad para atingir uma queda menos agressiva na taxa de aprendizado?

Discussão[121]

11.8 RMSProp

Um dos principais problemas em Section 11.7 é que a taxa de aprendizagem diminui em um cronograma predefinido de $\mathcal{O}(t^{-\frac{1}{2}})$. Embora geralmente seja apropriado para problemas convexos, pode não ser ideal para problemas não convexos, como os encontrados no aprendizado profundo. No entanto, a adaptabilidade coordenada do Adagrad é altamente desejável como um pré-condicionador.

(Tieleman & Hinton, 2012) propôs o algoritmo RMSProp como uma solução simples para desacoplar o escalonamento de taxas das taxas de aprendizagem adaptativa por coordenadas. O problema é que Adagrad acumula os quadrados do gradiente $\mathbf{g}_t$ em um vetor de estado $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$. Como resultado, $\mathbf{s}_t$ continua crescendo sem limites devido à falta de normalização, essencialmente linearmente conforme o algoritmo converge.

Uma maneira de corrigir esse problema seria usar $\mathbf{s}_t / t$. Para distribuições razoáveis de $\mathbf{g}_t$, isso convergirá. Infelizmente, pode levar muito tempo até que o comportamento do limite comece a importar, pois o procedimento lembra a trajetória completa dos valores. Uma alternativa é usar uma média de vazamento da mesma forma que usamos no método de momentum, ou seja, $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma) \mathbf{g}_t^2$ para algum parâmetro $\gamma > 0$. Manter todas as outras partes inalteradas resulta em RMSProp.

11.8.1 O Algoritmo

Vamos escrever as equações em detalhes.

$$\begin{aligned}
\mathbf{s}_t & \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\
\mathbf{x}_t & \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.
\end{aligned} \tag{11.8.1}$$

A constante $\epsilon > 0$ é normalmente definida como 10^{-6} para garantir que não soframos divisão por zero ou tamanhos de passos excessivamente grandes. Dada essa expansão, agora estamos livres para controlar a taxa de aprendizado η independentemente da escala que é aplicada por coordenada. Em termos de médias vazadas, podemos aplicar o mesmo raciocínio aplicado anteriormente no caso do método do momento. Expandindo a definição de $\mathbf{s}_t$ yields

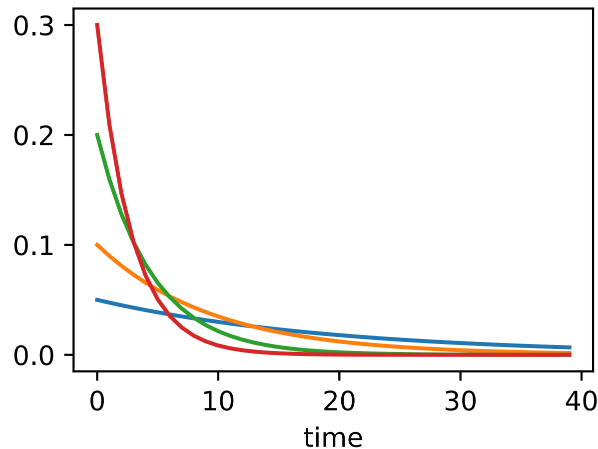
$$\begin{aligned}
\mathbf{s}_t & = (1 - \gamma) \mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\
& = (1 - \gamma) \left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots, \right).
\end{aligned} \tag{11.8.2}$$

Como antes em Section 11.6 usamos $1 + \gamma + \gamma^2 + \ldots, = \frac{1}{1-\gamma}$. Portanto, a soma dos pesos é normalizada para 1 com um tempo de meia-vida de uma observação de γ^{-1}. Vamos visualizar os pesos das últimas 40 etapas de tempo para várias opções de γ.

[121] https://discuss.d2l.ai/t/1072

```
import math
import torch
from d2l import torch as d2l
```

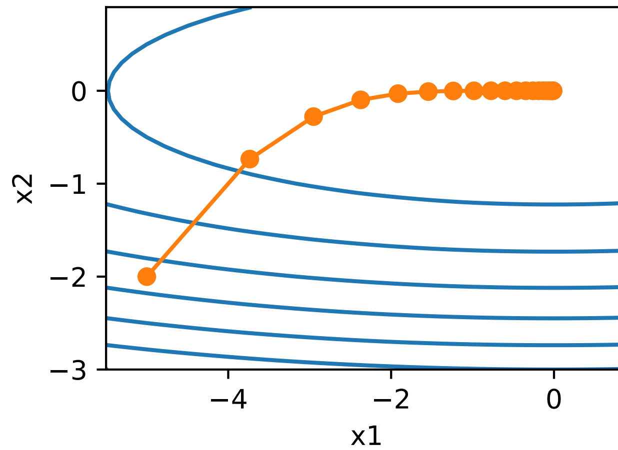
```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```



11.8.2 Implementação do zero

Como antes, usamos a função quadrática $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ para observar a trajetória de RMSProp. Lembre-se de que em Section 11.7, quando usamos o Adagrad com uma taxa de aprendizado de 0,4, as variáveis se moviam apenas muito lentamente nos estágios posteriores do algoritmo, pois a taxa de aprendizado diminuía muito rapidamente. Como η é controlado separadamente, isso não acontece com RMSProp.

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```



Em seguida, implementamos RMSProp para ser usado em uma rede profunda. Isso é igualmente simples.

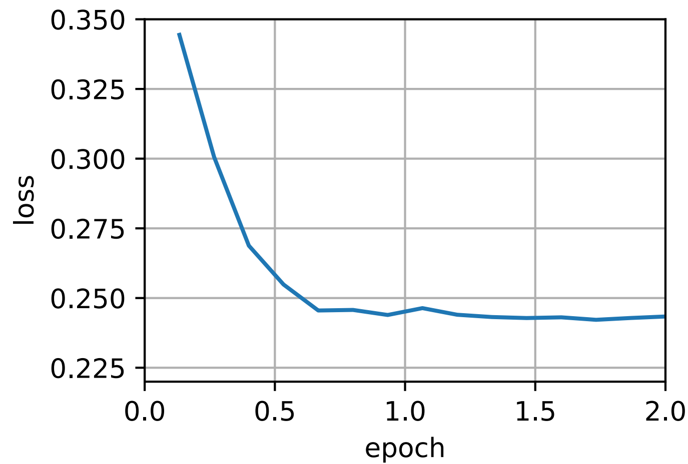
```
def init_rmsprop_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)
```

```
def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] = gamma * s + (1 - gamma) * torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

Definimos a taxa de aprendizado inicial como 0,01 e o termo de ponderação γ como 0,9. Ou seja, $\mathbf{s}$ agrega em média nas últimas $1/(1-\gamma) = 10$ observações do gradiente quadrado.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.243, 0.014 sec/epoch
```

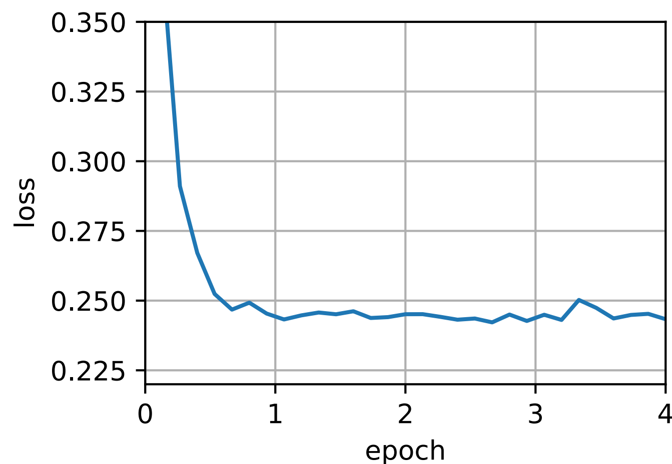


11.8.3 Implementação concisa

Como RMSProp é um algoritmo bastante popular, ele também está disponível na instância `Trainer`. Tudo o que precisamos fazer é instanciá-lo usando um algoritmo chamado `rmsprop`, atribuindo γ ao parâmetro `gamma1`.

```
trainer = torch.optim.RMSprop
d2l.train_concise_ch11(trainer, {'lr': 0.01, 'alpha': 0.9},
                       data_iter)
```

```
loss: 0.243, 0.013 sec/epoch
```



11.8.4 Sumário

- RMSProp é muito semelhante ao Adagrad na medida em que ambos usam o quadrado do gradiente para dimensionar os coeficientes.
- RMSProp compartilha com momentum a média que vaza. No entanto, RMSProp usa a técnica para ajustar o pré-condicionador do coeficiente.
- A taxa de aprendizagem precisa ser programada pelo experimentador na prática.
- O coeficiente γ determina quanto tempo o histórico é ao ajustar a escala por coordenada.

11.8.5 Exercícios

1. O que acontece experimentalmente se definirmos $\gamma = 1$? Por quê?
2. Gire o problema de otimização para minimizar $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. O que acontece com a convergência?
3. Experimente o que acontece com o RMSProp em um problema real de aprendizado de máquina, como o treinamento em Fashion-MNIST. Experimente diferentes opções para ajustar a taxa de aprendizagem.
4. Você gostaria de ajustar γ conforme a otimização progride? Quão sensível é o RMSProp a isso?

Discussão[122]

11.9 Adadelta

Adadelta é outra variante do AdaGrad (Section 11.7). A principal diferença reside no fato de que diminui a quantidade pela qual a taxa de aprendizagem é adaptável às coordenadas. Além disso, tradicionalmente é referido como não tendo uma taxa de aprendizagem, uma vez que usa a quantidade de mudança em si como calibração para mudanças futuras. O algoritmo foi proposto em (Zeiler, 2012). É bastante simples, dada a discussão de algoritmos anteriores até agora.

11.9.1 O Algoritmo

Em poucas palavras, Adadelta usa duas variáveis de estado, $\mathbf{s}_t$ para armazenar uma média de vazamento do segundo momento do gradiente e $\Delta\mathbf{x}_t$ para armazenar uma média de vazamento do segundo momento da mudança de parâmetros no próprio modelo. Observe que usamos a notação original e a nomenclatura dos autores para compatibilidade com outras publicações e implementações (não há outra razão real para usar variáveis gregas diferentes para indicar um parâmetro que serve ao mesmo propósito em momentum, Adagrad, RMSProp e Adadelta)

Aqui estão os detalhes técnicos do Adadelta. Dado que o parâmetro du jour é ρ, obtemos as seguintes atualizações vazadas de forma semelhante a Section 11.8:

$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1 - \rho) \mathbf{g}_t^2. \tag{11.9.1}$$

[122] https://discuss.d2l.ai/t/1074

A diferença para Section 11.8 é que realizamos atualizações com o gradiente redimensionado $\mathbf{g}_t'$, ou seja,

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{11.9.2}$$

Então, qual é o gradiente redimensionado $\mathbf{g}_t'$? Podemos calculá-lo da seguinte maneira:

$$\mathbf{g}_t' = \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \tag{11.9.3}$$

onde $\Delta \mathbf{x}_{t-1}$ é a média de vazamento dos gradientes redimensionados ao quadrado $\mathbf{g}_t'$. Inicializamos $\Delta \mathbf{x}_0$ para ser 0 e atualizamos em cada etapa com $\mathbf{g}_t'$, ou seja,

$$\Delta \mathbf{x}_t = \rho \Delta\mathbf{x}_{t-1} + (1 - \rho) {\mathbf{g}_t'}^2, \tag{11.9.4}$$

e ϵ (um pequeno valor como 10^{-5}) é adicionado para manter a estabilidade numérica.

11.9.2 Implementação

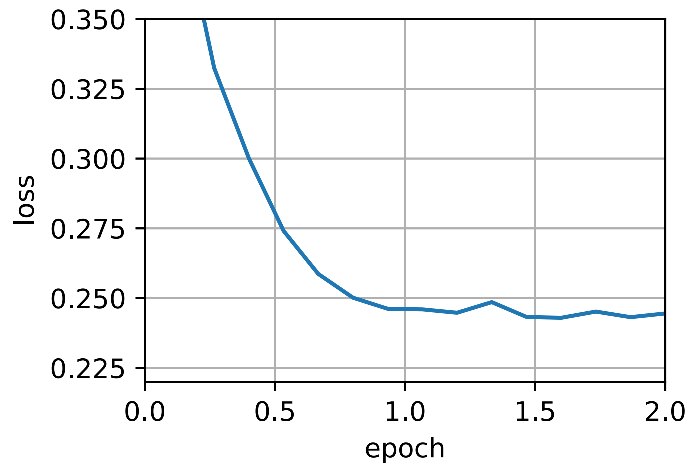
Adadelta precisa manter duas variáveis de estado para cada variável, $\mathbf{s}_t$ e $\Delta\mathbf{x}_t$. Isso produz a seguinte implementação.

```
%matplotlib inline
import torch
from d2l import torch as d2l


def init_adadelta_states(feature_dim):
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    delta_w, delta_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        with torch.no_grad():
            # In-place updates via [:]
            s[:] = rho * s + (1 - rho) * torch.square(p.grad)
            g = (torch.sqrt(delta + eps) / torch.sqrt(s + eps)) * p.grad
            p[:] -= g
            delta[:] = rho * delta + (1 - rho) * g * g
        p.grad.data.zero_()
```

Escolher $\rho = 0,9$ equivale a um tempo de meia-vida de 10 para cada atualização de parâmetro. Isso tende a funcionar muito bem. Obtemos o seguinte comportamento.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

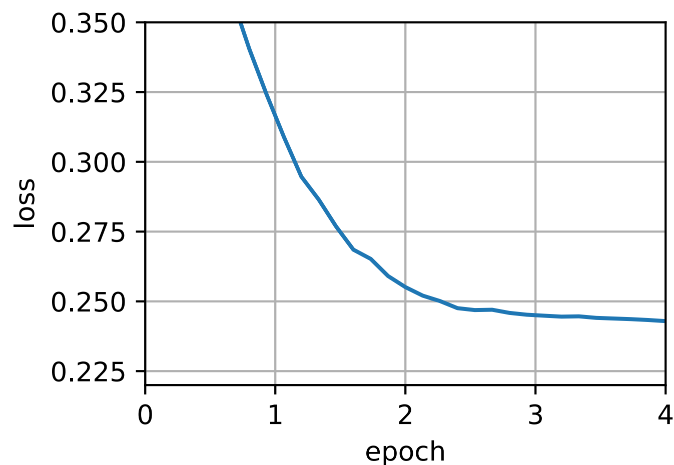
```
loss: 0.244, 0.015 sec/epoch
```



Para uma implementação concisa, simplesmente usamos o algoritmo `adadelta` da classe `Trainer`. Isso produz o seguinte one-liner para uma invocação muito mais compacta.

```
trainer = torch.optim.Adadelta
d2l.train_concise_ch11(trainer, {'rho': 0.9}, data_iter)
```

```
loss: 0.243, 0.013 sec/epoch
```



11.9.3 Sumário

- Adadelta não tem parâmetro de taxa de aprendizagem. Em vez disso, ele usa a taxa de mudança nos próprios parâmetros para adaptar a taxa de aprendizado.
- Adadelta requer duas variáveis de estado para armazenar os segundos momentos de gradiente e a mudança nos parâmetros.
- Adadelta usa médias vazadas para manter uma estimativa contínua das estatísticas apropriadas.

11.9.4 Exercícios

1. Ajuste o valor de ρ. O que acontece?
2. Mostre como implementar o algoritmo sem o uso de $\mathbf{g}'_t$. Por que isso pode ser uma boa ideia?
3. A taxa de aprendizagem Adadelta é realmente gratuita? Você conseguiu encontrar problemas de otimização que quebram o Adadelta?
4. Compare Adadelta com Adagrad e RMS prop para discutir seu comportamento de convergência.

Discussão[123]

11.10 Adam

Nas discussões que levaram a esta seção, encontramos várias técnicas para otimização eficiente. Vamos recapitulá-los em detalhes aqui:

- Vimos que Section 11.4 é mais eficaz do que Gradient Descent ao resolver problemas de otimização, por exemplo, devido à sua resiliência inerente a dados redundantes.
- Vimos que Section 11.5 proporciona eficiência adicional significativa decorrente da vetorização, usando conjuntos maiores de observações em um minibatch. Esta é a chave para um processamento paralelo eficiente em várias máquinas, várias GPUs e em geral.
- Section 11.6 adicionado um mecanismo para agregar um histórico de gradientes anteriores para acelerar a convergência.
- Section 11.7 usado por escala de coordenada para permitir um pré-condicionador computacionalmente eficiente.
- Section 11.8 desacoplado por escala de coordenada de um ajuste de taxa de aprendizagem.

Adam (Kingma & Ba, 2014) combina todas essas técnicas em um algoritmo de aprendizagem eficiente. Como esperado, este é um algoritmo que se tornou bastante popular como um dos algoritmos de otimização mais robustos e eficazes para uso no aprendizado profundo. Não é sem problemas, no entanto. Em particular, (Reddi et al., 2019) mostra que há situações em que Adam pode divergir devido a um controle de variação insuficiente. Em um trabalho de acompanhamento (Zaheer et al., 2018) propôs um hotfix para Adam, chamado Yogi, que trata dessas questões. Mais sobre isso mais tarde. Por enquanto, vamos revisar o algoritmo de Adam.

11.10.1 O Algoritmo

Um dos componentes principais de Adam é que ele usa médias móveis exponenciais ponderadas (também conhecidas como média com vazamento) para obter uma estimativa do momento e também do segundo momento do gradiente. Ou seja, ele usa as variáveis de estado

$$\begin{aligned}
\mathbf{v}_t & \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\
\mathbf{s}_t & \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.
\end{aligned} \tag{11.10.1}$$

Aqui β_1 e β_2 são parâmetros de ponderação não negativos. As escolhas comuns para eles são $\beta_1 = 0.9$ e $\beta_2 = 0.999$. Ou seja, a estimativa da variância se move *muito mais lentamente* do que

[123] https://discuss.d2l.ai/t/1076

o termo de momentum. Observe que se inicializarmos $\mathbf{v}_0 = \mathbf{s}_0 = 0$, teremos uma quantidade significativa de tendência inicialmente para valores menores. Isso pode ser resolvido usando o fato de que $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$ para normalizar os termos novamente. Correspondentemente, as variáveis de estado normalizadas são fornecidas por

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}. \tag{11.10.2}$$

Armados com as estimativas adequadas, podemos agora escrever as equações de atualização. Primeiro, nós redimensionamos o gradiente de uma maneira muito semelhante à do RMSProp para obter

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{11.10.3}$$

Ao contrário de RMSProp, nossa atualização usa o momento $\hat{\mathbf{v}}_t$ em vez do gradiente em si. Além disso, há uma pequena diferença estética, pois o redimensionamento acontece usando $\frac{1}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$ em vez de $\frac{1}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}}$. O primeiro funciona sem dúvida um pouco melhor na prática, daí o desvio de RMSProp. Normalmente escolhemos $\epsilon = 10^{-6}$ para uma boa troca entre estabilidade numérica e fidelidade.

Agora temos todas as peças no lugar para computar as atualizações. Isso é um pouco anticlimático e temos uma atualização simples do formulário

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{11.10.4}$$

Revendo o projeto de Adam, sua inspiração é clara. Momentum e escala são claramente visíveis nas variáveis de estado. Sua definição um tanto peculiar nos força a termos de debias (isso poderia ser corrigido por uma inicialização ligeiramente diferente e condição de atualização). Em segundo lugar, a combinação de ambos os termos é bastante direta, dado o RMSProp. Por último, a taxa de aprendizagem explícita η nos permite controlar o comprimento do passo para tratar de questões de convergência.

11.10.2 Implementação

Implementar Adam do zero não é muito assustador. Por conveniência, armazenamos o contador de intervalos de tempo t no dicionário de `hiperparâmetros`. Além disso, tudo é simples.

```
%matplotlib inline
import torch
from d2l import torch as d2l


def init_adam_states(feature_dim):
    v_w, v_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
```

(continues on next page)

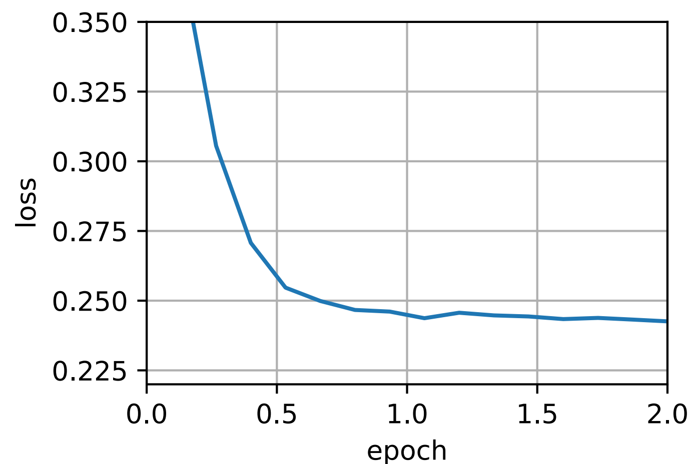
(continued from previous page)

```
            s[:] = beta2 * s + (1 - beta2) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

Estamos prontos para usar Adam para treinar o modelo. Usamos uma taxa de aprendizado de $\eta = 0,01$.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

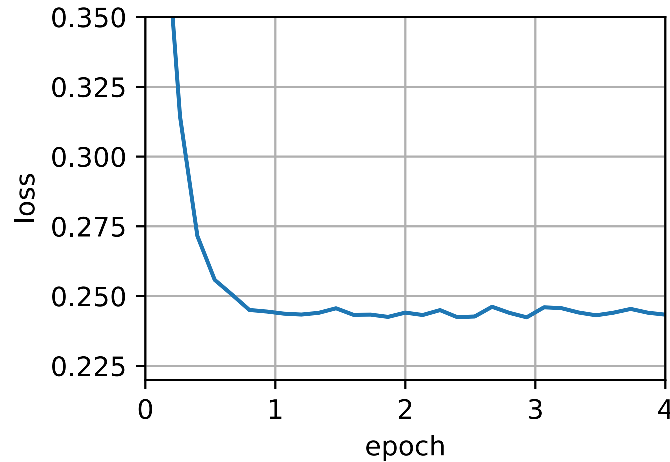
```
loss: 0.243, 0.015 sec/epoch
```



Uma implementação mais concisa é direta, pois `adam` é um dos algoritmos fornecidos como parte da biblioteca de otimização `trainer` Gluon. Portanto, só precisamos passar os parâmetros de configuração para uma implementação no Gluon.

```
trainer = torch.optim.Adam
d2l.train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.243, 0.014 sec/epoch
```



11.10.3 Yogi

Um dos problemas de Adam é que ele pode falhar em convergir mesmo em configurações convexas quando a estimativa do segundo momento em $\mathbf{s}_t$ explode. Como uma correção (Zaheer et al., 2018) propôs uma atualização refinada (e inicialização) para $\mathbf{s}_t$. Para entender o que está acontecendo, vamos reescrever a atualização do Adam da seguinte maneira:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \left(\mathbf{g}_t^2 - \mathbf{s}_{t-1}\right). \tag{11.10.5}$$

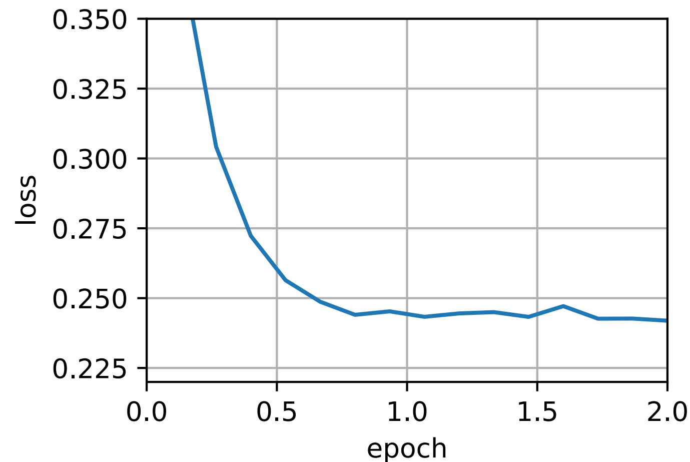
Sempre que $\mathbf{g}_t^2$ tem alta variância ou as atualizações são esparsas, $\mathbf{s}_t$ pode esquecer os valores anteriores muito rapidamente. Uma possível solução para isso é substituir $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ by $\mathbf{g}_t^2 \odot \mathop{\mathrm{sgn}}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$. Agora, a magnitude da atualização não depende mais da quantidade de desvio. Isso produz as atualizações Yogi

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 \odot \mathop{\mathrm{sgn}}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \tag{11.10.6}$$

Os autores, além disso, aconselham inicializar o momento em um lote inicial maior, em vez de apenas uma estimativa pontual inicial. Omitimos os detalhes, pois eles não são relevantes para a discussão e, mesmo sem essa convergência, ela permanece muito boa.

```python
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = s + (1 - beta2) * torch.sign(
                torch.square(p.grad) - s) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.242, 0.016 sec/epoch
```



11.10.4 Sumário

- Adam combina recursos de muitos algoritmos de otimização em uma regra de atualização bastante robusta.
- Criado com base no RMSProp, Adam também usa EWMA no gradiente estocástico do minibatch.
- Adam usa a correção de polarização para ajustar para uma inicialização lenta ao estimar o momentum e um segundo momento.
- Para gradientes com variação significativa, podemos encontrar problemas de convergência. Eles podem ser corrigidos usando minibatches maiores ou mudando para uma estimativa melhorada para $\mathbf{s}_t$. Yogi oferece essa alternativa.

11.10.5 Exercícios

1. Ajuste a taxa de aprendizagem e observe e analise os resultados experimentais.
2. Você pode reescrever atualizações de momentum e segundo momento de forma que não exija correção de viés?
3. Por que você precisa reduzir a taxa de aprendizado η conforme convergimos?
4. Tentar construir um caso para o qual Adam diverge e Yogi converge?

Discussão[124]

[124] https://discuss.d2l.ai/t/1078

11.11 Programação da taxa de aprendizagem

Até agora, focamos principalmente na otimização de *algoritmos* para como atualizar os vetores de peso, em vez de na *taxa* na qual eles estão sendo atualizados. No entanto, ajustar a taxa de aprendizagem é frequentemente tão importante quanto o algoritmo real. Existem vários aspectos a considerar:

- Obviamente, a *magnitude* da taxa de aprendizagem é importante. Se for muito grande, a otimização diverge; se for muito pequena, leva muito tempo para treinar ou terminamos com um resultado abaixo do ideal. Vimos anteriormente que o número da condição do problema é importante (consulte, por exemplo, Section 11.6 para obter detalhes). Intuitivamente, é a proporção da quantidade de mudança na direção menos sensível em relação à mais sensível.
- Em segundo lugar, a taxa de degradação é tão importante. Se a taxa de aprendizado permanecer alta, podemos simplesmente acabar saltando em torno do mínimo e, portanto, não atingir a otimização. Section 11.5 discutiu isso com alguns detalhes e analisamos as garantias de desempenho em Section 11.4. Resumindo, queremos que a taxa diminua, mas provavelmente mais lentamente do que $\mathcal{O}(t^{-\frac{1}{2}})$, o que seria uma boa escolha para problemas convexos.
- Outro aspecto igualmente importante é a *inicialização*. Isso se refere a como os parâmetros são definidos inicialmente (revise Section 4.8 para detalhes) e também como eles evoluem inicialmente. Isso tem o nome de *aquecimento*, ou seja, a rapidez com que começamos a nos mover em direção à solução inicialmente. Etapas grandes no início podem não ser benéficas, em particular porque o conjunto inicial de parâmetros é aleatório. As instruções iniciais de atualização também podem ser bastante insignificantes.
- Por último, há uma série de variantes de otimização que realizam ajustes de taxa de aprendizagem cíclica. Isso está além do escopo do capítulo atual. Recomendamos que o leitor analise os detalhes em (Izmailov et al., 2018), por exemplo, como obter melhores soluções calculando a média de um caminho inteiro de parâmetros.

Dado o fato de que são necessários muitos detalhes para gerenciar as taxas de aprendizado, a maioria dos frameworks de aprendizado profundo tem ferramentas para lidar com isso automaticamente. No capítulo atual, revisaremos os efeitos que diferentes programações têm na precisão e também mostraremos como isso pode ser gerenciado de forma eficiente por meio de um *programador de taxa de aprendizagem*.

11.11.1 Problema Amostra

Começamos com um problema de brinquedo que é barato o suficiente para ser computado facilmente, mas suficientemente não trivial para ilustrar alguns dos principais aspectos. Para isso, escolhemos uma versão ligeiramente modernizada do LeNet (`relu` em vez de ativação `sigmoid`, MaxPooling em vez de AveragePooling), aplicado ao Fashion-MNIST. Além disso, hibridamos a rede para desempenho. Como a maior parte do código é padrão, apenas apresentamos o básico sem uma discussão mais detalhada. Veja Chapter 6 para uma atualização conforme necessário.

```
%matplotlib inline
import math
import torch
```

(continues on next page)

(continued from previous page)

```
from torch import nn
from torch.optim import lr_scheduler
from d2l import torch as d2l


def net_fn():
    class Reshape(nn.Module):
        def forward(self, x):
            return x.view(-1,1,28,28)

    model = torch.nn.Sequential(
        Reshape(),
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
        nn.Linear(120, 84), nn.ReLU(),
        nn.Linear(84, 10))

    return model

loss = nn.CrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to `d2l.train_ch6` defined in the
# lenet section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
          scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))
```

(continues on next page)

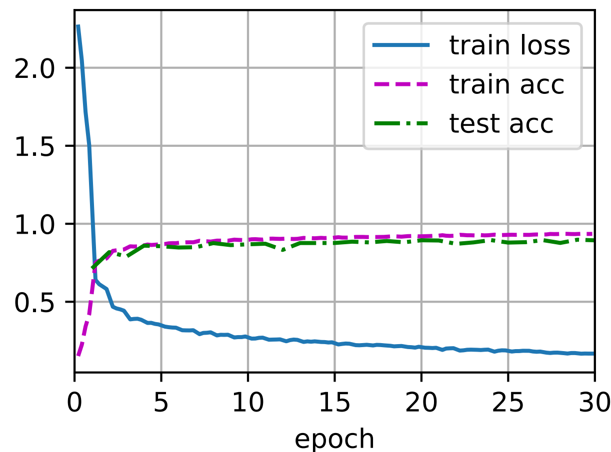
(continued from previous page)

```
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

        if scheduler:
            if scheduler.__module__ == lr_scheduler.__name__:
                # Using PyTorch In-Built scheduler
                scheduler.step()
            else:
                # Using custom defined scheduler
                for param_group in trainer.param_groups:
                    param_group['lr'] = scheduler(epoch)

    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

Vamos dar uma olhada no que acontece se invocarmos esse algoritmo com configurações padrão, como uma taxa de aprendizado de $0,3$ e treinar por 30 iterações. Observe como a precisão do treinamento continua aumentando enquanto o progresso em termos de precisão do teste para além de um ponto. A lacuna entre as duas curvas indica sobreajuste.

```
lr, num_epochs = 0.3, 30
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 0.168, train acc 0.935, test acc 0.894
```



11.11.2 Agendadores

Uma forma de ajustar a taxa de aprendizagem é defini-la explicitamente em cada etapa. Isso é convenientemente alcançado pelo método `set_learning_rate`. Poderíamos ajustá-lo para baixo após cada época (ou mesmo após cada minibatch), por exemplo, de uma maneira dinâmica em resposta a como a otimização está progredindo.

```
lr = 0.1
trainer.param_groups[0]["lr"] = lr
print(f'learning rate is now {trainer.param_groups[0]["lr"]:.2f}')
```

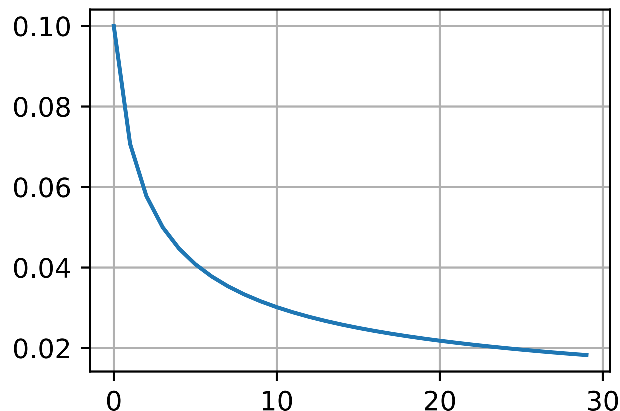
```
learning rate is now 0.10
```

De maneira mais geral, queremos definir um planejador. Quando chamado com o número de atualizações, ele retorna o valor apropriado da taxa de aprendizado. Vamos definir um simples que define a taxa de aprendizagem para $\eta = \eta_0 (t+1)^{-\frac{1}{2}}$.

```
class SquareRootScheduler:
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

Vamos representar graficamente seu comportamento em uma faixa de valores.

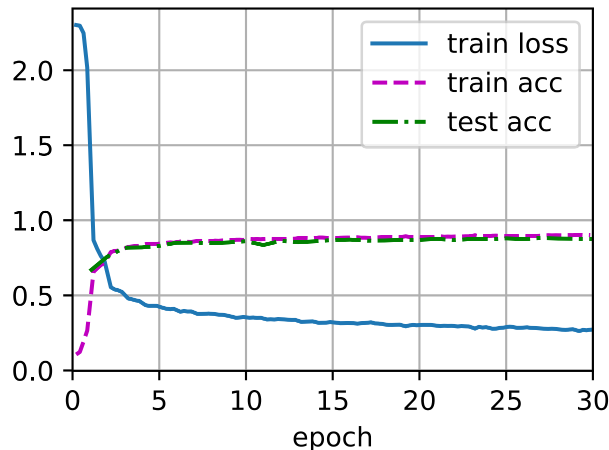
```
scheduler = SquareRootScheduler(lr=0.1)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Agora vamos ver como isso funciona para o treinamento no Fashion-MNIST. Simplesmente fornecemos o escalonador como um argumento adicional para o algoritmo de treinamento.

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.273, train acc 0.901, test acc 0.876
```



Isso funcionou um pouco melhor do que antes. Duas coisas se destacam: a curva era um pouco mais suave do que antes. Em segundo lugar, houve menos ajuste excessivo. Infelizmente, não é uma questão bem resolvida por que certas estratégias levam a menos ajustes excessivos em *teoria*. Há algum argumento de que um tamanho de passo menor levará a parâmetros mais próximos de zero e, portanto, mais simples. No entanto, isso não explica o fenômeno inteiramente, uma vez que não paramos realmente cedo, mas simplesmente reduzimos a taxa de aprendizagem suavemente.

11.11.3 Políticas

Embora não possamos cobrir toda a variedade de programadores de taxa de aprendizagem, tentamos fornecer uma breve visão geral das políticas populares abaixo. As escolhas comuns são decaimento polinomial e esquemas constantes por partes. Além disso, verificou-se que as programações de taxa de aprendizado de cosseno funcionam bem empiricamente em alguns problemas. Por último, em alguns problemas, é benéfico aquecer o otimizador antes de usar altas taxas de aprendizado.

Planejador de Fator

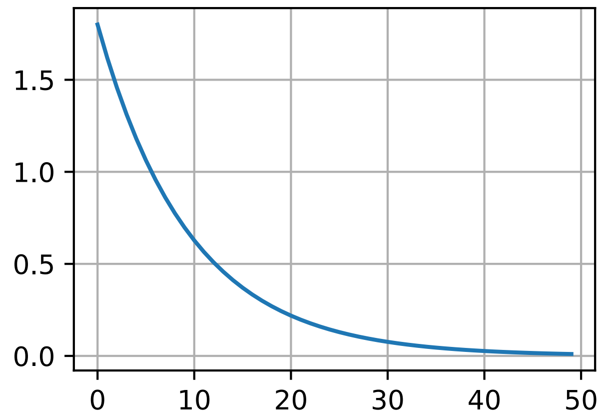
Uma alternativa para um decaimento polinomial seria um multiplicativo, que é $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ para $\alpha \in (0, 1)$. Para evitar que a taxa de aprendizagem diminua além de um limite inferior razoável, a equação de atualização é frequentemente modificada para $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$.

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr
```

(continues on next page)

(continued from previous page)

```
scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(torch.arange(50), [scheduler(t) for t in range(50)])
```

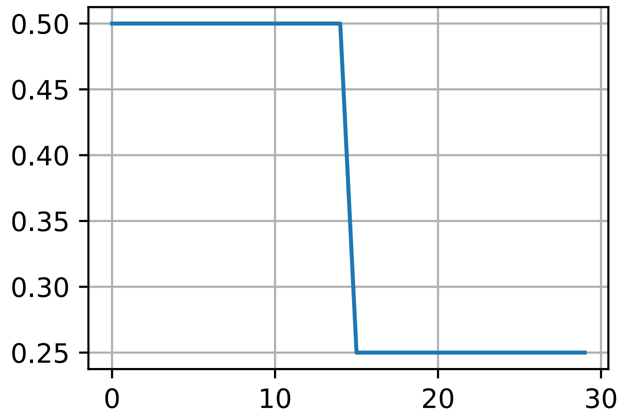


Isso também pode ser realizado por um agendador embutido no MXNet por meio do objeto `lr_scheduler.FactorScheduler`. Leva mais alguns parâmetros, como período de aquecimento, modo de aquecimento (linear ou constante), o número máximo de atualizações desejadas, etc .; No futuro, usaremos os agendadores integrados conforme apropriado e apenas explicaremos sua funcionalidade aqui. Conforme ilustrado, é bastante simples construir seu próprio agendador, se necessário.

Planejador de Fatores Multiplos

Uma estratégia comum para treinar redes profundas é manter a taxa de aprendizado constante e diminuí-la em uma determinada quantidade de vez em quando. Ou seja, dado um conjunto de vezes quando diminuir a taxa, como $s = \{5, 10, 20\}$ diminuir $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ sempre que $t \in s$. Supondo que os valores sejam reduzidos à metade em cada etapa, podemos implementar isso da seguinte maneira.

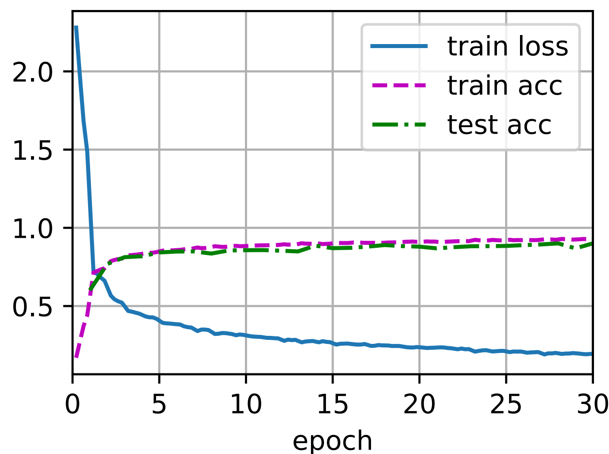
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
scheduler = lr_scheduler.MultiStepLR(trainer, milestones=[15, 30], gamma=0.5)

def get_lr(trainer, scheduler):
    lr = scheduler.get_last_lr()[0]
    trainer.step()
    scheduler.step()
    return lr

d2l.plot(torch.arange(num_epochs), [get_lr(trainer, scheduler)
                                  for t in range(num_epochs)])
```



A intuição por trás dessa programação de taxa de aprendizado constante por partes é que se permite que a otimização prossiga até que um ponto estacionário seja alcançado em termos de distribuição de vetores de peso. Então (e somente então) diminuímos a taxa de forma a obter um proxy de maior qualidade para um bom mínimo local. O exemplo abaixo mostra como isso pode produzir soluções cada vez melhores.

```
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.194, train acc 0.926, test acc 0.901
```



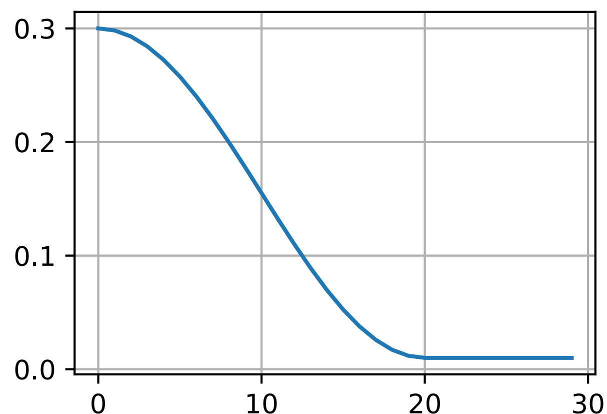
Programador de Cosseno

Uma heurística bastante desconcertante foi proposta por (Loshchilov & Hutter, 2016). Baseia-se na observação de que podemos não querer diminuir a taxa de aprendizado muito drasticamente no início e, além disso, podemos querer "refinar" a solução no final usando uma taxa de aprendizado muito pequena. Isso resulta em um esquema semelhante ao cosseno com a seguinte forma funcional para taxas de aprendizado no intervalo $t \in [0, T]$.

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} \left(1 + \cos(\pi t/T)\right) \tag{11.11.1}$$

Aqui η_0 é a taxa de aprendizado inicial, η_T é a taxa alvo no momento T. Além disso, para $t > T$ simplesmente fixamos o valor em η_T sem aumentá-lo novamente. No exemplo a seguir, definimos a etapa de atualização máxima $T = 20$.

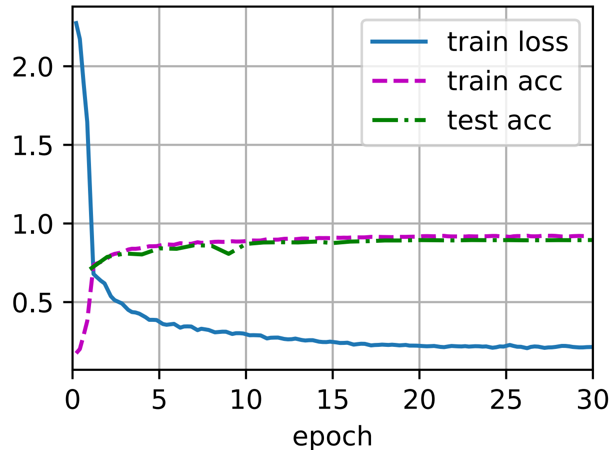
```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
               warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr

scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



No contexto da visão computacional, este cronograma *pode* levar a melhores resultados. Observe, entretanto, que tais melhorias não são garantidas (como pode ser visto abaixo).

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.215, train acc 0.920, test acc 0.895
```

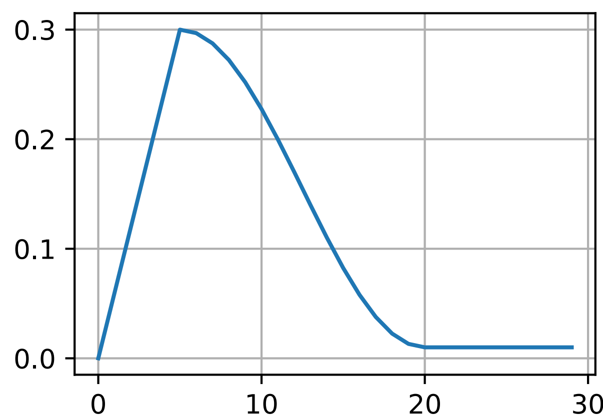


Aquecimento

Em alguns casos, inicializar os parâmetros não é suficiente para garantir uma boa solução. Isso é particularmente um problema para alguns projetos de rede avançados que podem levar a problemas de otimização instáveis. Poderíamos resolver isso escolhendo uma taxa de aprendizado suficientemente pequena para evitar divergências no início. Infelizmente, isso significa que o progresso é lento. Por outro lado, uma grande taxa de aprendizado inicialmente leva à divergência.

Uma solução bastante simples para esse dilema é usar um período de aquecimento durante o qual a taxa de aprendizado *aumenta* até seu máximo inicial e esfriar a taxa até o final do processo de otimização. Para simplificar, normalmente usa-se um aumento linear para esse propósito. Isso leva a uma programação do formulário indicado abaixo.

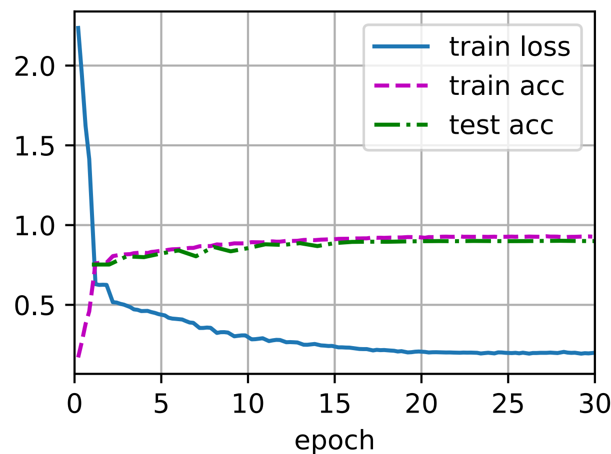
```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Observe que a rede converge melhor inicialmente (em particular observe o desempenho durante as primeiras 5 épocas).

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.197, train acc 0.928, test acc 0.900
```



O aquecimento pode ser aplicado a qualquer agendador (não apenas cosseno). Para uma discussão mais detalhada sobre cronogramas de taxas de aprendizagem e muitos outros experimentos, consulte também (Gotmare et al., 2018). Em particular, eles descobrem que uma fase de aquecimento limita a quantidade de divergência de parâmetros em redes muito profundas. Isso faz sentido intuitivamente, pois esperaríamos divergências significativas devido à inicialização aleatória nas partes da rede que levam mais tempo para progredir no início.

11.11.4 Sumário

- Diminuir a taxa de aprendizado durante o treinamento pode levar a uma maior precisão e (o que é mais desconcertante) à redução do ajuste excessivo do modelo.
- Uma diminuição por partes da taxa de aprendizagem sempre que o progresso atinge um platô é eficaz na prática. Essencialmente, isso garante que convergiremos de forma eficiente para uma solução adequada e só então reduziremos a variação inerente dos parâmetros, reduzindo a taxa de aprendizagem.
- Os planejadores de cosseno são populares para alguns problemas de visão computacional. Veja, por exemplo, GluonCV[125] para detalhes de tal planejador.
- Um período de aquecimento antes da otimização pode evitar divergências.
- A otimização serve a vários propósitos no aprendizado profundo. Além de minimizar o objetivo do treinamento, diferentes escolhas de algoritmos de otimização e programação da taxa de aprendizagem podem levar a quantidades bastante diferentes de generalização e overfitting no conjunto de teste (para a mesma quantidade de erro de treinamento).

[125] http://gluon-cv.mxnet.io

11.11.5 Exercícios

1. Experimente o comportamento de otimização para uma determinada taxa de aprendizagem fixa. Qual é o melhor modelo que você pode obter dessa forma?
2. Como a convergência muda se você alterar o expoente da diminuição na taxa de aprendizado? Use `PolyScheduler` para sua conveniência nos experimentos.
3. Aplique o programador de cosseno a grandes problemas de visão computacional, por exemplo, treinamento ImageNet. Como isso afeta o desempenho em relação a outros agendadores?
4. Quanto tempo deve durar o aquecimento?
5. Você pode conectar otimização e amostragem? Comece usando os resultados de (Welling & Teh, 2011) em Stochastic Gradient Langevin Dynamics.

Discussão[126]

[126] https://discuss.d2l.ai/t/1080

12 | Desempenho Computacional

No *deep learning* os *datasets* geralmente são grandes e a computação do modelo é complexa. Portanto, estamos sempre muito preocupados com o desempenho computacional. Este capítulo enfocará os fatores importantes que afetam o desempenho da computação: programação imperativa, programação simbólica, programação assíncrona, computação paralela automática e computação multi-GPU. Ao estudar este capítulo, você deve ser capaz de melhorar ainda mais o desempenho de computação dos modelos que foram implementados nos capítulos anteriores, por exemplo, reduzindo o tempo de treinamento do modelo sem afetar a precisão do modelo.

12.1 Compiladores e Interpretadores

Até agora, este livro se concentrou na programação imperativa, que faz uso de instruções como `print`,`+`ou`if` para alterar o estado de um programa. Considere o seguinte exemplo de um programa imperativo simples.

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

```
10
```

Python é uma linguagem interpretada. Ao avaliar `fancy_func` ele realiza as operações que compõem o corpo da função *em sequência*. Ou seja, ele avaliará `e = add (a, b)` e armazenará os resultados como a variável `e`, alterando assim o estado do programa. As próximas duas instruções `f = add (c, d)` e `g = add (e, f)` serão executadas de forma semelhante, realizando adições e armazenando os resultados como variáveis. Fig. 12.1.1 ilustra o fluxo de dados.

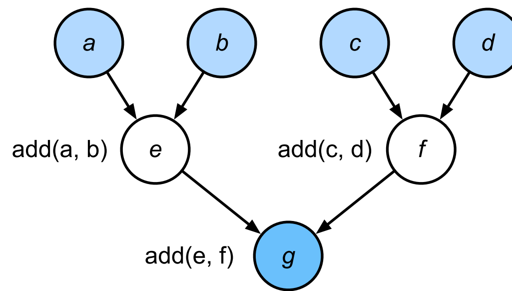


Fig. 12.1.1: Fluxo de dados em um programa imperativo.

Embora a programação imperativa seja conveniente, pode ser ineficiente. Por um lado, mesmo se a função `add` for repetidamente chamada em`fancy_func`, Python executará as três chamadas de função individualmente. Se elas forem executadas, digamos, em uma GPU (ou mesmo em várias GPUs), a sobrecarga decorrente do interpretador Python pode se tornar excessiva. Além disso, ele precisará salvar os valores das variáveis `e` e`f` até que todas as instruções em `fancy_func` tenham sido executadas. Isso ocorre porque não sabemos se as variáveis `e` e`f` serão usadas por outras partes do programa após as instruções `e = add (a, b)` e `f = add (c, d)` serem executadas.

12.1.1 Programação Simbólica

Considere a alternativa de programação simbólica, em que a computação geralmente é realizada apenas depois que o processo foi totalmente definido. Essa estratégia é usada por vários frameworks de aprendizado profundo, incluindo Theano, Keras e TensorFlow (os dois últimos adquiriram extensões imperativas). Geralmente envolve as seguintes etapas:

1. Definir as operações a serem executadas.
2. Compilar as operações em um programa executável.
3. Fornecer as entradas necessárias e chamar o programa compilado para execução.

Isso permite uma quantidade significativa de otimização. Em primeiro lugar, podemos pular o interpretador Python em muitos casos, removendo assim um gargalo de desempenho que pode se tornar significativo em várias GPUs rápidas emparelhadas com um único thread Python em uma CPU. Em segundo lugar, um compilador pode otimizar e reescrever o código acima em `print ((1 + 2) + (3 + 4))` ou mesmo `print (10)`. Isso é possível porque um compilador consegue ver o código completo antes de transformá-lo em instruções de máquina. Por exemplo, ele pode liberar memória (ou nunca alocá-la) sempre que uma variável não for mais necessária. Ou pode transformar o código inteiramente em uma parte equivalente. Para ter uma ideia melhor, considere a seguinte simulação de programação imperativa (afinal, é Python) abaixo.

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
```

(continues on next page)

(continued from previous page)

```
    f = add(c, d)
    g = add(e, f)
    return g
'''

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

As diferenças entre a programação imperativa (interpretada) e a programação simbólica são as seguintes:

- A programação imperativa é mais fácil. Quando a programação imperativa é usada em Python, a maior parte do código é direta e fácil de escrever. Também é mais fácil depurar o código de programação imperativo. Isso ocorre porque é mais fácil obter e imprimir todos os valores de variáveis intermediárias relevantes ou usar as ferramentas de depuração integradas do Python.
- A programação simbólica é mais eficiente e fácil de portar. Isso torna mais fácil otimizar o código durante a compilação, além de ter a capacidade de portar o programa para um formato independente do Python. Isso permite que o programa seja executado em um ambiente não-Python, evitando, assim, quaisquer problemas de desempenho em potencial relacionados ao interpretador Python.

12.1.2 Programação Híbrida

Historicamente, a maioria das estruturas de aprendizagem profunda escolhe entre uma abordagem imperativa ou simbólica. Por exemplo, Theano, TensorFlow (inspirado no último), Keras e CNTK formulam modelos simbolicamente. Por outro lado, Chainer e PyTorch adotam uma abordagem imperativa. Um modo imperativo foi adicionado ao TensorFlow 2.0 (via Eager) e Keras em revisões posteriores.

Como mencionado acima, PyTorch é baseado em programação imperativa e usa gráficos de computação dinâmica. Em um esforço para alavancar a portabilidade e eficiência da programação simbólica, os desenvolvedores consideraram se seria possível combinar os benefícios de ambos os modelos de programação. Isso levou a um *torchscript* que permite aos usuários desenvolver e depurar usando programação imperativa pura, ao mesmo tempo em que têm a capacidade de

converter a maioria dos programas em programas simbólicos para serem executados quando o desempenho e a implantação de computação em nível de produto forem necessários.

12.1.3 Híbrido-Sequencial

A maneira mais fácil de ter uma ideia de como a hibridização funciona é considerar redes profundas com várias camadas. Convencionalmente, o interpretador Python precisará executar o código para todas as camadas para gerar uma instrução que pode então ser encaminhada para uma CPU ou GPU. Para um único dispositivo de computação (rápido), isso não causa grandes problemas. Por outro lado, se usarmos um servidor avançado de 8 GPUs, como uma instância AWS P3dn.24xlarge, o Python terá dificuldade para manter todas as GPUs ocupadas. O interpretador Python de thread único torna-se o gargalo aqui. Vamos ver como podemos resolver isso para partes significativas do código, substituindo `Sequential` por `HybridSequential`. Começamos definindo um MLP simples.

```
import torch
from torch import nn
from d2l import torch as d2l


# Factory for networks
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)
```

```
tensor([[ 0.0200, -0.0186]], grad_fn=<AddmmBackward>)
```

Ao converter o modelo usando a função `torch.jit.script`, podemos compilar e otimizar a computação no MLP. O resultado do cálculo do modelo permanece inalterado.

```
net = torch.jit.script(net)
net(x)
```

```
tensor([[ 0.0200, -0.0186]], grad_fn=<AddmmBackward>)
```

Convertendo o modelo usando `torch.jit.script` Isso parece quase bom demais para ser verdade: escreva o mesmo código de antes e simplesmente converta o modelo usando`torch.jit.script`. Assim que isso acontecer, a rede estará otimizada (faremos um benchmark do desempenho abaixo).

Aceleração por Hibridização

Para demonstrar a melhoria de desempenho obtida pela compilação, comparamos o tempo necessário para avaliar `net (x)` antes e depois da hibridização. Vamos definir uma função para medir esse tempo primeiro. Será útil ao longo do capítulo à medida que nos propomos a medir (e melhorar) o desempenho.

```
#@save
class Benchmark:
    def __init__(self, description='Done'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(f'{self.description}: {self.timer.stop():.4f} sec')
```

Agora podemos chamar a rede duas vezes, uma com e outra sem torchscript.

```
net = get_net()
with Benchmark('Without torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('With torchscript'):
    for i in range(1000): net(x)
```

```
Without torchscript: 1.9314 sec
With torchscript: 0.8800 sec
```

Conforme observado nos resultados acima, depois que uma instância nn.Sequential é criada com o script da função `torch.jit.script`, o desempenho da computação é aprimorado com o uso de programação simbólica.

Serialização

Um dos benefícios de compilar os modelos é que podemos serializar (salvar) o modelo e seus parâmetros no disco. Isso nos permite armazenar um modelo de maneira independente da linguagem de front-end de escolha. Isso nos permite implantar modelos treinados em outros dispositivos e usar facilmente outras linguagens de programação front-end. Ao mesmo tempo, o código geralmente é mais rápido do que o que pode ser alcançado na programação imperativa. Vamos ver o método save em ação.

```
net.save('my_mlp')
!ls -lh my_mlp*
```

```
-rw-r--r-- 1 jenkins jenkins 651K Dec 11 06:53 my_mlp
```

12.2 Computação Assíncrona

Os computadores de hoje são sistemas altamente paralelos, consistindo em vários núcleos de CPU (geralmente várias *threads* por núcleo), vários elementos de processamento por GPU e, muitas vezes, várias GPUs por dispositivo. Resumindo, podemos processar muitas coisas diferentes ao mesmo tempo, geralmente em dispositivos diferentes. Infelizmente, Python não é uma ótima maneira de escrever código paralelo e assíncrono, pelo menos não com alguma ajuda extra. Afinal, o Python é de thread único e é improvável que isso mude no futuro. Estruturas de aprendizado profundo, como MXNet e TensorFlow, utilizam um modelo de programação assíncrona para melhorar o desempenho (o PyTorch usa o próprio programador do Python, levando a uma compensação de desempenho diferente). Para PyTorch, por padrão, as operações de GPU são assíncronas. Quando você chama uma função que usa a GPU, as operações são enfileiradas no dispositivo específico, mas não necessariamente executadas até mais tarde. Isso nos permite executar mais cálculos em paralelo, incluindo operações na CPU ou outras GPUs.

Portanto, entender como a programação assíncrona funciona nos ajuda a desenvolver programas mais eficientes, reduzindo proativamente os requisitos computacionais e as dependências mútuas. Isso nos permite reduzir a sobrecarga de memória e aumentar a utilização do processador. Começamos importando as bibliotecas necessárias.

```
import os
import subprocess
import numpy
import torch
from torch import nn
from d2l import torch as d2l
```

12.2.1 Assincronismo via *Back-end*

Para um aquecimento, considere o seguinte problema brinquedo - queremos gerar uma matriz aleatória e multiplicá-la. Vamos fazer isso tanto no NumPy quanto no tensor PyTorch para ver a diferença. Observe que o `tensor` do PyTorch é definido em uma gpu.

```
# warmup for gpu computation
device = d2l.try_gpu()
a = torch.randn(size=(1000, 1000), device=device)
b = torch.mm(a, a)

with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('torch'):
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
```

```
numpy: 0.8409 sec
torch: 0.0011 sec
```

Isso é ordens de magnitude mais rápido. Pelo menos parece que sim. O produto de ponto Numpy é executado no processador cpu enquanto A multiplicação da matriz de Pytorch é executada no gpu e, portanto, o último espera-se que seja muito mais rápida. Mas a enorme diferença de tempo sugere que algo mais deve estar acontecendo. Por padrão, as operações da GPU são assíncronas no PyTorch. Forçando PyTorch a terminar todos os cálculos antes de retornar os programas, o que aconteceu anteriormente: o cálculo está sendo executado pelo backend enquanto o front-end retorna o controle para Python.

```
with d2l.Benchmark():
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
    torch.cuda.synchronize(device)
```

```
Done: 0.0023 sec
```

Em termos gerais, o PyTorch tem um *front-end* para interação direta com os usuários, por exemplo, via Python, bem como um *back-end* usado pelo sistema para realizar a computação. Conforme mostrado em: numref: `fig_frontends`, os usuários podem escrever programas PyTorch em várias linguagens de *front-end*, como Python e C ++. Independentemente da linguagem de programação de frontend usada, a execução de programas PyTorch ocorre principalmente no backend de implementações C ++. As operações emitidas pela linguagem do *front-end* são passadas para o *back-end* para execução. O *back-end* gerencia suas próprias threads que continuamente coletam e executam tarefas enfileiradas. Observe que para que isso funcione, o *back-end* deve ser capaz de rastrear as dependências entre várias etapas no gráfico computacional. Portanto, não é possível paralelizar operações que dependem umas das outras.

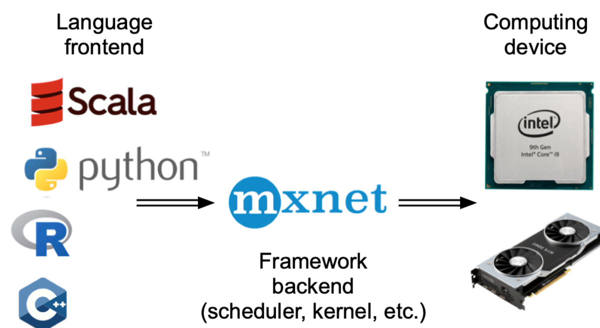


Fig. 12.2.1: Programação *Frontend*.

Vejamos outro exemplo brinquedo para entender um pouco melhor o grafo de dependência.

```
x = torch.ones((1, 2), device=device)
y = torch.ones((1, 2), device=device)
z = x * y + 2
z
```

```
tensor([[3., 3.]], device='cuda:0')
```

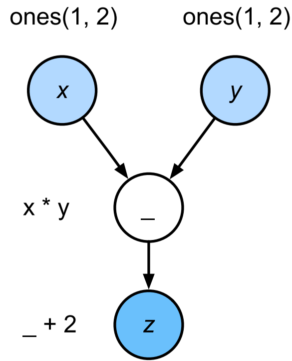


Fig. 12.2.2: Dependências.

O trecho de código acima também é ilustrado em Fig. 12.2.2. Sempre que a *thread* de *front-end* do Python executa uma das três primeiras instruções, ela simplesmente retorna a tarefa para a fila de *back-end*. Quando os resultados da última instrução precisam ser impressos, a *thread* de *front-end* do Python irá esperar que a*thread* de *back-end* do C ++ termine de calcular o resultado da variável z. Um benefício desse *design* é que a *thread* de *front-end* do Python não precisa realizar cálculos reais. Portanto, há pouco impacto no desempenho geral do programa, independentemente do desempenho do Python. Fig. 12.2.3 ilustra como *front-end* e *back-end* interagem.

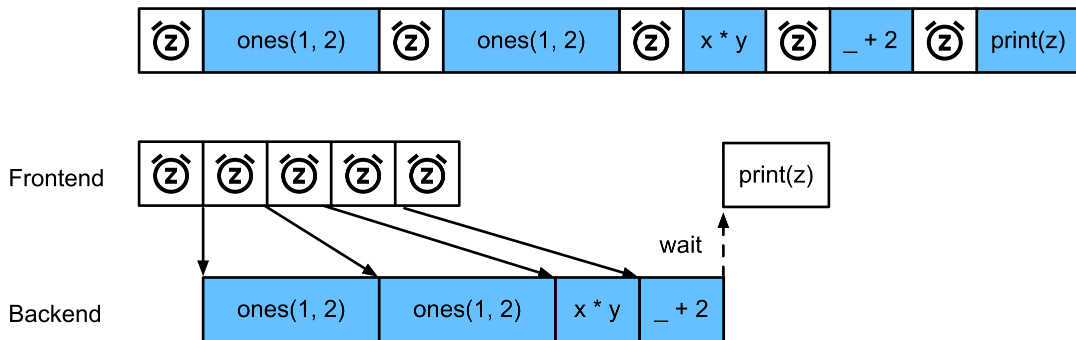


Fig. 12.2.3: Frontend and Backend.

12.2.2 Barreiras e Bloqueadores

Existem várias operações que forçam o Python a aguardar a conclusão: * Obviamente, `npx.waitall ()` espera até que todo o cálculo seja concluído, independentemente de quando as instruções de cálculo foram emitidas. Na prática, é uma má ideia usar este operador, a menos que seja absolutamente necessário, pois pode levar a um desempenho insatisfatório. * Se quisermos apenas esperar até que uma variável específica esteja disponível, podemos chamar `z.wait_to_read ()`. Nesse caso, os blocos MXNet retornam ao Python até que a variável z seja calculada. Outros cálculos podem continuar depois.

Vamos ver como isso funciona na prática:

Ambas as operações levam aproximadamente o mesmo tempo para serem concluídas. Além das operações de bloqueio óbvias, recomendamos que o leitor esteja ciente dos bloqueadores *implícitos*. Imprimir uma variável requer claramente que a variável esteja disponível e, portanto, é um bloqueador. Por último, as conversões para NumPy via `z.asnumpy ()` e conversões para escalares

via `z.item ()` estão bloqueando, uma vez que NumPy não tem noção de assincronismo. Ele precisa acessar os valores assim como a função `print`. Copiar pequenas quantidades de dados frequentemente do escopo do MXNet para NumPy e vice-versa pode destruir o desempenho de um código eficiente, uma vez que cada operação requer o gráfico computacional para avaliar todos os resultados intermediários necessários para obter o termo relevante *antes* que qualquer outra coisa possa ser feita.

12.2.3 Melhorando a Computação

Em um sistema altamente *multithread* (mesmo laptops regulares têm 4 threads ou mais e em servidores multithread esse número pode exceder 256), a sobrecarga das operações de agendamento pode se tornar significativa. É por isso que é altamente desejável que a computação e a programação ocorram de forma assíncrona e em paralelo. Para ilustrar o benefício de fazer isso, vamos ver o que acontece se incrementarmos uma variável em 1 várias vezes, tanto em sequência quanto de forma assíncrona. Simulamos a execução síncrona inserindo uma barreira `wait_to_read ()` entre cada adição.

Uma interação ligeiramente simplificada entre a *thread* de *front-end* Python e a *thread* de *back-end* C ++ pode ser resumida da seguinte maneira:

1. O *front-end* ordena que o *back-end* insira a tarefa de cálculo `y = x + 1` na fila.
2. O *back-end* então recebe as tarefas de computação da fila e executa os cálculos reais.
3. O *back-end* então retorna os resultados do cálculo para o *front-end*.

Suponha que as durações desses três estágios sejam t_1, t_2 e t_3, respectivamente. Se não usarmos a programação assíncrona, o tempo total necessário para realizar 1000 cálculos é de aproximadamente $1000(t_1 + t_2 + t_3)$. Se a programação assíncrona for usada, o tempo total gasto para realizar 1000 cálculos pode ser reduzido para $t_1 + 1000t_2 + t_3$ (assumindo $1000t_2 > 999t_1$), uma vez que o *front-end* não precisa esperar que o *back-end- retorne os resultados dos cálculos para cada*loop*.

12.2.4 Melhorando o *Footprint* de Memória

Imagine uma situação em que continuamos inserindo operações no *back-end*, executando o código Python no *front-end*. Por exemplo, o *front-end* pode inserir um grande número de tarefas de minibatch em um tempo muito curto. Afinal, se nenhum cálculo significativo acontecer no Python, isso pode ser feito rapidamente. Se cada uma dessas tarefas puder ser iniciada rapidamente ao mesmo tempo, isso pode causar um aumento no uso de memória. Dada uma quantidade finita de memória disponível nas GPUs (e mesmo nas CPUs), isso pode levar à contenção de recursos ou até mesmo travamentos do programa. Alguns leitores devem ter notado que as rotinas de treinamento anteriores faziam uso de métodos de sincronização como `item` ou mesmo `asnumpy`.

Recomendamos usar essas operações com cuidado, por exemplo, para cada minibatch, para equilibrar a eficiência computacional e a pegada de memória. Para ilustrar o que acontece, vamos implementar um *loop* de treinamento simples para uma rede profunda e medir seu consumo de memória e tempo. Abaixo está o gerador de dados simulado e a rede profunda.

Em seguida, precisamos de uma ferramenta para medir a pegada de memória de nosso código. Usamos uma chamada `ps` relativamente primitiva para fazer isso (observe que a última só funciona no Linux e MacOS). Para uma análise muito mais detalhada do que está acontecendo aqui, use,

por exemplo, o Nsight[127] da Nvidia ou o vTune[128] da Intel.

Antes de começarmos o teste, precisamos inicializar os parâmetros da rede e processar um lote. Caso contrário, seria complicado ver qual é o consumo de memória adicional. Veja `sec_deferred_init` para mais detalhes relacionados à inicialização.

Para garantir que não estouremos o *buffer* de tarefa no *back-end*, inserimos uma chamada `wait_to_read` para a função de perda no final de cada *loop*. Isso força a propagação direta a ser concluída antes que uma nova propagação direta seja iniciada. Observe que uma alternativa (possivelmente mais elegante) seria rastrear a perda em uma variável escalar e forçar uma barreira por meio da chamada de `item`.

Como vemos, o tempo dos minibatches se alinha muito bem com o tempo de execução geral do código de otimização. Além disso, o consumo de memória aumenta apenas ligeiramente. Agora vamos ver o que acontece se derrubarmos a barreira no final de cada minibatch.

Mesmo que o tempo para emitir instruções para o *back-end* seja uma ordem de magnitude menor, ainda precisamos realizar o cálculo. Consequentemente, uma grande quantidade de resultados intermediários não pode ser liberada e pode se acumular na memória. Embora isso não tenha causado nenhum problema no exemplo acima, pode muito bem ter resultado em situações de falta de memória quando não verificado em cenários do mundo real.

12.2.5 Resumo

- MXNet desacopla o *front-end* Python de um *back-end* de execução. Isso permite a rápida inserção assíncrona de comandos no *back-end* e o paralelismo associado.
- O assincronismo leva a uma interface bastante responsiva. No entanto, tenha cuidado para não sobrecarregar a fila de tarefas, pois isso pode levar ao consumo excessivo de memória.
- Recomenda-se sincronizar para cada minibatch para manter o *front-end* e o *back-end* aproximadamente sincronizados.
- Esteja ciente do fato de que as conversões do gerenciamento de memória do MXNet para Python forçarão o* back-end* a esperar até que a variável específica esteja pronta. `print`, `asnumpy` e `item` têm este efeito. Isso pode ser desejável, mas o uso sem carro da sincronização pode prejudicar o desempenho.
- Os fornecedores de chips oferecem ferramentas sofisticadas de análise de desempenho para obter uma visão muito mais detalhada da eficiência do *deep learning*.

12.2.6 Exercícios

1. Mencionamos acima que o uso de computação assíncrona pode reduzir a quantidade total de tempo necessária para realizar 1000 computações para $t_1 + 1000t_2 + t_3$. Por que temos que assumir $1000t_2 > 999t_1$ aqui?
2. Como você precisaria modificar o *loop* de treinamento se quisesse ter uma sobreposição de um minibatch cada? Ou seja, se você quiser garantir que o lote b_t termine antes que o lote b_{t+2} comece?

[127] https://developer.nvidia.com/nsight-compute-2019_5
[128] https://software.intel.com/en-us/vtune

3. O que pode acontecer se quisermos executar código em CPUs e GPUs simultaneamente? Você ainda deve insistir em sincronizar após cada minibatch ter sido emitido?
4. Meça a diferença entre `waitall` e`wait_to_read`. Dica: execute uma série de instruções e sincronize para um resultado intermediário.

12.3 Paralelismo Automático

O PyTorch constrói automaticamente gráficos computacionais no *back-end*. Usando um gráfico computacional, o sistema está ciente de todas as dependências e pode executar seletivamente várias tarefas não interdependentes em paralelo para melhorar a velocidade. Por exemplo, Fig. 12.2.2 em Section 12.2 inicializa duas variáveis independentemente. Consequentemente, o sistema pode optar por executá-las em paralelo.

Normalmente, um único operador usará todos os recursos computacionais em todas as CPUs ou em uma única GPU. Por exemplo, o operador `dot` usará todos os núcleos (e threads) em todas as CPUs, mesmo se houver vários processadores de CPU em uma única máquina. O mesmo se aplica a uma única GPU. Consequentemente, a paralelização não é tão útil em computadores de dispositivo único. Com vários dispositivos, as coisas são mais importantes. Embora a paralelização seja normalmente mais relevante entre várias GPUs, adicionar a CPU local aumentará um pouco o desempenho. Veja, por exemplo, (Hadjis et al., 2016) para um artigo que se concentra no treinamento de modelos de visão computacional combinando uma GPU e uma CPU. Com a conveniência de uma estrutura de paralelização automática, podemos atingir o mesmo objetivo em algumas linhas de código Python. De forma mais ampla, nossa discussão sobre computação paralela automática concentra-se na computação paralela usando CPUs e GPUs, bem como a paralelização de computação e comunicação. Começamos importando os pacotes e módulos necessários. Observe que precisamos de pelo menos duas GPUs para executar os experimentos nesta seção.

```
import torch
from d2l import torch as d2l
```

12.3.1 Computação Paralela em GPUs

Vamos começar definindo uma carga de trabalho de referência para testar - a função `run` abaixo realiza 10 multiplicações matriz-matriz no dispositivo de nossa escolha usando dados alocados em duas variáveis,`x_gpu1` e `x_gpu2`.

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])
```

Agora aplicamos a função aos dados. Para garantir que o cache não desempenhe um papel nos resultados, aquecemos os dispositivos realizando uma única passagem em cada um deles antes da medição. `torch.cuda.synchronize ()` espera que todos os kernels em todos os streams em um dispositivo CUDA sejam concluídos. Ele recebe um argumento `device`, o dispositivo para o qual precisamos sincronizar. Ele usa o dispositivo atual, fornecido por `current_device ()`, se o argumento do dispositivo for `None` (padrão).

```
run(x_gpu1)
run(x_gpu2)  # Warm-up all devices
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU 1 time'):
    run(x_gpu1)
    torch.cuda.synchronize(devices[0])

with d2l.Benchmark('GPU 2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])
```

```
GPU 1 time: 0.4915 sec
GPU 2 time: 0.4926 sec
```

Se removermos `torch.cuda.synchronize ()` entre as duas tarefas, o sistema fica livre para paralelizar a computação em ambos os dispositivos automaticamente.

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

```
GPU1 & GPU2: 0.4913 sec
```

No caso acima, o tempo total de execução é menor que a soma de suas partes, uma vez que o PyTorch programa automaticamente a computação em ambos os dispositivos GPU sem a necessidade de um código sofisticado em nome do usuário.

12.3.2 Computação Paralela e Comunicação

Em muitos casos, precisamos mover dados entre diferentes dispositivos, digamos, entre CPU e GPU, ou entre diferentes GPUs. Isso ocorre, por exemplo, quando queremos realizar a otimização distribuída onde precisamos agregar os gradientes em vários cartões aceleradores. Vamos simular isso computando na GPU e copiando os resultados de volta para a CPU.

```
def copy_to_cpu(x, non_blocking=False):
    return [y.to('cpu', non_blocking=non_blocking) for y in x]

with d2l.Benchmark('Run on GPU1'):
    y = run(x_gpu1)
    torch.cuda.synchronize()

with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    torch.cuda.synchronize()
```

```
Run on GPU1: 0.4916 sec
Copy to CPU: 2.3453 sec
```

Isso é um tanto ineficiente. Observe que já podemos começar a copiar partes de y para a CPU enquanto o restante da lista ainda está sendo calculado. Essa situação ocorre, por exemplo, quando calculamos o gradiente (*backprop*) em um minibatch. Os gradientes de alguns dos parâmetros estarão disponíveis antes dos outros. Portanto, é vantajoso começar a usar a largura de banda do barramento PCI-Express enquanto a GPU ainda está em execução. No PyTorch, várias funções como `to()` e `copy_()` admitem um argumento `non_blocking` explícito, que permite ao chamador ignorar a sincronização quando ela é desnecessária. Definir `non_blocking = True` nos permite simular este cenário.

```
with d2l.Benchmark('Run on GPU1 and copy to CPU'):
    y = run(x_gpu1)
    y_cpu = copy_to_cpu(y, True)
    torch.cuda.synchronize()
```

```
Run on GPU1 and copy to CPU: 1.6498 sec
```

O tempo total necessário para ambas as operações é (conforme esperado) significativamente menor do que a soma de suas partes. Observe que essa tarefa é diferente da computação paralela, pois usa um recurso diferente: o barramento entre a CPU e as GPUs. Na verdade, poderíamos computar em ambos os dispositivos e nos comunicar, tudo ao mesmo tempo. Como observado acima, há uma dependência entre computação e comunicação: `y[i]` deve ser calculado antes que possa ser copiado para a CPU. Felizmente, o sistema pode copiar `y[i-1]` enquanto calcula `y[i]` para reduzir o tempo total de execução.

Concluímos com uma ilustração do gráfico computacional e suas dependências para um MLP simples de duas camadas ao treinar em uma CPU e duas GPUs, conforme descrito em Fig. 12.3.1. Seria muito doloroso agendar o programa paralelo resultante disso manualmente. É aqui que é vantajoso ter um *back-end* de computação baseado em gráfico para otimização.

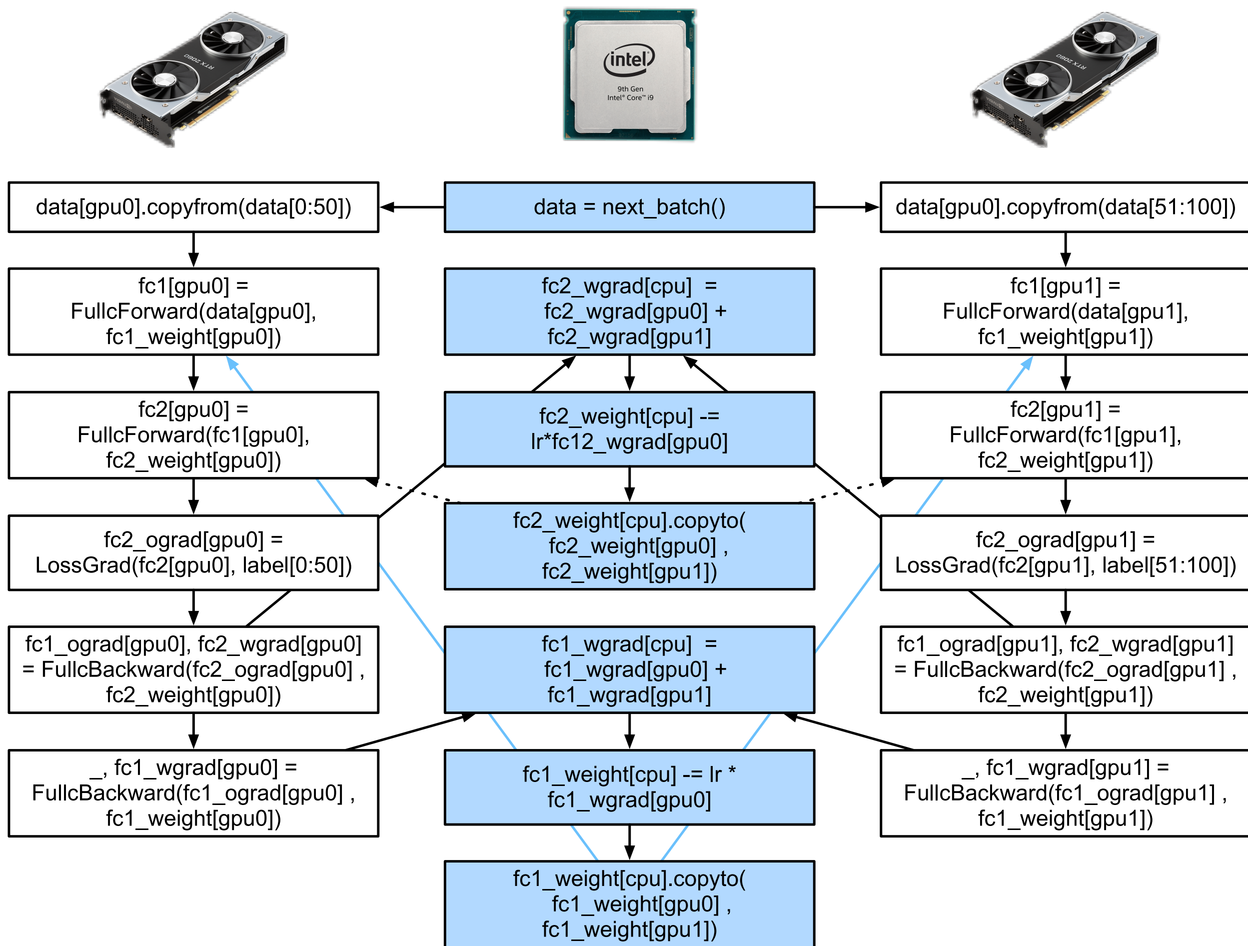


Fig. 12.3.1: MLP de duas camadas em uma CPU e 2 GPUs.

12.3.3 Resumo

- Os sistemas modernos têm uma variedade de dispositivos, como várias GPUs e CPUs. Eles podem ser usados em paralelo, de forma assíncrona.
- Os sistemas modernos também possuem uma variedade de recursos para comunicação, como PCI Express, armazenamento (normalmente SSD ou via rede) e largura de banda da rede. Eles podem ser usados em paralelo para eficiência máxima.
- O *back-end* pode melhorar o desempenho por meio de comunicação e computação paralela automática.

12.3.4 Exercícios

1. 10 operações foram realizadas na função `run` definida nesta seção. Não há dependências entre eles. Projete um experimento para ver se o MXNet irá executá-los automaticamente em paralelo.
2. Quando a carga de trabalho de um operador individual é suficientemente pequena, a paralelização pode ajudar até mesmo em uma única CPU ou GPU. Projete um experimento para verificar isso.
3. Projete um experimento que use computação paralela na CPU, GPU e comunicação entre os dois dispositivos.
4. Use um depurador como o Nsight da NVIDIA para verificar se o seu código é eficiente.
5. Projetar tarefas de computação que incluem dependências de dados mais complexas e executar experimentos para ver se você pode obter os resultados corretos enquanto melhora o desempenho.

Discussions[129]

12.4 *Hardware*

Construir sistemas com ótimo desempenho requer um bom entendimento dos algoritmos e modelos para capturar os aspectos estatísticos do problema. Ao mesmo tempo, também é indispensável ter pelo menos um mínimo de conhecimento do *hardware* subjacente. A seção atual não substitui um curso apropriado sobre design de *hardware* e sistemas. Em vez disso, pode servir como um ponto de partida para entender por que alguns algoritmos são mais eficientes do que outros e como obter um bom rendimento. Um bom design pode facilmente fazer uma diferença de ordem de magnitude e, por sua vez, pode fazer a diferença entre ser capaz de treinar uma rede (por exemplo, em uma semana) ou não (em 3 meses, perdendo o prazo) . Começaremos examinando os computadores. Em seguida, daremos zoom para observar com mais cuidado as CPUs e GPUs. Por último, diminuímos o zoom para revisar como vários computadores estão conectados em um centro de servidores ou na nuvem. Este não é um guia de compra de GPU. Para isto, veja Section 19.5. Uma introdução à computação em nuvem com AWS pode ser encontrada em Section 19.3.

Leitores impacientes podem se virar com Fig. 12.4.1. Ele foi retirado da postagem interativa[130] de Colin Scott , que oferece uma boa visão geral do progresso na última década. Os números originais são devidos à palestra de Stanford de 2010[131] de Jeff Dean. A discussão abaixo explica algumas das razões para esses números e como eles podem nos guiar no projeto de algoritmos. A discussão abaixo é de alto nível e superficial. Claramente, *não substitui* um curso adequado, mas apenas fornece informações suficientes para que um modelador estatístico tome decisões de projeto adequadas. Para uma visão geral aprofundada da arquitetura de computadores, recomendamos ao leitor (Hennessy & Patterson, 2011) ou um curso recente sobre o assunto, como o de Arste Asanovic[132].

[129] https://discuss.d2l.ai/t/1681
[130] https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html
[131] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf
[132] http://inst.eecs.berkeley.edu/~cs152/sp19/

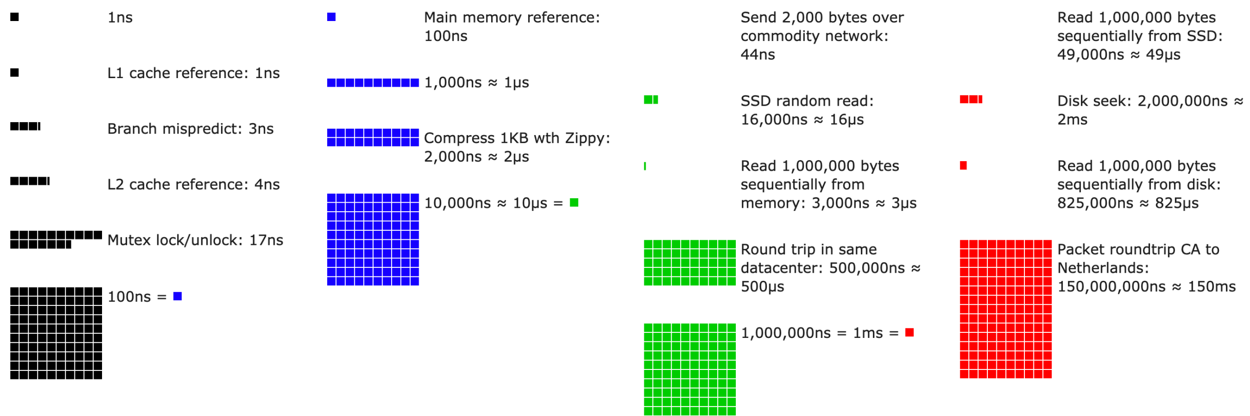


Fig. 12.4.1: Números de latência que todo programador deve conhecer.

12.4.1 Computadores

A maioria dos pesquisadores de aprendizagem profunda tem acesso a um computador com uma boa quantidade de memória, computação, alguma forma de acelerador, como uma GPU, ou múltiplos deles. Consiste em vários componentes principais:

- Um processador, também conhecido como CPU, que é capaz de executar os programas que fornecemos (além de executar um sistema operacional e muitas outras coisas), normalmente consistindo de 8 ou mais núcleos.
- Memória (RAM) para armazenar e recuperar os resultados da computação, como vetores de peso, ativações, geralmente dados de treinamento.
- Uma conexão de rede Ethernet (às vezes múltipla) com velocidades que variam de 1 Gbit / s a 100 Gbit / s (em servidores de ponta podem ser encontradas interconexões mais avançadas).
- Um barramento de expansão de alta velocidade (PCIe) para conectar o sistema a uma ou mais GPUs. Os servidores têm até 8 aceleradores, geralmente conectados em uma topologia avançada, os sistemas desktop têm 1-2, dependendo do orçamento do usuário e do tamanho da fonte de alimentação.
- O armazenamento durável, como um disco rígido magnético (HDD), estado sólido (SSD), em muitos casos conectado usando o barramento PCIe, fornece transferência eficiente de dados de treinamento para o sistema e armazenamento de pontos de verificação intermediários conforme necessário.

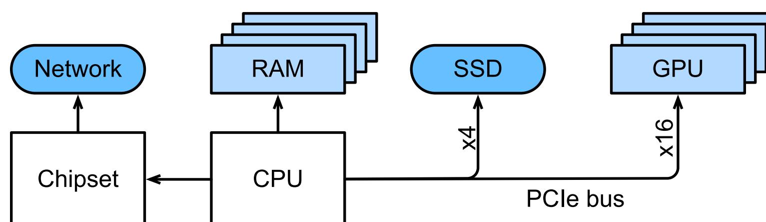


Fig. 12.4.2: Conectividade de componentes

Como Fig. 12.4.2 indica, a maioria dos componentes (rede, GPU, armazenamento) são conectados à CPU através do barramento PCI Express. Ele consiste em várias pistas diretamente conectadas à CPU. Por exemplo, o Threadripper 3 da AMD tem 64 pistas PCIe 4.0, cada uma das quais é capaz de

transferência de dados de 16 Gbit/s em ambas as direções. A memória é conectada diretamente à CPU com uma largura de banda total de até 100 GB/s.

Quando executamos o código em um computador, precisamos embaralhar os dados para os processadores (CPU ou GPU), realizarem cálculos e, em seguida, mover os resultados do processador de volta para a RAM e armazenamento durável. Portanto, para obter um bom desempenho, precisamos nos certificar de que isso funcione perfeitamente, sem que nenhum dos sistemas se torne um grande gargalo. Por exemplo, se não conseguirmos carregar as imagens com rapidez suficiente, o processador não terá nenhum trabalho a fazer. Da mesma forma, se não conseguirmos mover as matrizes com rapidez suficiente para a CPU (ou GPU), seus elementos de processamento morrerão de fome. Finalmente, se quisermos sincronizar vários computadores na rede, o último não deve retardar a computação. Uma opção é intercalar comunicação e computação. Vamos dar uma olhada nos vários componentes com mais detalhes.

12.4.2 Memória

Em sua forma mais básica, a memória é usada para armazenar dados que precisam estar prontamente acessíveis. Atualmente, a CPU RAM é tipicamente da variedade DDR4[133], oferecendo largura de banda de 20-25GB/s por módulo. Cada módulo possui um barramento de 64 bits. Normalmente, pares de módulos de memória são usados para permitir vários canais. As CPUs têm entre 2 e 4 canais de memória, ou seja, têm entre 40 GB/s e 100 GB/s de largura de banda de memória de pico. Freqüentemente, existem dois bancos por canal. Por exemplo, o Zen 3 *Threadripper* da AMD tem 8 slots.

Embora esses números sejam impressionantes, na verdade, eles contam apenas parte da história. Quando queremos ler uma parte da memória, primeiro precisamos dizer ao módulo de memória onde a informação pode ser encontrada. Ou seja, primeiro precisamos enviar o *endereço* para a RAM. Feito isso, podemos escolher ler apenas um único registro de 64 bits ou uma longa sequência de registros. Este último é chamado de *leitura intermitente*. Resumindo, enviar um endereço para a memória e configurar a transferência leva aproximadamente 100 ns (os detalhes dependem dos coeficientes de tempo específicos dos chips de memória usados), cada transferência subsequente leva apenas 0,2 ns. Resumindo, a primeira leitura é 500 vezes mais cara que as subsequentes! Poderíamos realizar até $10.000.000$ leituras aleatórias por segundo. Isso sugere que evitamos o acesso aleatório à memória o máximo possível e, em vez disso, usamos leituras (e gravações) intermitentes.

As coisas ficam um pouco mais complexas quando levamos em consideração que temos vários bancos. Cada banco pode ler a memória independentemente. Isso significa duas coisas: o número efetivo de leituras aleatórias é até 4x maior, desde que sejam distribuídas uniformemente pela memória. Isso também significa que ainda é uma má ideia realizar leituras aleatórias, uma vez que as leituras intermitentes também são 4x mais rápidas. Em segundo lugar, devido ao alinhamento da memória aos limites de 64 bits, é uma boa ideia alinhar quaisquer estruturas de dados com os mesmos limites. Compiladores fazem isso automaticamente[134] quando os sinalizadores apropriados são definidos. Os leitores curiosos são incentivados a revisar uma palestra sobre DRAMs, como a de Zeshan Chishti[135].

A memória da GPU está sujeita a requisitos de largura de banda ainda maiores, uma vez que têm muito mais elementos de processamento do que CPUs. Em geral, existem duas opções para

[133] https://en.wikipedia.org/wiki/DDR4_SDRAM
[134] https://en.wikipedia.org/wiki/Data_structure_alignment
[135] http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf

resolvê-los. Uma é tornar o barramento de memória significativamente mais amplo. Por exemplo, o RTX 2080 Ti da NVIDIA tem um barramento de 352 bits. Isso permite que muito mais informações sejam transferidas ao mesmo tempo. Em segundo lugar, as GPUs usam memória específica de alto desempenho. Dispositivos de consumo, como as séries RTX e Titan da NVIDIA, geralmente usam chips GDDR6[136] com largura de banda agregada superior a 500 GB / s. Uma alternativa é usar módulos HBM (memória de alta largura de banda). Eles usam uma interface muito diferente e se conectam diretamente com GPUs em um *wafer* de silício dedicado. Isso os torna muito caros e seu uso é normalmente limitado a chips de servidor de ponta, como a série de aceleradores NVIDIA Volta V100. Sem surpresa, a memória da GPU é *muito* menor do que a memória da CPU devido ao seu custo mais alto. Para nossos propósitos, em geral, suas características de desempenho são semelhantes, apenas muito mais rápidas. Podemos ignorar com segurança os detalhes para o propósito deste livro. Eles só importam ao ajustar os kernels da GPU para alto rendimento.

12.4.3 Armazenamento

Vimos que algumas das principais características da RAM eram *largura de banda* e *latência*. O mesmo é verdade para dispositivos de armazenamento, só que as diferenças podem ser ainda mais extremas.

Discos rígidos são usados há mais de meio século. Em poucas palavras, eles contêm uma série de pratos giratórios com cabeças que podem ser posicionadas para ler / escrever em qualquer faixa. Discos de última geração suportam até 16 TB em 9 pratos. Um dos principais benefícios dos HDDs é que eles são relativamente baratos. Uma de suas muitas desvantagens são seus modos de falha tipicamente catastróficos e sua latência de leitura relativamente alta.

Para entender o último, considere o fato de que os HDDs giram em torno de 7.200 RPM. Se fossem muito mais rápidos, estilhaçariam devido à força centrífuga exercida nas travessas. Isso tem uma grande desvantagem quando se trata de acessar um setor específico no disco: precisamos esperar até que o prato tenha girado na posição (podemos mover as cabeças, mas não acelerar os discos reais). Portanto, pode demorar mais de 8 ms até que os dados solicitados estejam disponíveis. Uma maneira comum de expressar isso é dizer que os HDDs podem operar a aproximadamente 100 IOPs. Este número permaneceu essencialmente inalterado nas últimas duas décadas. Pior ainda, é igualmente difícil aumentar a largura de banda (é da ordem de 100-200 MB/s). Afinal, cada cabeçote lê uma trilha de bits, portanto, a taxa de bits é dimensionada apenas com a raiz quadrada da densidade da informação. Como resultado, os HDDs estão rapidamente se tornando relegados ao armazenamento de arquivos e armazenamento de baixo nível para conjuntos de dados muito grandes.

Unidades de estado sólido usam memória Flash para armazenar informações de forma persistente. Isso permite um acesso *muito mais rápido* aos registros armazenados. Os SSDs modernos podem operar de 100.000 a 500.000 IOPs, ou seja, até 3 ordens de magnitude mais rápido do que os HDDs. Além disso, sua largura de banda pode chegar a 1-3 GB/s, ou seja, uma ordem de magnitude mais rápida do que os HDDs. Essas melhorias parecem boas demais para ser verdade. Na verdade, eles vêm com uma série de ressalvas, devido à forma como os SSDs são projetados.

- SSDs armazenam informações em blocos (256 KB ou maiores). Eles só podem ser escritos como um todo, o que leva um tempo significativo. Consequentemente, as gravações aleatórias bit a bit no SSD têm um desempenho muito ruim. Da mesma forma, a gravação de dados em geral leva um tempo significativo, pois o bloco deve ser lido, apagado e então reescrito com novas informações. Até agora, os controladores SSD e *firmware* desenvolveram

[136] https://en.wikipedia.org/wiki/GDDR6_SDRAM

algoritmos para atenuar isso. No entanto, as gravações podem ser muito mais lentas, em particular para SSDs QLC (célula de nível quádruplo). A chave para um melhor desempenho é manter uma *fila* de operações, preferir leituras e escrever em grandes blocos, se possível.

- As células de memória em SSDs se desgastam com relativa rapidez (geralmente após alguns milhares de gravações). Algoritmos de proteção de nível de desgaste são capazes de espalhar a degradação por muitas células. Dito isso, não é recomendado usar SSDs para arquivos de troca ou para grandes agregações de arquivos de log.
- Por último, o grande aumento na largura de banda forçou os designers de computador a conectar SSDs diretamente ao barramento PCIe. As unidades capazes de lidar com isso, chamadas de NVMe (Non Volatile Memory Enhanced), podem usar até 4 pistas PCIe. Isso equivale a até 8 GB / s no PCIe 4.0.

Cloud Storage oferece uma gama configurável de desempenho. Ou seja, a atribuição de armazenamento às máquinas virtuais é dinâmica, tanto em termos de quantidade quanto em termos de velocidade, escolhidos pelo usuário. Recomendamos que o usuário aumente o número provisionado de IOPs sempre que a latência for muito alta, por exemplo, durante o treinamento com muitos registros pequenos.

12.4.4 CPUs

As Unidades de Processamento Central (CPUs) são a peça central de qualquer computador (como antes, damos uma descrição de alto nível focando principalmente no que importa para modelos de aprendizado profundo eficientes). Eles consistem em uma série de componentes principais: núcleos de processador que são capazes de executar código de máquina, um barramento conectando-os (a topologia específica difere significativamente entre modelos de processador, gerações e fornecedores) e caches para permitir maior largura de banda e menor latência de acesso à memória do que o que é possível por leituras da memória principal. Por fim, quase todas as CPUs modernas contêm unidades de processamento vetorial para auxiliar na álgebra linear e convoluções de alto desempenho, pois são comuns no processamento de mídia e no aprendizado de máquina.

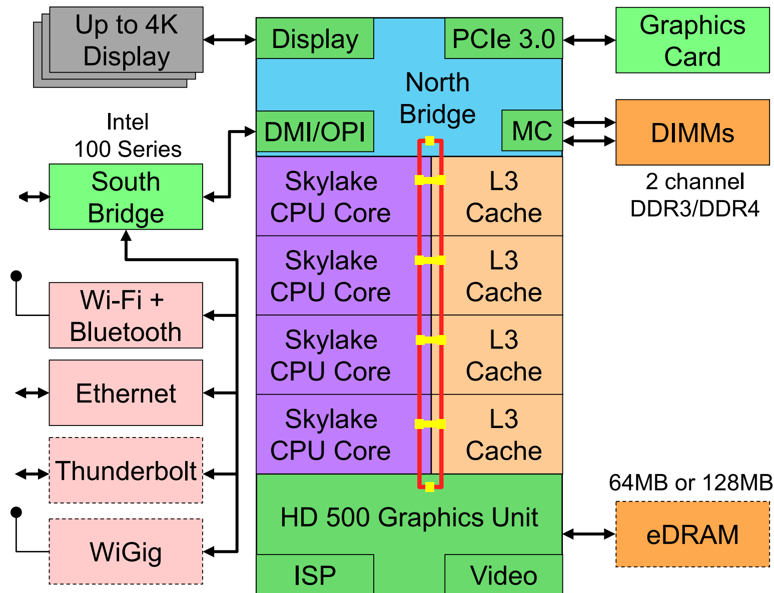


Fig. 12.4.3: CPU Intel Skylake para consumidor quad-core

Fig. 12.4.3 representa uma CPU quad-core Intel Skylake para o consumidor. Ele tem uma GPU integrada, caches e um ringbus conectando os quatro núcleos. Os periféricos (Ethernet, WiFi, Bluetooth, controlador SSD, USB, etc.) fazem parte do chipset ou são conectados diretamente (PCIe) à CPU.

Microarquitetura

Cada um dos núcleos do processador consiste em um conjunto bastante sofisticado de componentes. Embora os detalhes sejam diferentes entre gerações e fornecedores, a funcionalidade básica é praticamente padrão. O *front-end* carrega instruções e tenta prever qual caminho será seguido (por exemplo, para o fluxo de controle). As instruções são então decodificadas do código de montagem para as microinstruções. O código assembly geralmente não é o código de nível mais baixo que um processador executa. Em vez disso, instruções complexas podem ser decodificadas em um conjunto de operações de nível mais baixo. Em seguida, eles são processados pelo núcleo de execução real. Freqüentemente, o último é capaz de realizar várias operações simultaneamente. Por exemplo, o núcleo ARM Cortex A77 de Fig. 12.4.4 é capaz de realizar até 8 operações simultaneamente.

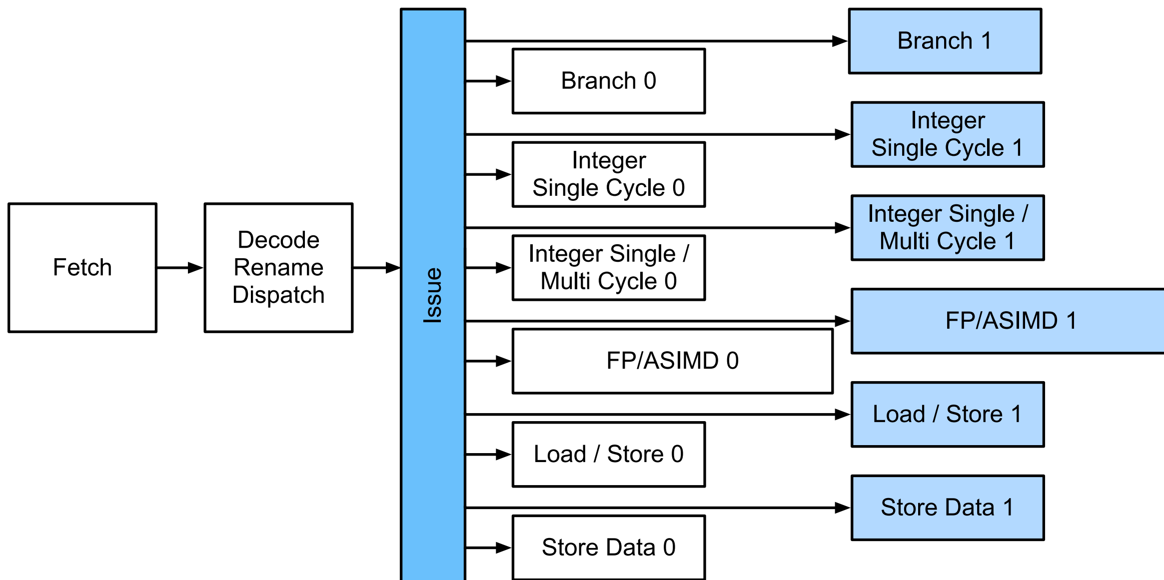


Fig. 12.4.4: Visão geral da microarquitetura ARM Cortex A77

Isso significa que programas eficientes podem ser capazes de executar mais de uma instrução por ciclo de clock, *desde que* possam ser executados independentemente. Nem todas as unidades são criadas iguais. Alguns se especializam em instruções de inteiros, enquanto outros são otimizados para desempenho de ponto flutuante. Para aumentar a taxa de transferência, o processador também pode seguir vários *codepaths* simultaneamente em uma instrução de ramificação e, em seguida, descartar os resultados das ramificações não realizadas. É por isso que as unidades de previsão de ramificação são importantes (no *front-end*) de forma que apenas os caminhos mais promissores são seguidos.

Vetorização

O aprendizado profundo exige muita computação. Portanto, para tornar as CPUs adequadas para aprendizado de máquina, é necessário realizar muitas operações em um ciclo de clock. Isso é obtido por meio de unidades vetoriais. Eles têm nomes diferentes: no ARM são chamados de NEON, no x86 a última geração é conhecida como unidades AVX2[137]. Um aspecto comum é que eles são capazes de realizar operações SIMD (instrução única, dados múltiplos). Fig. 12.4.5 mostra como 8 inteiros curtos podem ser adicionados em um ciclo de clock no ARM.

[137] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

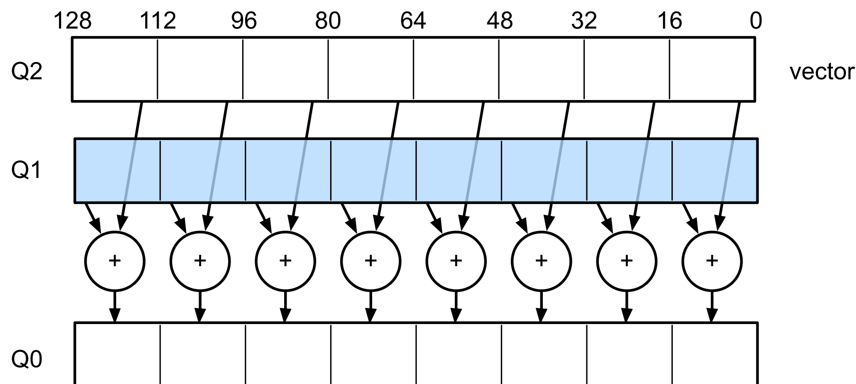


Fig. 12.4.5: Vetorização NEON de 128 bits

Dependendo das opções de arquitetura, esses registros têm até 512 bits de comprimento, permitindo a combinação de até 64 pares de números. Por exemplo, podemos multiplicar dois números e adicioná-los a um terceiro, também conhecido como multiplicação-adição fundida. O OpenVino[138] da Intel usa isso para atingir um rendimento respeitável para aprendizado profundo em CPUs de nível de servidor. Observe, porém, que esse número é totalmente diminuto em relação ao que as GPUs são capazes de alcançar. Por exemplo, o RTX 2080 Ti da NVIDIA tem 4.352 núcleos CUDA, cada um dos quais é capaz de processar tal operação a qualquer momento.

Cache

Considere a seguinte situação: temos um modesto núcleo de CPU com 4 núcleos, conforme ilustrado em Fig. 12.4.3 acima, rodando na frequência de 2 GHz. Além disso, vamos supor que temos uma contagem IPC (instruções por *clock*) de 1 e que as unidades têm AVX2 com largura de 256 bits habilitada. Além disso, vamos supor que pelo menos um dos registradores usados para as operações AVX2 precisa ser recuperado da memória. Isso significa que a CPU consome 4x256bit = 1kbit de dados por ciclo de clock. A menos que possamos transferir $2 \cdot 10^9 \cdot 128 = 256 \cdot 10^9$ bytes para o processador por segundo, os elementos de processamento morrerão de fome. Infelizmente, a interface de memória de um tal chip suporta apenas transferência de dados de 20-40 GB/s, ou seja, uma ordem de magnitude a menos. A correção é evitar o carregamento de *novos* dados da memória o máximo possível e, em vez disso, armazená-los em cache localmente na CPU. É aqui que os caches são úteis (consulte este artigo da Wikipedia[139] para uma introdução). Normalmente, os seguintes nomes / conceitos são usados:

- **Registradores**, estritamente falando, não fazem parte do cache. Eles ajudam a preparar as instruções. Dito isso, os registradores da CPU são locais de memória que uma CPU pode acessar na velocidade do *clock* sem nenhuma penalidade de atraso. CPUs têm dezenas de registradores. Cabe ao compilador (ou programador) usar os registradores de maneira eficiente. Por exemplo, a linguagem de programação C tem uma palavra-chave `register`.
- Os caches **L1** são a primeira linha de defesa contra os altos requisitos de largura de banda da memória. Os caches L1 são minúsculos (os tamanhos típicos podem ser de 32-64kB) e geralmente se dividem em caches de dados e instruções. Quando os dados são encontrados no cache L1, o acesso é muito rápido. Se não puder ser encontrado lá, a pesquisa progride para baixo na hierarquia do cache.

[138] https://01.org/openvinotoolkit
[139] https://en.wikipedia.org/wiki/Cache_hierarchy

- **Caches L2** são a próxima parada. Dependendo do design da arquitetura e do tamanho do processador, eles podem ser exclusivos. Eles podem ser acessíveis apenas por um determinado núcleo ou compartilhados entre vários núcleos. Os caches L2 são maiores (normalmente 256-512kB por núcleo) e mais lentos do que L1. Além disso, para acessar algo em L2, primeiro precisamos verificar para perceber que os dados não estão em L1, o que adiciona uma pequena quantidade de latência extra.
- **Caches L3** são compartilhados entre vários núcleos e podem ser muito grandes. As CPUs do servidor Epyc 3 da AMD têm 256 MB de cache espalhados por vários chips. Os números mais comuns estão na faixa de 4 a 8 MB.

Prever quais elementos de memória serão necessários em seguida é um dos principais parâmetros de otimização no projeto do chip. Por exemplo, é aconselhável percorrer a memória em uma direção *para frente*, pois a maioria dos algoritmos de cache tentará *ler para a frente* em vez de para trás. Da mesma forma, manter os padrões de acesso à memória locais é uma boa maneira de melhorar o desempenho. Adicionar caches é uma faca de dois gumes. Por um lado, eles garantem que os núcleos do processador não morram de fome de dados. Ao mesmo tempo, eles aumentam o tamanho do chip, consumindo área que, de outra forma, poderia ser gasta no aumento do poder de processamento. Além disso, *falhas de cache* podem ser caras. Considere o pior cenário, descrito em: Fig. 12.4.6. Um local de memória é armazenado em cache no processador 0 quando uma *thread* no processador 1 solicita os dados. Para obtê-lo, o processador 0 precisa parar o que está fazendo, gravar as informações de volta na memória principal e deixar que o processador 1 as leia da memória. Durante esta operação, os dois processadores aguardam. Muito potencialmente, esse código é executado *mais lentamente* em vários processadores quando comparado a uma implementação eficiente de processador único. Esse é mais um motivo pelo qual há um limite prático para o tamanho da cache (além do tamanho físico).

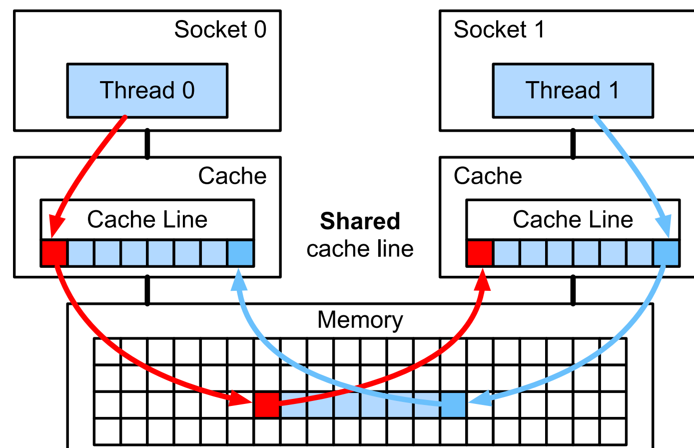


Fig. 12.4.6: Compartilhamento falso (imagem cortesia da Intel)

12.4.5 GPUs e outros Aceleradores

Não é exagero afirmar que o *deep learning* não teria tido sucesso sem as GPUs. Da mesma forma, é bastante razoável argumentar que as fortunas dos fabricantes de GPUs aumentaram significativamente devido ao *deep learning*. Essa coevolução de *hardware* e algoritmos levou a uma situação em que, para melhor ou pior, o *deep learning* é o paradigma de modelagem estatística preferível. Portanto, vale a pena entender os benefícios específicos que as GPUs e aceleradores relacionados, como a TPU (Jouppi et al., 2017) oferecem.

É digno de nota uma distinção que é frequentemente feita na prática: os aceleradores são otimizados para treinamento ou inferência. Para o último, só precisamos calcular a propagação direta em uma rede. Nenhum armazenamento de dados intermediários é necessário para retropropagação. Além disso, podemos não precisar de cálculos muito precisos (FP16 ou INT8 normalmente são suficientes). Por outro lado, durante o treinamento, todos os resultados intermediários precisam ser armazenados para calcular os gradientes. Além disso, o acúmulo de gradientes requer maior precisão para evitar *underflow* numérico (ou estouro). Isso significa que FP16 (ou precisão mista com FP32) é o mínimo necessário. Tudo isso requer memória maior e mais rápida (HBM2 vs. GDDR6) e mais poder de processamento. Por exemplo, as GPUs T4 da NVIDIA Turing[140] são otimizadas para inferência, enquanto as GPUs V100 são preferíveis para treinamento.

Lembre-se de Fig. 12.4.5. Adicionar unidades de vetor a um núcleo de processador nos permitiu aumentar o rendimento significativamente (no exemplo da figura, fomos capazes de realizar 16 operações simultaneamente). E se adicionássemos operações que otimizassem não apenas as operações entre vetores, mas também entre matrizes? Essa estratégia levou ao Tensor Cores (mais sobre isso em breve). Em segundo lugar, e se adicionarmos muitos mais núcleos? Em suma, essas duas estratégias resumem as decisões de design em GPUs. Fig. 12.4.7 dá uma visão geral sobre um bloco de processamento básico. Ele contém 16 unidades inteiras e 16 unidades de ponto flutuante. Além disso, dois núcleos do Tensor aceleram um subconjunto estreito de operações adicionais relevantes para o aprendizado profundo. Cada Streaming Multiprocessor (SM) consiste em quatro desses blocos.

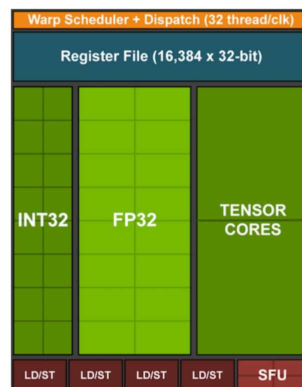


Fig. 12.4.7: Bloco de Processamento NVIDIA Turing (imagem cortesia da NVIDIA)

12 multiprocessadores de streaming são então agrupados em clusters de processamento gráfico que compõem os processadores TU102 de última geração. Canais de memória amplos e um cache L2 complementam a configuração. Fig. 12.4.8 tem os detalhes relevantes. Uma das razões para projetar tal dispositivo é que blocos individuais podem ser adicionados ou removidos conforme necessário para permitir chips mais compactos e lidar com problemas de rendimento (módulos

[140] https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/

defeituosos podem não ser ativados). Felizmente, a programação de tais dispositivos está bem escondida do pesquisador casual de aprendizado profundo sob camadas de CUDA e código de estrutura. Em particular, mais de um dos programas pode muito bem ser executado simultaneamente na GPU, desde que haja recursos disponíveis. No entanto, vale a pena estar ciente das limitações dos dispositivos para evitar escolher modelos que não cabem na memória do dispositivo.

Fig. 12.4.8: Arquitetura NVIDIA Turing (imagem cortesia da NVIDIA)

Um último aspecto que vale a pena mencionar com mais detalhes são os TensorCores. Eles são um exemplo de uma tendência recente de adicionar circuitos mais otimizados que são especificamente eficazes para o aprendizado profundo. Por exemplo, o TPU adicionou uma matriz sistólica (Kung, 1988)para multiplicação rápida da matriz. Lá, o projeto deveria suportar um número muito pequeno (um para a primeira geração de TPUs) de grandes operações. TensorCores estão na outra extremidade. Eles são otimizados para pequenas operações envolvendo matrizes entre 4x4 e 16x16, dependendo de sua precisão numérica. Fig. 12.4.9 dá uma visão geral das otimizações.

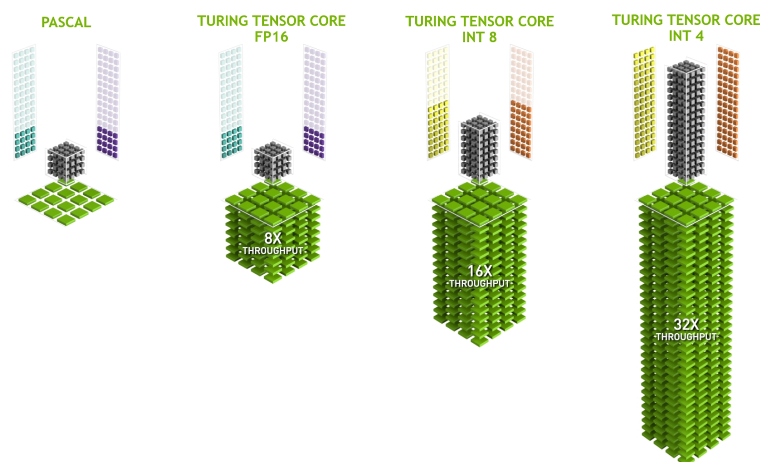


Fig. 12.4.9: NVIDIA TensorCores de Turing (imagem cortesia da NVIDIA)

Obviamente, ao otimizar para computação, acabamos fazendo certos compromissos. Um deles

é que as GPUs não são muito boas em lidar com interrupções e dados esparsos. Embora haja exceções notáveis, como Gunrock[141] (Wang et al., 2016), o padrão de acesso de matrizes e vetores esparsos não vai bem com as operações de leitura intermitente de alta largura de banda em que as GPUs se destacam. Combinar os dois objetivos é uma área de pesquisa ativa. Veja, por exemplo, DGL[142], uma biblioteca ajustada para *deep learning* em gráficos.

12.4.6 Redes e Barramentos

Sempre que um único dispositivo é insuficiente para a otimização, precisamos transferir dados de e para ele para sincronizar o processamento. É aqui que as redes e os barramentos são úteis. Temos vários parâmetros de design: largura de banda, custo, distância e flexibilidade. Por um lado, temos WiFi que tem um alcance muito bom, é muito fácil de usar (sem fios, afinal), barato, mas oferece largura de banda e latência comparativamente medíocres. Nenhum pesquisador de aprendizado de máquina em sã consciência o usaria para construir um cluster de servidores. A seguir, enfocamos as interconexões adequadas para o *deep learning*.

- **PCIe** é um barramento dedicado para conexões ponto a ponto de largura de banda muito alta (até 16 Gbs em PCIe 4.0) por *lane*. A latência é da ordem de microssegundos de um dígito (5 μs). Os links PCIe são preciosos. Os processadores têm apenas um número limitado deles: o EPYC 3 da AMD tem 128 *lanes*, o Xeon da Intel tem até 48 *lanes* por chip; em CPUs para desktop os números são 20 (Ryzen 9) e 16 (Core i9), respectivamente. Como as GPUs têm normalmente 16 *lanes*, isso limita o número de GPUs que podem se conectar à CPU em largura de banda total. Afinal, elas precisam compartilhar os links com outros periféricos de alta largura de banda, como armazenamento e Ethernet. Assim como com o acesso à RAM, grandes transferências em massa são preferíveis devido à sobrecarga de pacotes reduzida.
- **Ethernet** é a forma mais comumente usada de conectar computadores. Embora seja significativamente mais lento que o PCIe, é muito barato e resistente para instalar e cobre distâncias muito maiores. A largura de banda típica para servidores de baixo nível é de 1 GBit/s. Dispositivos de ponta (por exemplo, instâncias C5[143] na nuvem) oferecem entre 10 e 100 GBit/s de largura de banda. Como em todos os casos anteriores, a transmissão de dados tem sobrecargas significativas. Observe que quase nunca usamos Ethernet bruta diretamente, mas sim um protocolo executado na parte superior da interconexão física (como UDP ou TCP/IP). Isso adiciona mais sobrecarga. Como PCIe, Ethernet é projetada para conectar dois dispositivos, por exemplo, um computador e um switch.
- **Switches** nos permitem conectar vários dispositivos de maneira que qualquer par deles possa realizar uma conexão ponto a ponto (normalmente largura de banda total) simultaneamente. Por exemplo, comutadores Ethernet podem conectar 40 servidores em alta largura de banda transversal. Observe que os *switches* não são exclusivos das redes de computadores tradicionais. Mesmo as pistas PCIe podem ser trocadas[144]. Isso ocorre, por exemplo, para conectar um grande número de GPUs a um processador host, como é o caso das instâncias P2[145].
- **NVLink** é uma alternativa ao PCIe quando se trata de interconexões de largura de banda muito alta. Ele oferece uma taxa de transferência de dados de até 300 Gbit/s por link. As GPUs de servidor (Volta V100) têm 6 links, enquanto as GPUs de consumidor (RTX 2080 Ti) têm

[141] https://github.com/gunrock/gunrock
[142] http://dgl.ai
[143] https://aws.amazon.com/ec2/instance-types/c5/
[144] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches
[145] https://aws.amazon.com/ec2/instance-types/p2/

apenas um link, operando a uma taxa reduzida de 100 Gbit/s. Recomendamos usar NCCL[146] para obter alta transferência de dados entre GPUs.

12.4.7 Resumo

- Dispositivos têm sobrecargas para operações. Portanto, é importante ter como objetivo um pequeno número de transferências grandes, em vez de muitas transferências pequenas. Isso se aplica a RAM, SSDs, redes e GPUs.
- A vetorização é a chave para o desempenho. Certifique-se de estar ciente das habilidades específicas de seu acelerador. Por exemplo, algumas CPUs Intel Xeon são particularmente boas para operações INT8, GPUs NVIDIA Volta se destacam em operações matriz-matriz FP16 e NVIDIA Turing brilha em operações FP16, INT8 e INT4.
- O estouro numérico devido a pequenos tipos de dados pode ser um problema durante o treinamento (e em menor extensão durante a inferência).
- O *aliasing* pode degradar significativamente o desempenho. Por exemplo, o alinhamento da memória em CPUs de 64 bits deve ser feito em relação aos limites de 64 bits. Em GPUs, é uma boa ideia manter os tamanhos de convolução alinhados, por exemplo, para TensorCores.
- Combine seus algoritmos com o *hardware* (pegada de memória, largura de banda, etc.). Grande aceleração (ordens de magnitude) podem ser alcançadas ao ajustar os parâmetros em caches.
- Recomendamos que você esboce o desempenho de um novo algoritmo no papel antes de verificar os resultados experimentais. Discrepâncias de ordem de magnitude ou mais são motivos de preocupação.
- Use *profilers* para depurar gargalos de desempenho.
- O *hardware* de treinamento e inferência tem diferentes pontos ideais em termos de preço / desempenho.

12.4.8 Mais Números de Latência

O resumo em Table 12.4.1 e Table 12.4.2 são devidos a Eliot Eshelman[147] que mantém uma versão atualizada dos números como um GitHub Gist[148].

Table 12.4.1: Números de latência comuns.

Ação	T e m p o	Notas
Referência / acerto do cache L1	5 n s	4 ciclos
Adicionar ponto flutuante / mult / FMA	5 n s	4 ciclos
Referência / acerto do cache L2	5 n s	12 ~ 17 ciclos
Erro de previsão de branch	6 n s	15 ~ 20 ciclos
Acerto de cache L3 (cache não compartilhado)	1 6 n s	42 ciclos
Acerto de cache L3 (compartilhado em outro núcleo)	2 5 n s	65 ciclos
Bloqueio / desbloqueio Mutex	2 5 n s	

continues on next page

[146] https://github.com/NVIDIA/nccl
[147] https://gist.github.com/eshelman
[148] https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646

Table 12.4.1 – continued from previous page

Ação	Tempo	Notas
Acerto de cache L3 (modificado em outro núcleo)	29 ns	75 ciclos
Acerto de cache L3 (em um soquete de CPU remoto)	40 ns	100 ~ 300 ciclos (40 ~ 116 ns)
Salto QPI para outra CPU (por salto)	40 ns	
Ref. de memória 64 MB (CPU local)	46 ns	TinyMemBench na Broadwell E5-2690v4
Ref. de memória 64 MB (CPU remota)	70 ns	TinyMemBench na Broadwell E5-2690v4
Ref. de memória 256 MB (CPU local)	75 ns	TinyMemBench na Broadwell E5-2690v4
Gravação aleatória Intel Optane	94 ns	UCSD Systems Lab Não-Volátil
Ref. de memória 256 MB (CPU remota)	120 ns	TinyMemBench na Broadwell E5-2690v4
Leitura aleatória Intel Optane	305 ns	UCSD Systems Lab Não-Volátil
Enviar 4KB em HPC fabric com mais de 100 Gbps	1 μs	MVAPICH2 sobre Intel Omni-Path
Comprimir 1 KB com Google Snappy	3 μs	
Enviar 4 KB por Ethernet de 10 Gbps	10 μs	
Gravar 4KB aleatoriamente no NVMe SSD	30 μs	DC P3608 NVMe SSD (QOS 99% é 500μs)
Transferir 1 MB de/para GPU NVLink	30 μs	~33GB/s na NVIDIA 40GB NVLink
Transferir 1 MB de/para GPU PCI-E	80 μs	~12GB/s na PCIe 3.0 x16 link
Ler 4 KB aleatoriamente do NVMe SSD	120 μs	DC P3608 NVMe SSD (QOS 99%)
Ler 1 MB sequencialmente do NVMe SSD	208 μs	~4.8GB/s DC P3608 NVMe SSD
Gravar 4 KB aleatoriamente em SATA SSD	500 μs	DC S3510 SATA SSD (QOS 99.9%)
Ler 4 KB aleatoriamente de SATA SSD	500 μs	DC S3510 SATA SSD (QOS 99.9%)
Ida e volta no mesmo datacenter	500 μs	Ping *one-way* é ~250μs
Ler 1 MB sequencialmente de SATA SSD	2 ms	~550MB/s DC S3510 SATA SSD
Ler 1 MB sequencialmente do disco	5 ms	~200MB/s servidor HDD
Acesso aleatório ao disco (busca + rotação)	10 ms	
Enviar pacote Califórnia-> Holanda-> Califórnia	150 ms	

Table 12.4.2: Números de latência para GPU NVIDIA Tesla.

Ação	Tempo	Notas
Acesso à memória compartilhada da GPU	30 ns	30~90 cycles (conflitos de banco adicionam latência)
Acesso à memória global da GPU	200 ns	200~800 ciclos
Iniciar o kernel CUDA na GPU	10 μs	A CPU do host instrui a GPU a iniciar o kernel
Transferir 1 MB de / para GPU NVLink	30 μs	~33GB/s na NVIDIA 40GB NVLink
Transferir 1 MB de / para GPU PCI-E	80 μs	~12GB/s na PCI-Express x16 link

12.4.9 Exercícios

1. Escreva o código C para testar se há alguma diferença na velocidade entre acessar a memória alinhada ou desalinhada em relação à interface de memória externa. Dica: tome cuidado com os efeitos do cache.
2. Teste a diferença de velocidade entre acessar a memória em sequência ou com um determinado passo.
3. Como você pode medir o tamanho da cache em uma CPU?

4. Como você distribuiria os dados em vários canais de memória para obter largura de banda máxima? Como você o colocaria se tivesse muitos fios pequenos?
5. Um HDD de classe empresarial está girando a 10.000 rpm. Qual é o tempo absolutamente mínimo que um HDD precisa gastar no pior caso antes de poder ler os dados (você pode supor que os cabeçotes se movem quase instantaneamente)? Por que os HDDs de 2,5 "estão se tornando populares para servidores comerciais (em relação às unidades de 3,5" e 5,25 ")?
6. Suponha que um fabricante de HDD aumente a densidade de armazenamento de 1 Tbit por polegada quadrada para 5 Tbit por polegada quadrada. Quanta informação você pode armazenar em um anel em um HDD de 2,5 "? Há uma diferença entre as trilhas interna e externa?
7. As instâncias AWS P2 têm 16 GPUs K80 Kepler. Use `lspci` em uma instância p2.16xlarge e p2.8xlarge para entender como as GPUs estão conectadas às CPUs. Dica: fique de olho nas pontes PCI PLX.
8. Ir de tipos de dados de 8 para 16 bits aumenta a quantidade de silício em aproximadamente 4x. Por quê? Por que a NVIDIA pode ter adicionado operações INT4 a suas GPUs Turing?
9. Dados 6 links de alta velocidade entre GPUs (como para as GPUs Volta V100), como você conectaria 8 deles? Procure a conectividade usada nos servidores P3.16xlarge.
10. Quão mais rápido é ler para a frente na memória do que ler para trás? Este número difere entre diferentes computadores e fornecedores de CPU? Por quê? Escreva o código C e experimente-o.
11. Você pode medir o tamanho do cache do seu disco? O que é um HDD típico? Os SSDs precisam de um cache?
12. Meça a sobrecarga do pacote ao enviar mensagens pela Ethernet. Procure a diferença entre as conexões UDP e TCP / IP.
13. O acesso direto à memória permite que outros dispositivos além da CPU gravem (e leiam) diretamente na (da) memória. Por que isso é uma boa ideia?
14. Observe os números de desempenho da GPU Turing T4. Por que o desempenho "apenas" dobra quando você passa de FP16 para INT8 e INT4?
15. Qual é o menor tempo necessário para um pacote em uma viagem de ida e volta entre São Francisco e Amsterdã? Dica: você pode assumir que a distância é de 10.000 km.

Discussions[149]

12.5 Treinamento em Várias GPUs

Até agora, discutimos como treinar modelos de forma eficiente em CPUs e GPUs. Nós até mostramos como frameworks de aprendizado profundo como MXNet (e TensorFlow) permitem paralelizar computação e comunicação automaticamente entre elas em Section 12.3. Por último, mostramos em Section 5.5 como listar todas as GPUs disponíveis em um computador usando `nvidia-smi`. O que *não* discutimos é como realmente paralelizar o treinamento de aprendizado profundo (omitimos qualquer discussão de *inferência* em várias GPUs aqui, pois é um tópico raramente usado e avançado que vai além do escopo deste livro). Em vez disso, sugerimos que, de

[149] https://discuss.d2l.ai/t/363

alguma forma, seria possível dividir os dados em vários dispositivos e fazê-los funcionar. A presente seção preenche os detalhes e mostra como treinar uma rede em paralelo ao começar do zero. Detalhes sobre como tirar proveito da funcionalidade do Gluon são relegados a Section 12.6. Assumimos que o leitor está familiarizado com algoritmos SGD de minibatch, como os descritos em Section 11.5.

12.5.1 Dividindo o Problema

Vamos começar com um problema de visão computacional simples e uma rede ligeiramente arcaica, por exemplo, com várias camadas de convoluções, agrupamento e, possivelmente, algumas camadas densas no final. Ou seja, vamos começar com uma rede que se parece bastante com LeNet (LeCun et al., 1998) ou AlexNet (Krizhevsky et al., 2012). Dadas várias GPUs (2 se for um servidor de desktop, 4 em um g4dn.12xlarge, 8 em um AWS p3.16xlarge, ou 16 em um p2.16xlarge), queremos particionar o treinamento de maneira a obter uma boa aceleração enquanto beneficiando simultaneamente de opções de design simples e reproduzíveis. Afinal, várias GPUs aumentam a capacidade de *memória* e *computação*. Em suma, temos várias opções, dado um minibatch de dados de treinamento que desejamos classificar.

- Podemos particionar as camadas de rede em várias GPUs. Ou seja, cada GPU recebe como entrada os dados que fluem para uma camada específica, processa os dados em várias camadas subsequentes e, em seguida, envia os dados para a próxima GPU.
  - Isso nos permite processar dados com redes maiores, em comparação com o que uma única GPU poderia suportar.
  - A pegada de memória por GPU pode ser bem controlada (é uma fração da pegada total da rede)
  - A interface entre as camadas (e, portanto, as GPUs) requer uma sincronização rígida. Isso pode ser complicado, especialmente se as cargas de trabalho computacionais não forem correspondidas adequadamente entre as camadas. O problema é agravado por um grande número de GPUs.
  - A interface entre as camadas requer grandes quantidades de transferência de dados (ativações, gradientes). Isso pode sobrecarregar a largura de banda dos barramentos da GPU.
  - Computação intensiva, mas operações sequenciais não são triviais para particionar. Veja, por exemplo, (Mirhoseini et al., 2017) para um melhor esforço a este respeito. Continua sendo um problema difícil e não está claro se é possível obter uma boa escala (linear) em problemas não triviais. Não o recomendamos, a menos que haja um excelente suporte de estrutura / sistema operacional para encadear várias GPUs.
- Podemos dividir o trabalho necessário em camadas individuais. Por exemplo, em vez de computar 64 canais em uma única GPU, poderíamos dividir o problema em 4 GPUs, cada uma gerando dados para 16 canais. Da mesma forma, para uma camada densa, poderíamos dividir o número de neurônios de saída. Fig. 12.5.1 ilustra este design. A figura foi tirada de (Krizhevsky et al., 2012) onde esta estratégia foi usada para lidar com GPUs que tinham uma pegada de memória muito pequena (2 GB na época).
  - Isso permite um bom dimensionamento em termos de computação, desde que o número de canais (ou neurônios) não seja muito pequeno.
  - Várias GPUs podem processar redes cada vez maiores, uma vez que a memória disponível é dimensionada linearmente.

- Precisamos de um número *muito grande* de operações de sincronização / barreira, pois cada camada depende dos resultados de todas as outras camadas.
- A quantidade de dados que precisa ser transferida é potencialmente ainda maior do que ao distribuir camadas entre GPUs. Não recomendamos esta abordagem devido ao seu custo de largura de banda e complexidade.

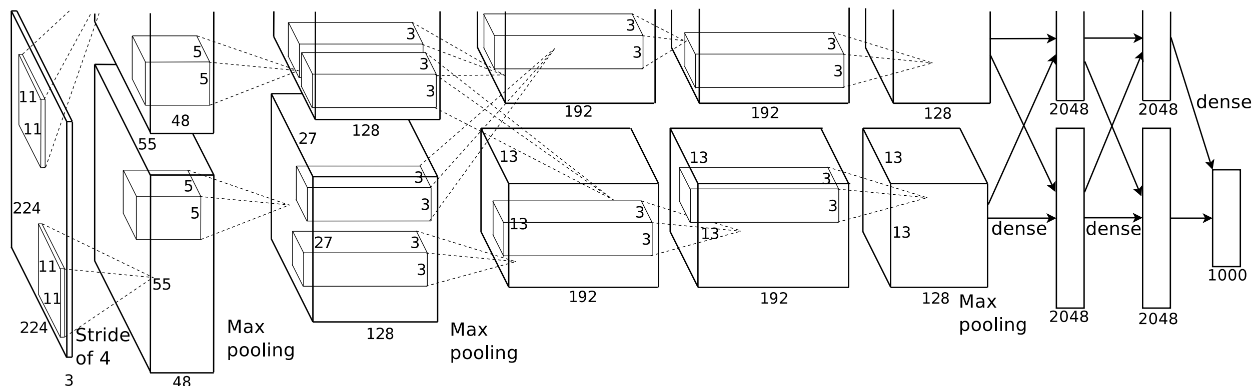


Fig. 12.5.1: Paralelismo de modelo no design AlexNet original devido à memória GPU limitada.

- Por último, podemos particionar os dados em várias GPUs. Desta forma, todas as GPUs realizam o mesmo tipo de trabalho, embora em observações diferentes. Os gradientes são agregados entre as GPUs após cada minibatch.
  - Esta é a abordagem mais simples e pode ser aplicada em qualquer situação.
  - Adicionar mais GPUs não nos permite treinar modelos maiores.
  - Só precisamos sincronizar após cada minibatch. Dito isso, é altamente desejável começar a trocar parâmetros de gradientes enquanto outros ainda estão sendo calculados.
  - Um grande número de GPUs leva a tamanhos de minibatch muito grandes, reduzindo assim a eficiência do treinamento.

Em geral, o paralelismo de dados é a maneira mais conveniente de proceder, desde que tenhamos acesso a GPUs com memória suficientemente grande. Veja também: cite: `Li.Andersen.Park.ea.2014` para uma descrição detalhada do particionamento para treinamento distribuído. A memória da GPU costumava ser um problema nos primeiros dias do *deep learning*. Até agora, esse problema foi resolvido para todos, exceto os casos mais incomuns. Nos concentramos no paralelismo de dados a seguir.

12.5.2 Paralelismo de Dados

Suponha que haja k GPUs em uma máquina. Dado o modelo a ser treinado, cada GPU manterá um conjunto completo de parâmetros do modelo de forma independente. O treinamento prossegue da seguinte maneira (consulte Fig. 12.5.2 para obter detalhes sobre o treinamento paralelo de dados em duas GPUs).

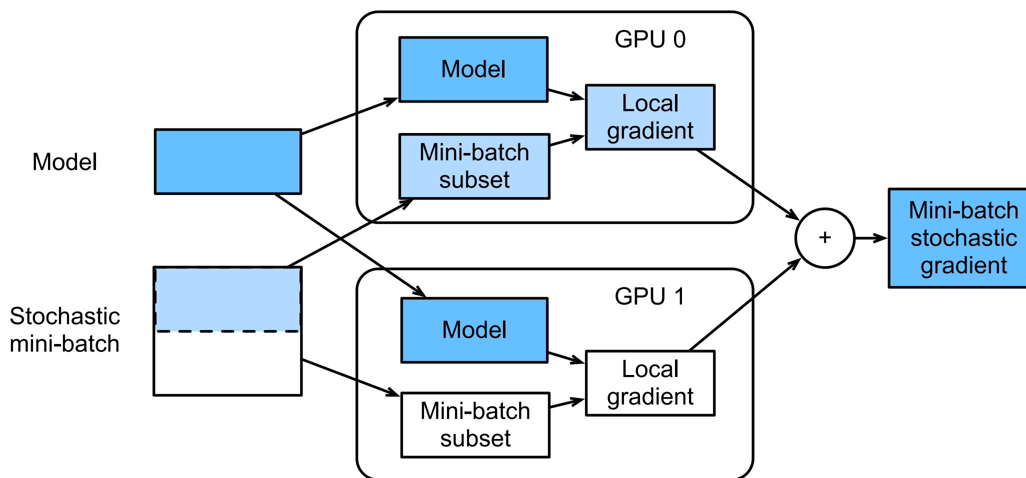


Fig. 12.5.2: Cálculo do gradiente estocástico de minibatch usando paralelismo de dados e duas GPUs.

- Em qualquer iteração de treinamento, dado um minibatch aleatório, dividimos os exemplos no lote em porções de k e os distribuímos uniformemente pelas GPUs.
- Cada GPU calcula a perda e o gradiente dos parâmetros do modelo com base no subconjunto de minibatch que foi atribuído e nos parâmetros do modelo que mantém.
- Os gradientes locais de cada uma das k GPUs são agregados para obter o gradiente estocástico do minibatch atual.
- O gradiente agregado é redistribuído para cada GPU.
- Cada GPU usa este gradiente estocástico de minibatch para atualizar o conjunto completo de parâmetros do modelo que ele mantém.

Uma comparação de diferentes formas de paralelização em várias GPUs é descrita em Fig. 12.5.3. Observe que, na prática, *aumentamos* o tamanho do minibatch k-fold ao treinar em k GPUs, de forma que cada GPU tenha a mesma quantidade de trabalho a fazer como se estivéssemos treinando em apenas uma única GPU. Em um servidor de 16 GPUs, isso pode aumentar o tamanho do minibatch consideravelmente e podemos ter que aumentar a taxa de aprendizado de acordo. Observe também que Section 7.5 precisa ser ajustado (por exemplo, mantendo um coeficiente de norma de lote separado por GPU). A seguir, usaremos Section 6.6 como a rede modelo para ilustrar o treinamento multi-GPU. Como sempre, começamos importando os pacotes e módulos relevantes.

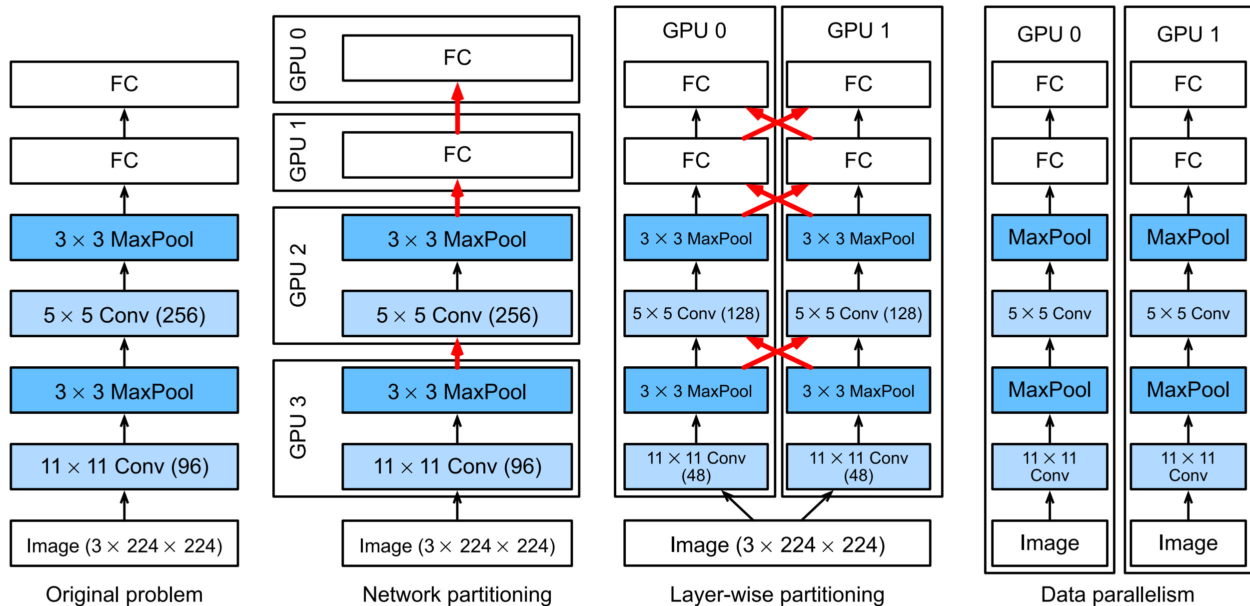


Fig. 12.5.3: Paralelização em várias GPUs. Da esquerda para a direita - problema original, particionamento de rede, particionamento de camada, paralelismo de dados.

```
%matplotlib inline
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

12.5.3 Uma Rede Exemplo

Usamos LeNet conforme apresentado em Section 6.6. Nós o definimos do zero para ilustrar a troca de parâmetros e a sincronização em detalhes.

```
# Initialize model parameters
scale = 0.01
W1 = torch.randn(size=(20, 1, 3, 3)) * scale
b1 = torch.zeros(20)
W2 = torch.randn(size=(50, 20, 5, 5)) * scale
b2 = torch.zeros(50)
W3 = torch.randn(size=(800, 128)) * scale
b3 = torch.zeros(128)
W4 = torch.randn(size=(128, 10)) * scale
b4 = torch.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
    h1_conv = F.conv2d(input=X, weight=params[0], bias=params[1])
    h1_activation = F.relu(h1_conv)
    h1 = F.avg_pool2d(input=h1_activation, kernel_size=(2, 2), stride=(2, 2))
    h2_conv = F.conv2d(input=h1, weight=params[2], bias=params[3])
```

(continues on next page)

(continued from previous page)

```
    h2_activation = F.relu(h2_conv)
    h2 = F.avg_pool2d(input=h2_activation, kernel_size=(2, 2), stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = torch.mm(h2, params[4]) + params[5]
    h3 = F.relu(h3_linear)
    y_hat = torch.mm(h3, params[6]) + params[7]
    return y_hat

# Cross-entropy loss function
loss = nn.CrossEntropyLoss(reduction='none')
```

12.5.4 Sincronização de Dados

Para um treinamento multi-GPU eficiente, precisamos de duas operações básicas: em primeiro lugar, precisamos ter a capacidade de distribuir uma lista de parâmetros para vários dispositivos e anexar gradientes (`get_params`). Sem parâmetros, é impossível avaliar a rede em uma GPU. Em segundo lugar, precisamos da capacidade de somar parâmetros em vários dispositivos, ou seja, precisamos de uma função `allreduce`.

```
def get_params(params, device):
    new_params = [p.clone().to(device) for p in params]
    for p in new_params:
        p.requires_grad_()
    return new_params
```

Vamos tentar copiar os parâmetros do modelo de lenet para gpu (0).

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 weight: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.
↪, 0.],
       device='cuda:0', requires_grad=True)
b1 grad: None
```

Como ainda não realizamos nenhum cálculo, o gradiente em relação aos pesos de polarização ainda é 0. Agora, vamos supor que temos um vetor distribuído por várias GPUs. A função `allreduce` a seguir adiciona todos os vetores e transmite o resultado de volta para todas as GPUs. Observe que para que isso funcione precisamos copiar os dados para o dispositivo acumulando os resultados.

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].to(data[0].device)
    for i in range(1, len(data)):
        data[i] = data[0].to(data[i].device)
```

Vamos testar isso criando vetores com diferentes valores em diferentes dispositivos e agregando-os.

```
data = [torch.ones((1, 2), device=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
before allreduce:
 tensor([[1., 1.]], device='cuda:0')
 tensor([[2., 2.]], device='cuda:1')
after allreduce:
 tensor([[3., 3.]], device='cuda:0')
 tensor([[3., 3.]], device='cuda:1')
```

12.5.5 Distribuindo Dados

Precisamos de uma função de utilitário simples para distribuir um minibatch uniformemente em várias GPUs. Por exemplo, em 2 GPUs, gostaríamos de ter metade dos dados a serem copiados para cada uma das GPUs. Por ser mais conveniente e conciso, usamos a função embutida de divisão e carga no Gluon (para experimentá-la em uma matriz 4×5).

```
data = torch.arange(20).reshape(4, 5)
devices = [torch.device('cuda:0'), torch.device('cuda:1')]
split = nn.parallel.scatter(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

```
input : tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]])
load into [device(type='cuda', index=0), device(type='cuda', index=1)]
output: (tensor([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]], device='cuda:0'), tensor([[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]], device='cuda:1'))
```

Para reutilização posterior, definimos uma função `split_batch` que divide os dados e rótulos.

```
#@save
def split_batch(X, y, devices):
    """Split `X` and `y` into multiple devices."""
    assert X.shape[0] == y.shape[0]
    return (nn.parallel.scatter(X, devices),
            nn.parallel.scatter(y, devices))
```

12.5.6 Treinamento

Agora podemos implementar o treinamento multi-GPU em um único minibatch. Sua implementação é baseada principalmente na abordagem de paralelismo de dados descrita nesta seção. Usaremos as funções auxiliares que acabamos de discutir, `allreduce` e`split_and_load`, para sincronizar os dados entre várias GPUs. Observe que não precisamos escrever nenhum código específico para atingir o paralelismo. Uma vez que o gráfico computacional não tem nenhuma dependência entre dispositivos dentro de um minibatch, ele é executado em paralelo *automaticamente*.

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    # Loss is calculated separately on each GPU
    losses = [loss(lenet(X_shard, device_W), y_shard).sum()
              for X_shard, y_shard, device_W in zip(
                  X_shards, y_shards, device_params)]
    for l in losses:  # Back Propagation is performed separately on each GPU
        l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    with torch.no_grad():
        for i in range(len(device_params[0])):
            allreduce([device_params[c][i].grad for c in range(len(devices))])
    # The model parameters are updated separately on each GPU
    for param in device_params:
        d2l.sgd(param, lr, X.shape[0]) # Here, we use a full-size batch
```

Agora, podemos definir a função de treinamento. É um pouco diferente dos usados nos capítulos anteriores: precisamos alocar as GPUs e copiar todos os parâmetros do modelo para todos os dispositivos. Obviamente, cada lote é processado usando `train_batch` para lidar com várias GPUs. Por conveniência (e concisão do código), calculamos a precisão em uma única GPU (isso é *ineficiente*, pois as outras GPUs estão ociosas).

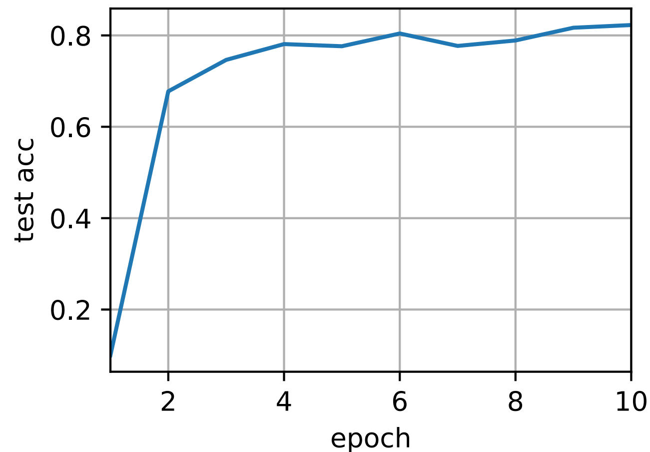
```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Copy model parameters to num_gpus GPUs
    device_params = [get_params(params, d) for d in devices]
    # num_epochs, times, acces = 10, [], []
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, device_params, devices, lr)
            torch.cuda.synchronize()
        timer.stop()
        # Verify the model on GPU 0
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, device_params[0]), test_iter, devices[0]),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

12.5.7 Experimento

Vamos ver como isso funciona bem em uma única GPU. Usamos um tamanho de lote de 256 e uma taxa de aprendizado de 0,2.

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

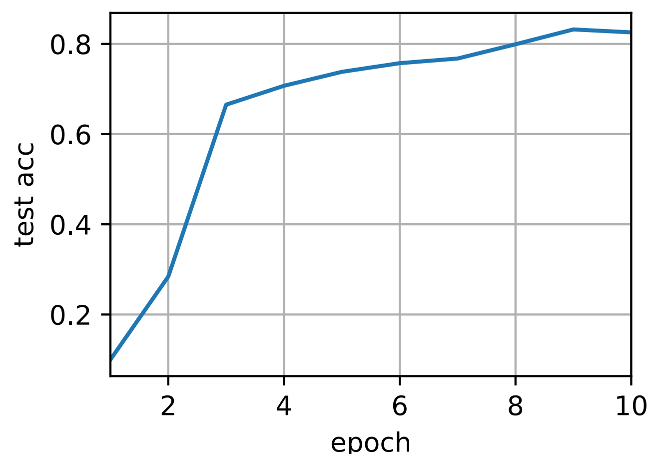
```
test acc: 0.82, 2.4 sec/epoch on [device(type='cuda', index=0)]
```



Mantendo o tamanho do lote e a taxa de aprendizado inalterados e alterando o número de GPUs para 2, podemos ver que a melhoria na precisão do teste é aproximadamente a mesma que nos resultados do experimento anterior. Em termos de algoritmos de otimização, eles são idênticos. Infelizmente, não há aumento significativo a ser obtido aqui: o modelo é simplesmente muito pequeno; além disso, temos apenas um pequeno conjunto de dados, onde nossa abordagem um pouco menos sofisticada para implementar o treinamento multi-GPU sofreu com a sobrecarga significativa do Python. Encontraremos modelos mais complexos e formas mais sofisticadas de paralelização daqui para frente. Vamos ver o que acontece, no entanto, com o Fashion-MNIST.

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

```
test acc: 0.83, 2.4 sec/epoch on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



12.5.8 Resumo

- Existem várias maneiras de dividir o treinamento de rede profunda em várias GPUs. Podemos dividi-los entre camadas, entre camadas ou entre dados. Os dois primeiros requerem transferências de dados fortemente coreografadas. O paralelismo de dados é a estratégia mais simples.
- O treinamento paralelo de dados é direto. No entanto, aumenta o tamanho efetivo do minibatch para ser eficiente.
- Os dados são divididos em várias GPUs, cada GPU executa sua própria operação de avanço e retrocesso e, posteriormente, os gradientes são agregados e os resultados transmitidos de volta às GPUs.
- Minibatches grandes podem exigir uma taxa de aprendizado ligeiramente maior.

12.5.9 Exercícios

1. Ao treinar em várias GPUs, altere o tamanho do minibatch de b para $k \cdot b$, ou seja, aumente pelo número de GPUs.
2. Compare a precisão para diferentes taxas de aprendizagem. Como isso se dimensiona com o número de GPUs.
3. Implemente um `allreduce` mais eficiente que agregue diferentes parâmetros em diferentes GPUs (por que isso é mais eficiente em primeiro lugar).
4. Implementar cálculo de precisão de teste multi-GPU.

Discussions[150]

12.6 Implementação Concisa para Várias GPUs

Implementar o paralelismo do zero para cada novo modelo não é divertido. Além disso, há um benefício significativo na otimização de ferramentas de sincronização para alto desempenho. A seguir, mostraremos como fazer isso usando o Gluon. A matemática e os algoritmos são os mesmos que em :numref:sec_multi_gpu. Como antes, começamos importando os módulos necessários (sem surpresa, você precisará de pelo menos duas GPUs para rodar este notebook).

```
import torch
from torch import nn
from d2l import torch as d2l
```

[150] https://discuss.d2l.ai/t/1669

12.6.1 Uma Rede de Exemplo

Vamos usar uma rede um pouco mais significativa do que a LeNet da seção anterior, que ainda é suficientemente fácil e rápida de treinar. Escolhemos uma variante do ResNet-18 (He et al., 2016). Como as imagens de entrada são pequenas, nós as modificamos ligeiramente. Em particular, a diferença para Section 7.6 é que usamos um kernel de convolução menor, stride e preenchimento no início. Além disso, removemos a camada de *pooling* máximo.

```
#@save
def resnet18(num_classes, in_channels=1):
    """A slightly modified ResNet-18 model."""
    def resnet_block(in_channels, out_channels, num_residuals,
                     first_block=False):
        blk = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.append(d2l.Residual(in_channels, out_channels,
                                        use_1x1conv=True, strides=2))
            else:
                blk.append(d2l.Residual(out_channels, out_channels))
        return nn.Sequential(*blk)

    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the maximum pooling layer
    net = nn.Sequential(
        nn.Conv2d(in_channels, 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU())
    net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
    net.add_module("resnet_block2", resnet_block(64, 128, 2))
    net.add_module("resnet_block3", resnet_block(128, 256, 2))
    net.add_module("resnet_block4", resnet_block(256, 512, 2))
    net.add_module("global_avg_pool", nn.AdaptiveAvgPool2d((1,1)))
    net.add_module("fc", nn.Sequential(nn.Flatten(),
                                       nn.Linear(512, num_classes)))
    return net
```

12.6.2 Inicialização de Parâmetros e Logística

O método `initialize` nos permite definir padrões iniciais para parâmetros em um dispositivo de nossa escolha. Para uma atualização, consulte Section 4.8. O que é particularmente conveniente é que também nos permite inicializar a rede em *vários* dispositivos simultaneamente. Vamos tentar como isso funciona na prática.

```
net = resnet18(10)
# get a list of GPUs
devices = d2l.try_all_gpus()
# we'll initialize the network inside the training loop
```

Usando a função `split_and_load` introduzida na seção anterior, podemos dividir um minibatch de dados e copiar porções para a lista de dispositivos fornecida pela variável de contexto. O objeto de rede *automaticamente* usa a GPU apropriada para calcular o valor da propagação direta. Como antes, geramos 4 observações e as dividimos nas GPUs.

Depois que os dados passam pela rede, os parâmetros correspondentes são inicializados *no dispositivo pelo qual os dados passaram*. Isso significa que a inicialização ocorre por dispositivo. Como escolhemos a GPU 0 e a GPU 1 para inicialização, a rede é inicializada apenas lá, e não na CPU. Na verdade, os parâmetros nem existem no dispositivo. Podemos verificar isso imprimindo os parâmetros e observando quaisquer erros que possam surgir.

Por último, vamos substituir o código para avaliar a precisão por um que funcione em paralelo em vários dispositivos. Isso serve como uma substituição da função `evaluate_accuracy_gpu` de Section 6.6. A principal diferença é que dividimos um lote antes de chamar a rede. Tudo o mais é essencialmente idêntico.

12.6.3 Treinamento

Como antes, o código de treinamento precisa realizar uma série de funções básicas para paralelismo eficiente:

- Os parâmetros de rede precisam ser inicializados em todos os dispositivos.
- Durante a iteração no conjunto de dados, os minibatches devem ser divididos em todos os dispositivos.
- Calculamos a perda e seu gradiente em paralelo entre os dispositivos.
- As perdas são agregadas (pelo método `trainer`) e os parâmetros são atualizados de acordo.

No final, calculamos a precisão (novamente em paralelo) para relatar o valor final da rede. A rotina de treinamento é bastante semelhante às implementações nos capítulos anteriores, exceto que precisamos dividir e agregar dados.

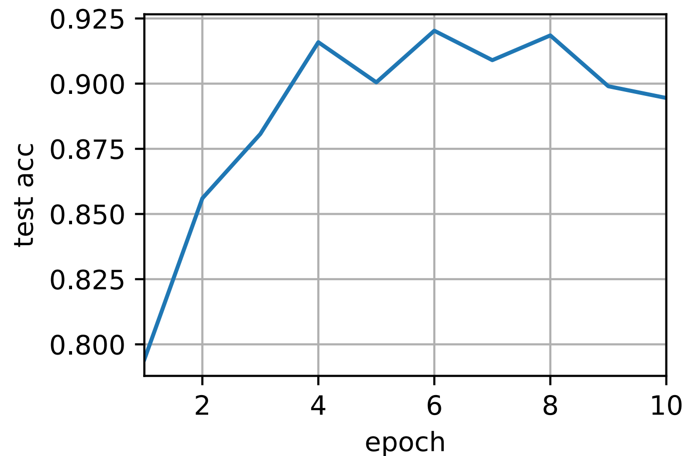
```
def train(net, num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    def init_weights(m):
        if type(m) in [nn.Linear, nn.Conv2d]:
            nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)
    # Set model on multiple gpus
    net = nn.DataParallel(net, device_ids=devices)
    trainer = torch.optim.SGD(net.parameters(), lr)
    loss = nn.CrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        net.train()
        timer.start()
        for X, y in train_iter:
            trainer.zero_grad()
            X, y = X.to(devices[0]), y.to(devices[0])
            l = loss(net(X), y)
            l.backward()
            trainer.step()
        timer.stop()
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(net, test_iter),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

12.6.4 Experimentos

Vamos ver como isso funciona na prática. Como aquecimento, treinamos a rede em uma única GPU.

```
train(net, num_gpus=1, batch_size=256, lr=0.1)
```

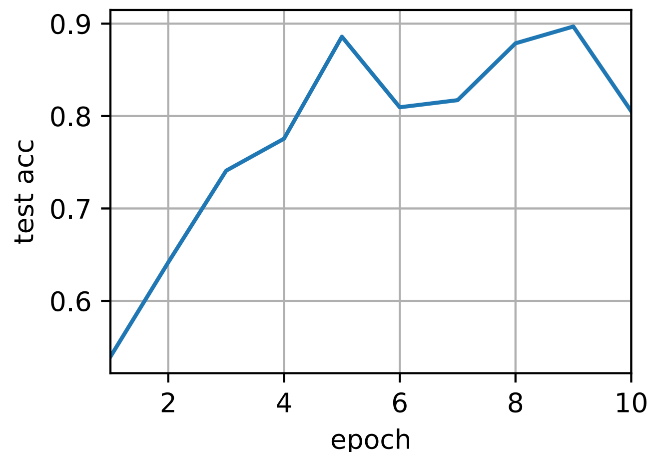
```
test acc: 0.89, 14.2 sec/epoch on [device(type='cuda', index=0)]
```



Em seguida, usamos 2 GPUs para treinamento. Comparado ao LeNet, o modelo do ResNet-18 é consideravelmente mais complexo. É aqui que a paralelização mostra sua vantagem. O tempo de cálculo é significativamente maior do que o tempo de sincronização de parâmetros. Isso melhora a escalabilidade, pois a sobrecarga para paralelização é menos relevante.

```
train(net, num_gpus=2, batch_size=512, lr=0.2)
```

```
test acc: 0.81, 9.3 sec/epoch on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



12.6.5 Resumo

- Gluon fornece primitivas para inicialização de modelo em vários dispositivos, fornecendo uma lista de contexto.
- Os dados são avaliados automaticamente nos dispositivos onde os dados podem ser encontrados.
- Tome cuidado ao inicializar as redes em cada dispositivo antes de tentar acessar os parâmetros naquele dispositivo. Caso contrário, você encontrará um erro.
- Os algoritmos de otimização agregam-se automaticamente em várias GPUs.

12.6.6 Exercícios

1. Esta seção usa o ResNet-18. Experimente diferentes épocas, tamanhos de lote e taxas de aprendizagem. Use mais GPUs para computação. O que acontece se você tentar isso em uma instância p2.16xlarge com 16 GPUs?
2. Às vezes, dispositivos diferentes fornecem poder de computação diferente. Poderíamos usar as GPUs e a CPU ao mesmo tempo. Como devemos dividir o trabalho? Vale a pena o esforço? Por quê? Por que não?
3. O que acontece se eliminarmos `npx.waitall()`? Como você modificaria o treinamento de forma que houvesse uma sobreposição de até duas etapas para o paralelismo?

Discussions[151]

12.7 Servidores de Parâmetros

À medida que mudamos de GPUs únicas para várias GPUs e depois para vários servidores contendo várias GPUs, possivelmente todos espalhadas por vários racks e switches de rede, nossos algoritmos para treinamento distribuído e paralelo precisam se tornar muito mais sofisticados. Os detalhes são importantes, já que diferentes interconexões têm larguras de banda muito diferentes (por exemplo, NVLink pode oferecer até 100 GB/s em 6 links em uma configuração apropriada, PCIe 3.0 16x lanes oferecem 16 GB/s, enquanto mesmo Ethernet de 100 GbE de alta velocidade atinge apenas 10 GB/s) . Ao mesmo tempo, não é razoável esperar que um modelador estatístico seja um especialista em redes e sistemas.

A ideia central do servidor de parâmetros foi introduzida em (Smola & Narayanamurthy, 2010) no contexto de modelos de variáveis latentes distribuídas. Uma descrição da semântica push e pull seguida em (Ahmed et al., 2012) e uma descrição do sistema e uma biblioteca de código aberto seguida em (Li et al., 2014). A seguir, iremos motivar os componentes necessários para a eficiência.

[151] https://discuss.d2l.ai/t/1403

12.7.1 Treinamento Paralelo de Dados

Vamos revisar a abordagem de treinamento paralelo de dados para treinamento distribuído. Usaremos isso com a exclusão de todos os outros nesta seção, uma vez que é significativamente mais simples de implementar na prática. Praticamente não há casos de uso (além do aprendizado profundo em gráficos) onde qualquer outra estratégia de paralelismo é preferida, já que as GPUs têm muita memória hoje em dia. Fig. 12.7.1 descreve a variante de paralelismo de dados que implementamos na seção anterior. O aspecto principal nisso é que a agregação de gradientes ocorre na GPU0 antes que os parâmetros atualizados sejam retransmitidos para todas as GPUs.

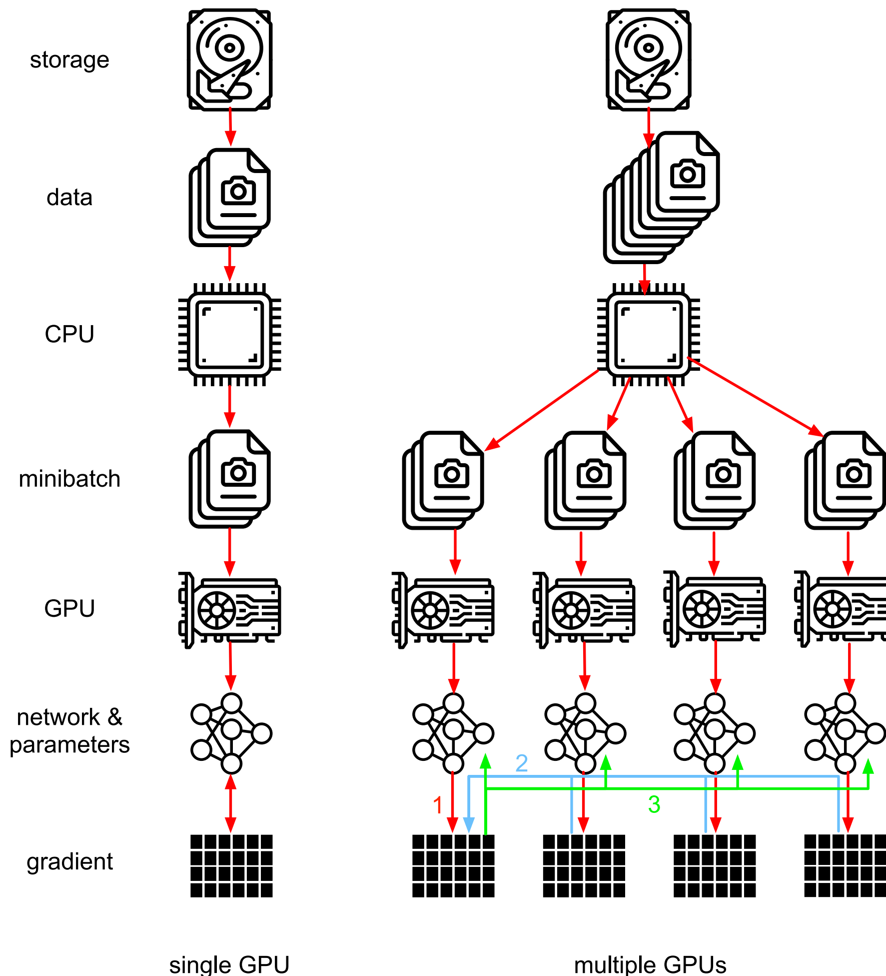


Fig. 12.7.1: Na esquerda: treinamento de GPU único; Na direita: uma variante do treinamento multi-GPU. Ele procede da seguinte maneira. (1) calculamos a perda e o gradiente, (2) todos os gradientes são agregados em uma GPU, (3) a atualização dos parâmetros acontece e os parâmetros são redistribuídos para todas as GPUs.

Em retrospecto, a decisão de agregar na GPU0 parece bastante ad-hoc. Afinal, podemos muito bem agregar na CPU. Na verdade, poderíamos até decidir agregar alguns dos parâmetros em uma GPU e alguns outros em outra. Desde que o algoritmo de otimização suporte isso, não há razão real para que não o possamos fazer. Por exemplo, se tivermos quatro vetores de parâmetro $\mathbf{v}_1, \ldots, \mathbf{v}_4$ com gradientes associados $\mathbf{g}_1, \ldots, \mathbf{g}_4$ poderíamos agreguar os gradientes em uma GPU cada.

$$\mathbf{g}_i = \sum_{j \in \text{GPUs}} \mathbf{g}_{ij} \tag{12.7.1}$$

Esse raciocínio parece arbitrário e frívolo. Afinal, a matemática é a mesma do começo ao fim. No entanto, estamos lidando com *hardware* físico real onde diferentes barramentos têm diferentes larguras de banda, conforme discutido em Section 12.4. Considere um servidor GPU real de 4 vias conforme descrito em Fig. 12.7.2. Se estiver bem conectado, pode ter uma placa de rede 100 GbE. Os números mais comuns estão na faixa de 1-10 GbE com uma largura de banda efetiva de 100 MB/s a 1 GB/s. Uma vez que as CPUs têm poucas pistas PCIe para se conectar a todas as GPUs diretamente (por exemplo, CPUs da Intel para consumidores têm 24 pistas), precisamos de um multiplexador[152]. A largura de banda da CPU em um link Gen3 16x é de 16 GB/s. Esta também é a velocidade na qual *cada* uma das GPUs é conectada ao *switch*. Isso significa que é mais eficaz a comunicação entre os dispositivos.

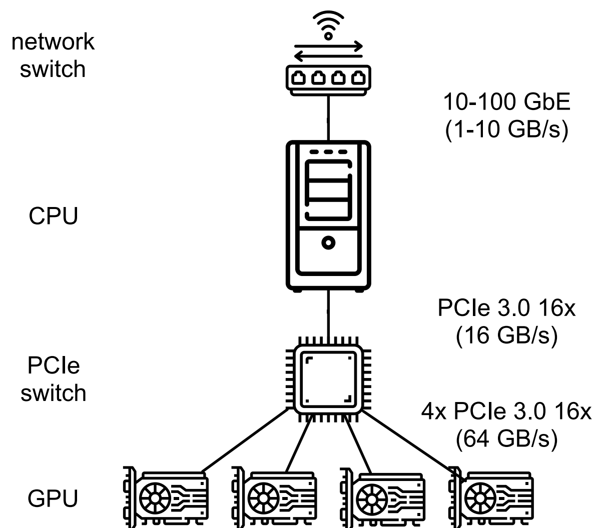


Fig. 12.7.2: Um servidor GPU de 4 vias.

Para o efeito do argumento, vamos supor que os gradientes 'pesam' 160 MB. Nesse caso, leva 30ms para enviar os gradientes de todas as 3 GPUs restantes para a quarta (cada transferência leva 10 ms = 160 MB / 16 GB/s). Adicione mais 30ms para transmitir os vetores de peso de volta, chegamos a um total de 60ms. Se enviarmos todos os dados para a CPU, incorremos em uma penalidade de 40ms, pois *cada* uma das quatro GPUs precisa enviar os dados para a CPU, resultando em um total de 80ms. Por último, suponha que somos capazes de dividir os gradientes em 4 partes de 40 MB cada. Agora podemos agregar cada uma das partes em uma GPU *diferente simultaneamente*, já que o switch PCIe oferece uma operação de largura de banda total entre todos os links. Em vez de 30 ms, isso leva 7,5 ms, resultando em um total de 15 ms para uma operação de sincronização. Resumindo, dependendo de como sincronizamos os parâmetros, a mesma operação pode levar de 15ms a 80ms. Fig. 12.7.3 descreve as diferentes estratégias para a troca de parâmetros.

[152] https://www.broadcom.com/products/pcie-switches-bridges/%20chaves%20pcie

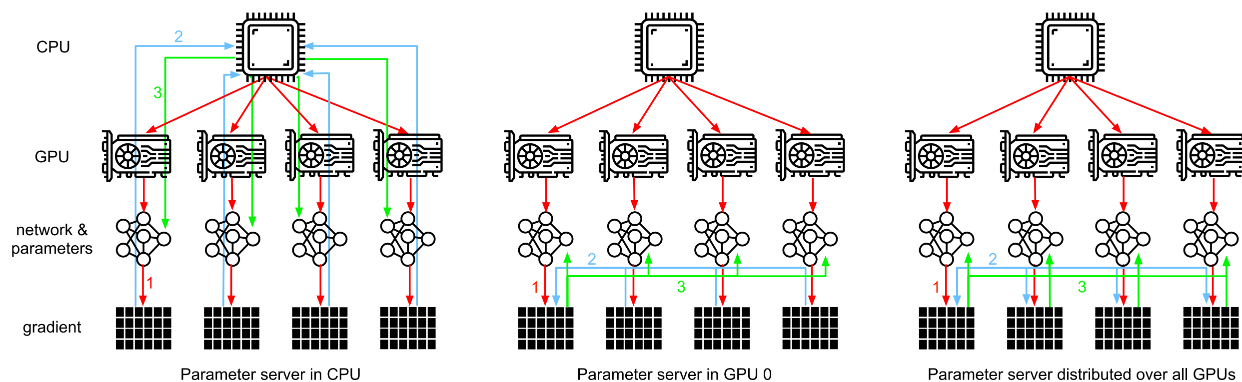


Fig. 12.7.3: Estratégias de sincronização.

Observe que temos mais uma ferramenta à nossa disposição quando se trata de melhorar o desempenho: em uma rede profunda, leva algum tempo para calcular todos os gradientes de cima para baixo. Podemos começar a sincronizar gradientes para alguns grupos de parâmetros, mesmo enquanto ainda estamos ocupados computando-os para outros (os detalhes técnicos disso estão um tanto envolvidos). Veja, por exemplo, (Sergeev & DelBalso, 2018) para obter detalhes sobre como fazer isso em Horovod[153].

12.7.2 Sincronização em Anel

Quando se trata de sincronização em *hardware* de *deep learning* moderno, frequentemente encontramos conectividade de rede significativamente personalizada. Por exemplo, as instâncias AWS P3.16xlarge e NVIDIA DGX-2 compartilham a estrutura de conectividade de Fig. 12.7.4. Cada GPU se conecta a uma CPU host por meio de um link PCIe que opera no máximo a 16 GB/s. Além disso, cada GPU também possui 6 conexões NVLink, cada uma das quais é capaz de transferir 300 Gbit/s bidirecionalmente. Isso equivale a cerca de 18 GB/s por link por direção. Resumindo, a largura de banda NVLink agregada é significativamente maior do que a largura de banda PCIe. A questão é como usá-lo de forma mais eficiente.

[153] https://github.com/horovod/horovod

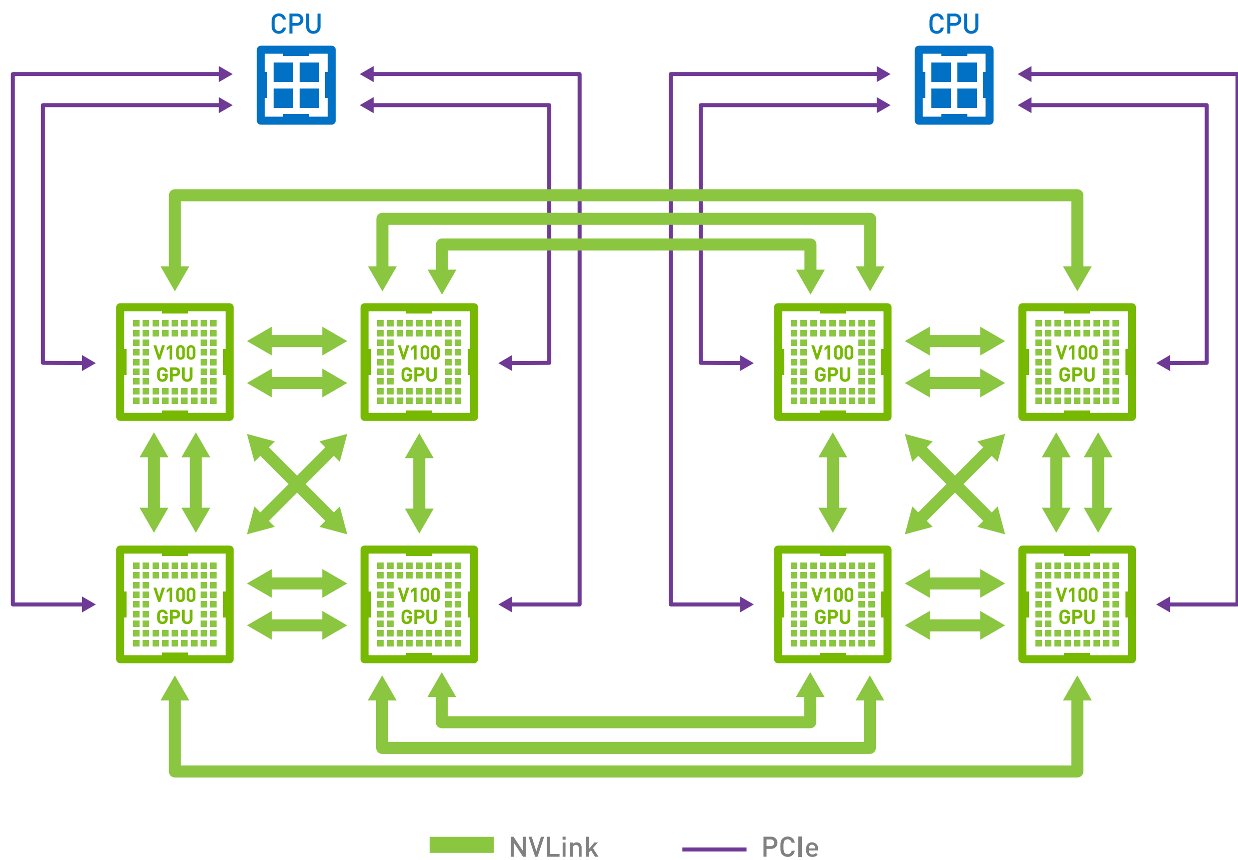


Fig. 12.7.4: Conectividade NVLink em servidores 8GPU V100 (imagem cortesia da NVIDIA).

Acontece que (Wang et al., 2018) a estratégia de sincronização ideal é decompor a rede em dois anéis e usá-los para sincronizar os dados diretamente. Fig. 12.7.5 ilustra que a rede pode ser decomposta em um anel (1-2-3-4-5-6-7-8-1) com largura de banda NVLink dupla e em um (1-4-6-3-5-8-2-7-1) com largura de banda regular. Projetar um protocolo de sincronização eficiente nesse caso não é trivial.

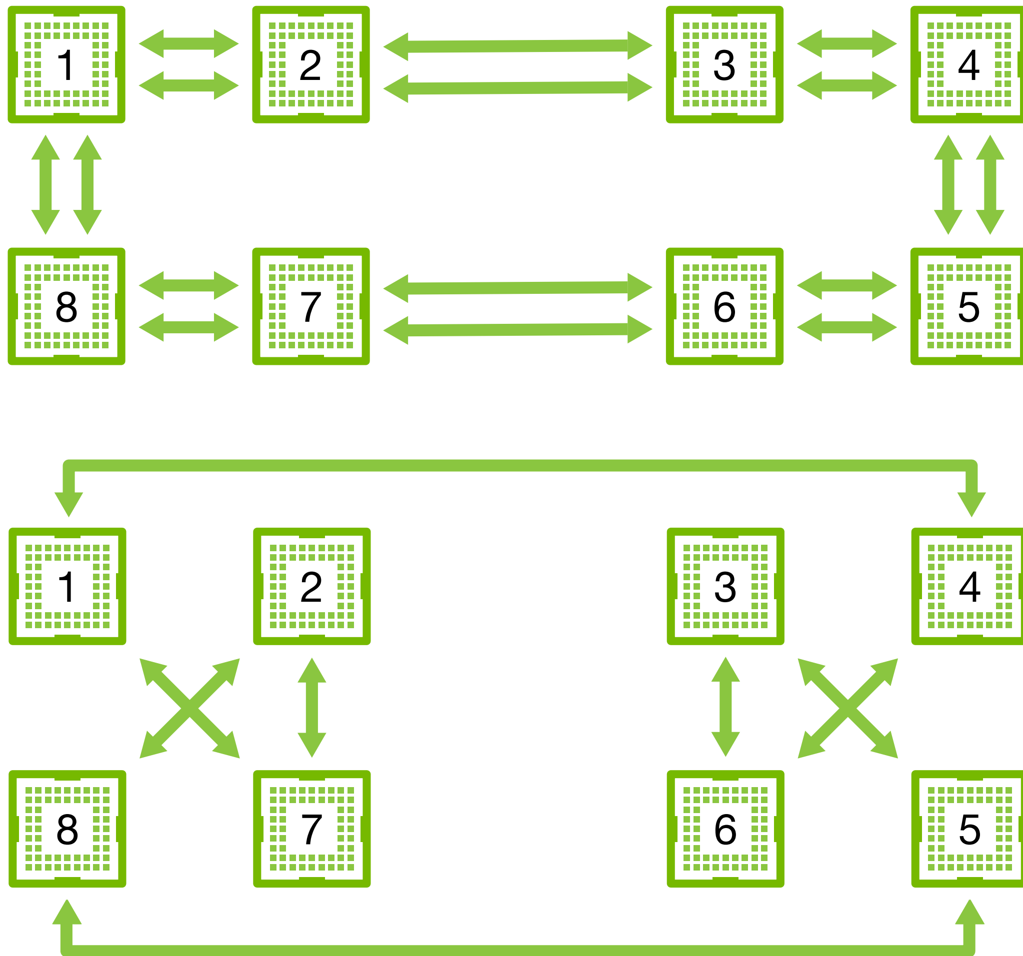


Fig. 12.7.5: Decomposição da rede NVLink em dois anéis.

Considere o seguinte experimento de pensamento: dado um anel de n nós de computação (ou GPUs), podemos enviar gradientes do primeiro para o segundo nó. Lá, ele é adicionado ao gradiente local e enviado para o terceiro nó e assim por diante. Após $n-1$ passos, o gradiente agregado pode ser encontrado no último nó visitado. Ou seja, o tempo para agregar gradientes cresce linearmente com o número de nós. Mas, se fizermos isso, o algoritmo será bastante ineficiente. Afinal, a qualquer momento, há apenas um dos nós se comunicando. E se quebrássemos os gradientes em n pedaços e começássemos a sincronizar o pedaço i começando no nó i. Como cada pedaço tem o tamanho $1/n$, o tempo total agora é $(n-1)/n \approx 1$ Em outras palavras, o tempo gasto para agregar gradientes *não aumenta* à medida que aumentamos o tamanho do anel. Este é um resultado surpreendente. Fig. 12.7.6 ilustra a sequência de etapas em $n = 4$ nodes.

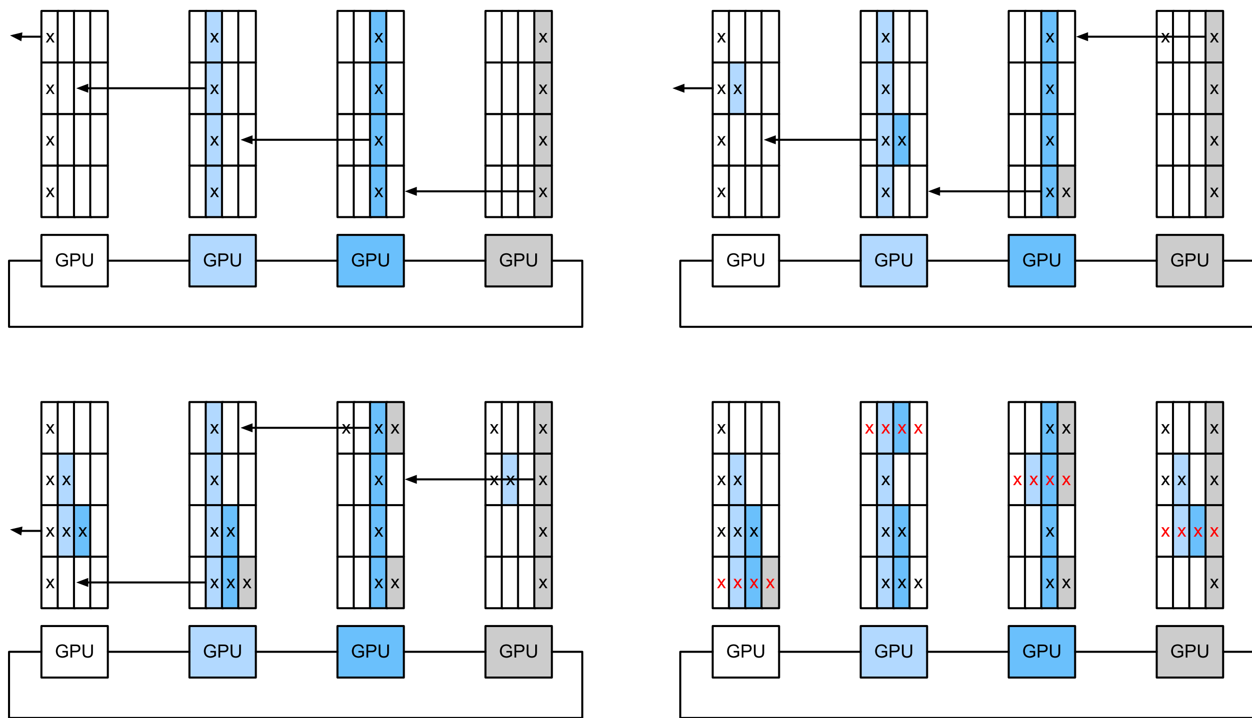


Fig. 12.7.6: Sincronização de anel em 4 nós. Cada nó começa a transmitir partes de gradientes para seu vizinho esquerdo até que o gradiente montado possa ser encontrado em seu vizinho direito.

Se usarmos o mesmo exemplo de sincronização de 160 MB em 8 GPUs V100, chegaremos a aproximadamente $2 \cdot 160\text{MB}/(3 \cdot 18\text{GB/s}) \approx 6\text{ms}$. Isto é um pouco melhor do que usar o barramento PCIe, embora agora estejamos usando 8 GPUs. Observe que, na prática, esses números são um pouco piores, uma vez que os *frameworks* de aprendizado profundo geralmente falham em reunir a comunicação em grandes transferências de burst. Além disso, o tempo é crítico. Observe que há um equívoco comum de que a sincronização em anel é fundamentalmente diferente de outros algoritmos de sincronização. A única diferença é que o caminho de sincronização é um pouco mais elaborado quando comparado a uma árvore simples.

12.7.3 Treinamento Multi-Máquina

O treinamento distribuído em várias máquinas adiciona mais um desafio: precisamos nos comunicar com servidores que estão conectados apenas por meio de uma malha de largura de banda comparativamente mais baixa, que pode ser mais do que uma ordem de magnitude mais lenta em alguns casos. A sincronização entre dispositivos é complicada. Afinal, diferentes máquinas que executam código de treinamento terão velocidades sutilmente diferentes. Portanto, precisamos *sincronizá-los* se quisermos usar a otimização distribuída síncrona. Fig. 12.7.7 ilustra como ocorre o treinamento paralelo distribuído.

1. Um lote (diferente) de dados é lido em cada máquina, dividido em várias GPUs e transferido para a memória da GPU. Essas previsões e gradientes são calculados em cada lote de GPU separadamente.
2. Os gradientes de todas as GPUs locais são agregados em uma GPU (ou, alternativamente, partes dela são agregadas em diferentes GPUs.
3. Os gradientes são enviados para a CPU.

4. A CPU envia os gradientes para um servidor de parâmetros central que agrega todos os gradientes.
5. Os gradientes agregados são então usados para atualizar os vetores de peso e os vetores de peso atualizados são transmitidos de volta para as CPUs individuais.
6. As informações são enviadas para uma (ou várias) GPUs.
7. Os vetores de peso atualizados são espalhados por todas as GPUs.

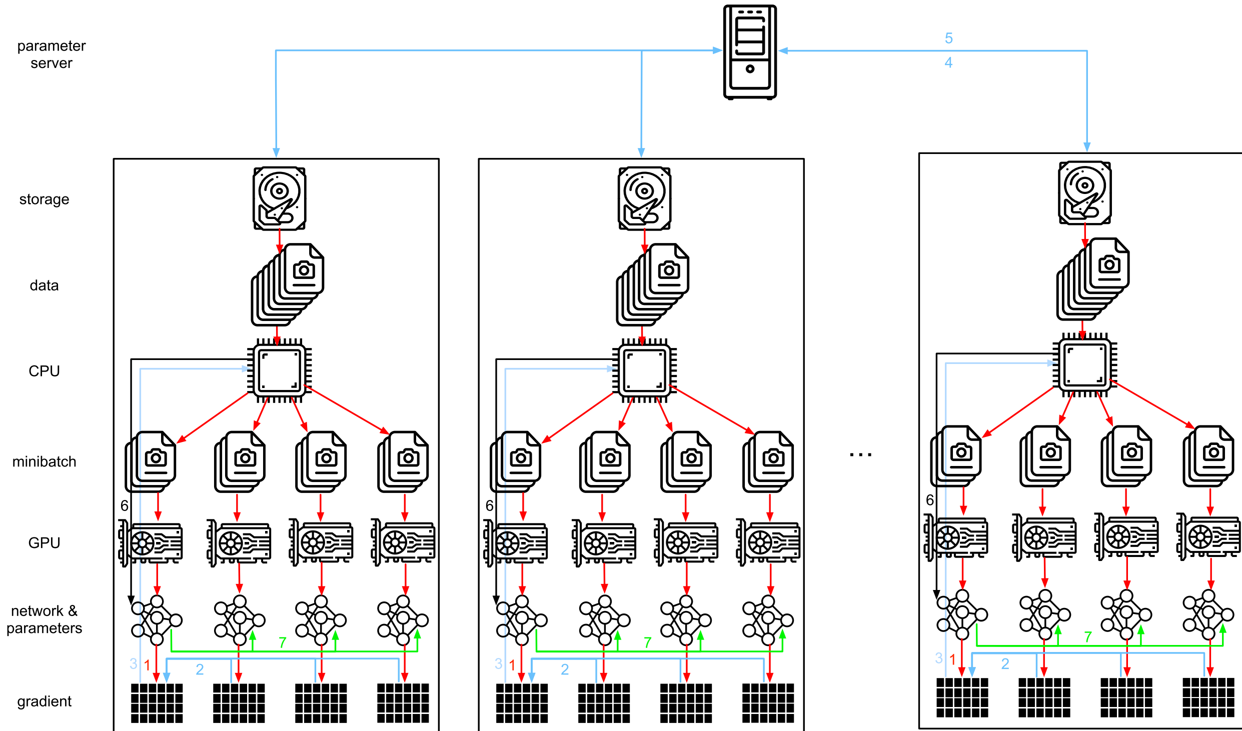


Fig. 12.7.7: Treinamento paralelo distribuído em várias máquinas e em várias GPUs.

Cada uma dessas operações parece bastante direta. E, de fato, eles podem ser realizadas com eficiência *dentro* de uma única máquina. Quando olhamos para várias máquinas, no entanto, podemos ver que o servidor de parâmetros central se torna o gargalo. Afinal, a largura de banda por servidor é limitada, portanto, para m trabalhadores, o tempo que leva para enviar todos os gradientes ao servidor é $O(m)$. Podemos quebrar essa barreira aumentando o número de servidores para n. Neste ponto, cada servidor só precisa armazenar $O(1/n)$ dos parâmetros, portanto, o tempo total para atualizações e otimização torna-se $O(m/n)$. Combinar os dois números resulta em um escalonamento constante, independentemente de quantos trabalhadores estamos lidando. Na prática, usamos as *mesmas* máquinas tanto como trabalhadores quanto como servidores. Fig. 12.7.8 ilustra o design. Veja também (Li et al., 2014) para detalhes. Em particular, garantir que várias máquinas funcionem sem atrasos excessivos não é trivial. Omitimos detalhes sobre barreiras e apenas abordaremos brevemente as atualizações síncronas e assíncronas abaixo.

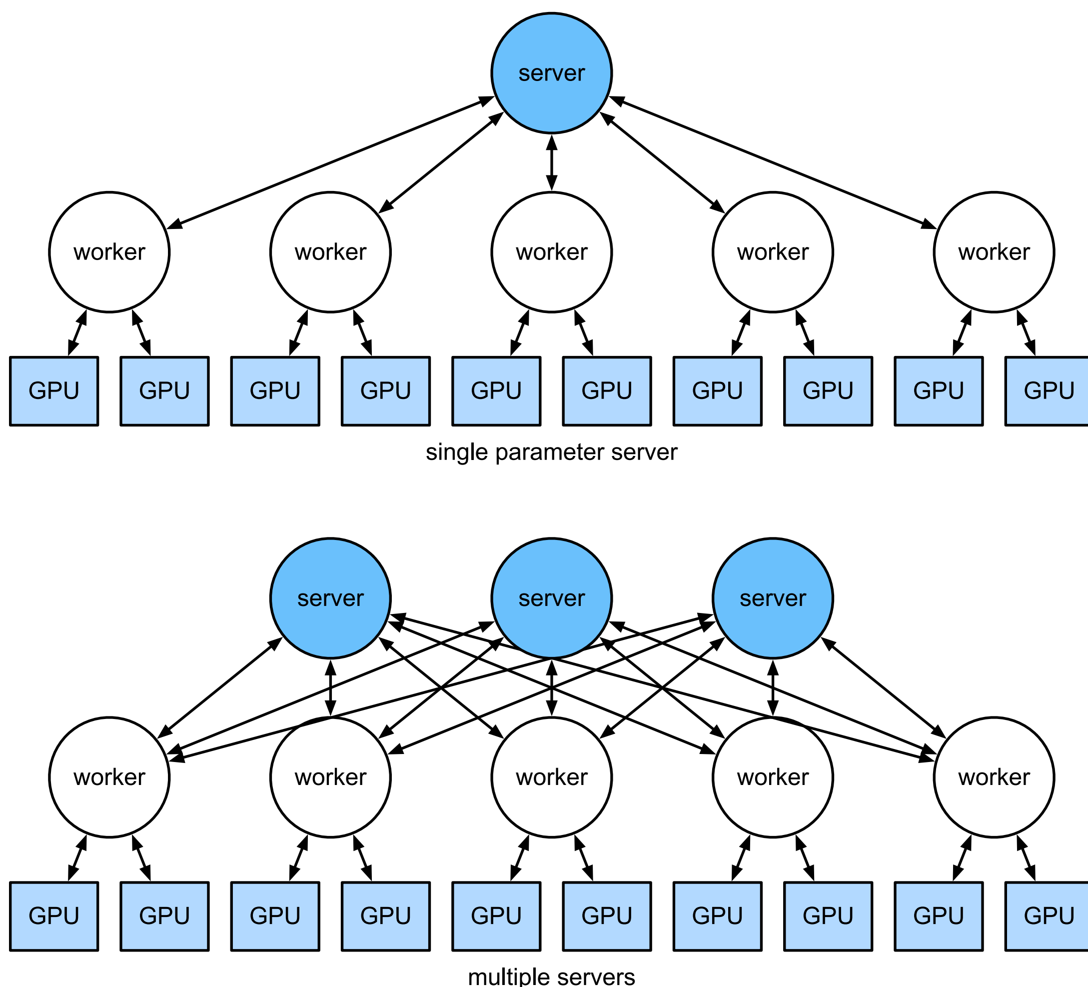


Fig. 12.7.8: Topo - um único servidor de parâmetro é um gargalo, pois sua largura de banda é finita. Parte inferior - servidores de vários parâmetros armazenam partes dos parâmetros com largura de banda agregada.

12.7.4 Armazenamento de (key,value)

A implementação das etapas necessárias para o treinamento distribuído de várias GPUs na prática não é trivial. Em particular, dadas as muitas opções diferentes que podemos encontrar. É por isso que vale a pena usar uma abstração comum, ou seja, a de um armazenamento (key, value) com semântica de atualização redefinida. Em muitos servidores e muitas GPUs, a computação de gradiente pode ser definida como

$$\mathbf{g}_{i} = \sum_{k \in \text{workers}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}. \tag{12.7.2}$$

O aspecto chave nesta operação é que se trata de uma *redução comutativa*, ou seja, ela transforma muitos vetores em um e a ordem de aplicação da operação não importa. Isso é ótimo para nossos propósitos, uma vez que não (precisamos) ter um controle refinado sobre quando qual gradiente é recebido. Observe que é possível realizarmos a redução em etapas. Além disso, observe que esta operação é independente entre os blocos i pertencentes a diferentes parâmetros (e gradientes).

Isso nos permite definir as duas operações a seguir: *push*, que acumula gradientes, e *pull*, que recupera gradientes agregados. Como temos muitos conjuntos diferentes de gradientes (afinal,

temos muitas camadas), precisamos indexar os gradientes com a chave i. Essa semelhança com armazenamento (key, value), como aquela introduzida no Dynamo (DeCandia et al., 2007) não é por acaso. Eles também satisfazem muitas características semelhantes, em particular quando se trata de distribuir os parâmetros em vários servidores.

- **push (key, value)** envia um gradiente específico (o valor) de um trabalhador para um armazenamento comum. Lá, o parâmetro é agregado, por exemplo, somando-o.
- **pull (key, value)** recupera um parâmetro agregado do armazenamento comum, por exemplo, depois de combinar os gradientes de todos os trabalhadores.

Ao ocultar toda a complexidade sobre a sincronização por trás de uma operação simples de *push* e *pull*, podemos dissociar as preocupações do modelador estatístico que deseja ser capaz de expressar a otimização em termos simples e do engenheiro de sistemas que precisa lidar com a complexidade inerente à sincronização distribuída. Na próxima seção, faremos experiências com esse armazenamento (key, value) na prática.

12.7.5 Resumo

- A sincronização precisa ser altamente adaptável à infraestrutura de rede específica e à conectividade em um servidor. Isso pode fazer uma diferença significativa no tempo que leva para sincronizar.
- A sincronização de anel pode ser ideal para servidores P3 e DGX-2. Para outros, possivelmente nem tanto.
- Uma estratégia de sincronização hierárquica funciona bem ao adicionar vários servidores de parâmetros para aumentar a largura de banda.
- A comunicação assíncrona (enquanto a computação ainda está em andamento) pode melhorar o desempenho.

12.7.6 Exercícios

1. Você pode aumentar ainda mais a sincronização do toque? Dica: você pode enviar mensagens em ambas as direções.
2. Totalmente assíncrono. Alguns atrasos são permitidos?
3. Tolerância a falhas. Como? E se perdermos um servidor? Isso é um problema?
4. *Checkpoint*
5. Agregação de árvores. Você pode fazer isso mais rápido?
6. Outras reduções (semirregamento comutativo).

Discussions[154]

[154] https://discuss.d2l.ai/t/366

13 | Visão Computacional

Muitas aplicações na área de visão computacional estão intimamente relacionadas às nossas vidas diárias, agora e no futuro, sejam diagnósticos médicos, veículos sem motorista, monitoramento de câmeras ou filtros inteligentes. Nos últimos anos, a tecnologia de aprendizado profundo melhorou muito o desempenho dos sistemas de visão computacional. Pode-se dizer que as aplicações de visão computacional mais avançadas são quase inseparáveis do aprendizado profundo.

Introduzimos modelos de aprendizagem profunda comumente usados na área de visão computacional no capítulo "Redes Neurais Convolucionais" e praticamos tarefas simples de classificação de imagens. Neste capítulo, apresentaremos os métodos de aumento e ajuste fino de imagens e os aplicaremos à classificação de imagens. Em seguida, exploraremos vários métodos de detecção de objetos. Depois disso, aprenderemos como usar redes totalmente convolucionais para realizar segmentação semântica em imagens. Em seguida, explicamos como usar a tecnologia de transferência de estilo para gerar imagens que se parecem com a capa deste livro. Finalmente, realizaremos exercícios práticos em dois importantes conjuntos de dados de visão computacional para revisar o conteúdo deste capítulo e dos capítulos anteriores.

13.1 Aumento de Imagem

Mencionamos que conjuntos de dados em grande escala são pré-requisitos para a aplicação bem-sucedida de redes neurais profundas em Section 7.1. A tecnologia de aumento de imagem expande a escala dos conjuntos de dados de treinamento, fazendo uma série de alterações aleatórias nas imagens de treinamento para produzir exemplos de treinamento semelhantes, mas diferentes. Outra maneira de explicar o aumento de imagem é que exemplos de treinamento que mudam aleatoriamente podem reduzir a dependência de um modelo em certas propriedades, melhorando assim sua capacidade de generalização. Por exemplo, podemos recortar as imagens de diferentes maneiras, para que os objetos de interesse apareçam em diferentes posições, reduzindo a dependência do modelo da posição onde os objetos aparecem. Também podemos ajustar o brilho, a cor e outros fatores para reduzir a sensibilidade do modelo à cor. Pode-se dizer que a tecnologia de aumento de imagem contribuiu muito para o sucesso do AlexNet. Nesta seção, discutiremos essa tecnologia, que é amplamente usada na visão computacional.

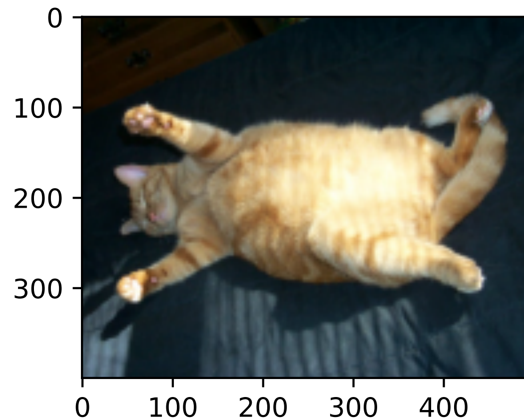
Primeiro, importe os pacotes ou módulos necessários para o experimento nesta seção.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.1.1 Método Comum de Aumento de Imagem

Neste experimento, usaremos uma imagem com um formato de 400×500 como exemplo.

```
d2l.set_figsize()
img = d2l.Image.open('../img/cat1.jpg')
d2l.plt.imshow(img);
```



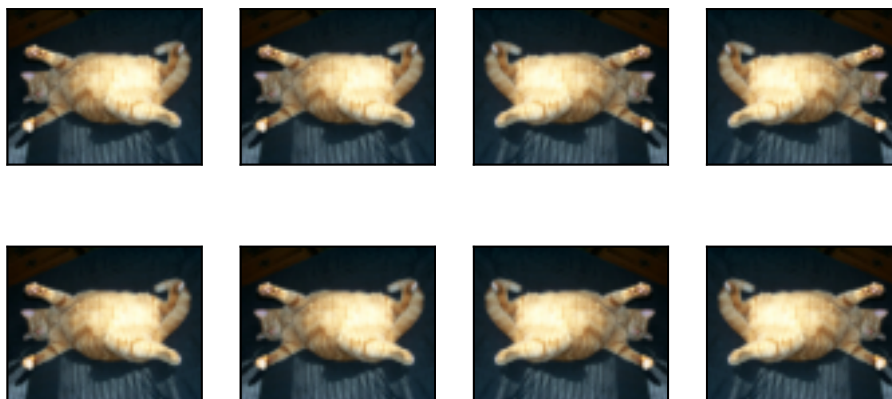
A maioria dos métodos de aumento de imagem tem um certo grau de aleatoriedade. Para facilitar a observação do efeito do aumento da imagem, definimos a seguir a função auxiliar `aplicar`. Esta função executa o método de aumento de imagem `aug` várias vezes na imagem de entrada`img` e mostra todos os resultados.

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

Invertendo e Recortando

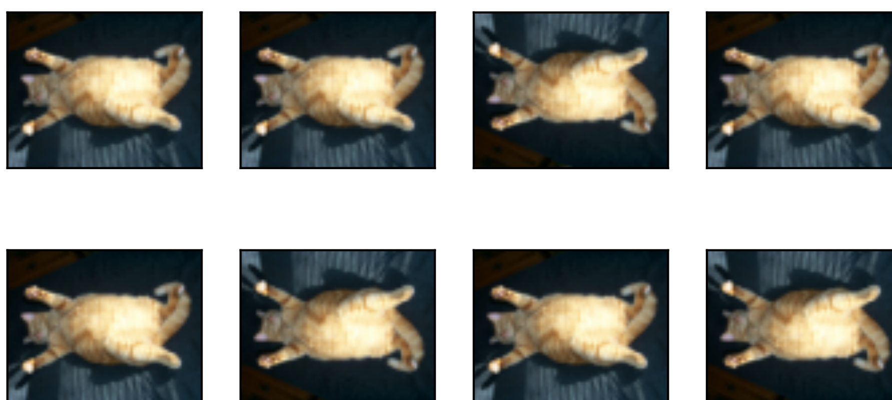
Virar a imagem para a esquerda e para a direita geralmente não altera a categoria do objeto. Este é um dos métodos mais antigos e mais amplamente usados de aumento de imagem. Em seguida, usamos o módulo `transforms` para criar a instância `RandomFlipLeftRight`, que apresenta uma chance de 50% de que a imagem seja virada para a esquerda e para a direita.

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```



Virar para cima e para baixo não é tão comumente usado como girar para a esquerda e para a direita. No entanto, pelo menos para esta imagem de exemplo, virar para cima e para baixo não impede o reconhecimento. Em seguida, criamos uma instância `RandomFlipTopBottom` para uma chance de 50% de virar a imagem para cima e para baixo.

```
apply(img, torchvision.transforms.RandomVerticalFlip())
```



Na imagem de exemplo que usamos, o gato está no meio da imagem, mas pode não ser o caso para todas as imagens. Em Section 6.5, explicamos que a camada de pooling pode reduzir a sensibilidade da camada convolucional ao local de destino. Além disso, podemos fazer os objetos aparecerem em diferentes posições na imagem em diferentes proporções aleatoriamente recortando a imagem. Isso também pode reduzir a sensibilidade do modelo à posição de destino.

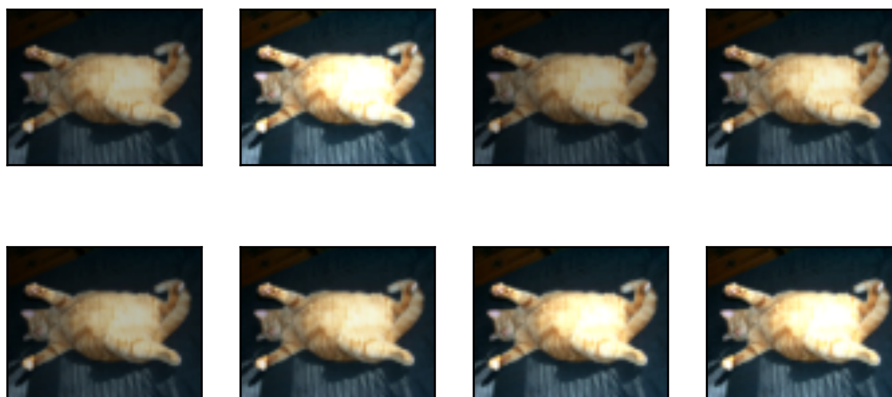
No código a seguir, recortamos aleatoriamente uma região com uma área de 10% a 100% da área original, e a proporção entre largura e altura da região é selecionada aleatoriamente entre 0,5 e 2. Em seguida, a largura e a altura de as regiões são dimensionadas para 200 pixels. Salvo indicação em contrário, o número aleatório entre a e b nesta seção refere-se a um valor contínuo obtido por amostragem uniforme no intervalo $[a, b]$.

```
shape_aug = torchvision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```

Mudando a Cor

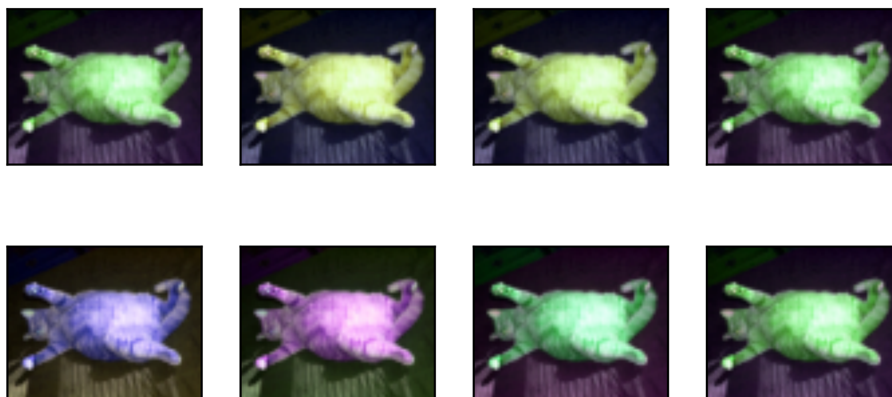
Outro método de aumento é mudar as cores. Podemos alterar quatro aspectos da cor da imagem: brilho, contraste, saturação e matiz. No exemplo abaixo, alteramos aleatoriamente o brilho da imagem para um valor entre 50% $(1 - 0.5)$ e 150% $(1 + 0.5)$ da imagem original.

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0, saturation=0, hue=0))
```



Da mesma forma, podemos alterar aleatoriamente o matiz da imagem.

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0, contrast=0, saturation=0, hue=0.5))
```



Também podemos criar uma instância `RandomColorJitter` e definir como alterar aleatoriamente o `brightness`, `contrast`, `saturation`, e `hue` da imagem ao mesmo tempo.

```
color_aug = torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```

Métodos de Aumento de Imagem Múltipla Sobreposta

Na prática, iremos sobrepor vários métodos de aumento de imagem. Podemos sobrepor os diferentes métodos de aumento de imagem definidos acima e aplicá-los a cada imagem usando uma instância `Compose`.

```
augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])
apply(img, augs)
```

13.1.2 Usando um Modelo de Treinamento de Aumento de Imagem

A seguir, veremos como aplicar o aumento de imagem no treinamento real. Aqui, usamos o conjunto de dados CIFAR-10, em vez do conjunto de dados Fashion-MNIST que usamos. Isso ocorre porque a posição e o tamanho dos objetos no conjunto de dados Fashion-MNIST foram normalizados, e as diferenças de cor e tamanho dos objetos no conjunto de dados CIFAR-10 são mais significativas. As primeiras 32 imagens de treinamento no conjunto de dados CIFAR-10 são mostradas abaixo.

```
all_images = torchvision.datasets.CIFAR10(train=True, root="../data",
                                          download=True)
d2l.show_images([all_images[i][0] for i in range(32)], 4, 8, scale=0.8);
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data/cifar-10-
↪python.tar.gz
9.6%
```

Para obter resultados definitivos durante a previsão, geralmente aplicamos apenas o aumento da imagem ao exemplo de treinamento e não usamos o aumento da imagem com operações aleatórias durante a previsão. Aqui, usamos apenas o método de inversão aleatório da esquerda para a direita mais simples. Além disso, usamos uma instância `ToTensor` para converter imagens de minibatch no formato exigido pelo MXNet, ou seja, números de ponto flutuante de 32 bits com a forma de (tamanho do lote, número de canais, altura, largura) e intervalo de valores entre 0 e 1.

```
train_augs = torchvision.transforms.Compose([
     torchvision.transforms.RandomHorizontalFlip(),
     torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
     torchvision.transforms.ToTensor()])
```

A seguir, definimos uma função auxiliar para facilitar a leitura da imagem e aplicar o aumento da imagem. A função `transform_first` fornecida pelo conjunto de dados do Gluon aplica o aumento da imagem ao primeiro elemento de cada exemplo de treinamento (imagem e rótulo), ou seja, o elemento na parte superior da imagem. Para descrições detalhadas de `DataLoader`, consulte Section 3.5.

```
def load_cifar10(is_train, augs, batch_size):
    dataset = torchvision.datasets.CIFAR10(root="../data", train=is_train,
                                           transform=augs, download=True)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                    shuffle=is_train, num_workers=d2l.get_dataloader_workers())
    return dataloader
```

Usando um Modelo de Treinamento Multi-GPU

Treinamos o modelo ResNet-18 descrito em: numref: `sec_resnet` no conjunto de dados CIFAR-10. Também aplicaremos os métodos descritos em Section 12.6 e usaremos um modelo de treinamento multi-GPU.

Em seguida, definimos a função de treinamento para treinar e avaliar o modelo usando várias GPUs.

```
#@save
def train_batch_ch13(net, X, y, loss, trainer, devices):
    if isinstance(X, list):
        # Required for BERT Fine-tuning (to be covered later)
        X = [x.to(devices[0]) for x in X]
    else:
        X = X.to(devices[0])
    y = y.to(devices[0])
    net.train()
    trainer.zero_grad()
    pred = net(X)
    l = loss(pred, y)
    l.sum().backward()
    trainer.step()
    train_loss_sum = l.sum()
    train_acc_sum = d2l.accuracy(pred, y)
    return train_loss_sum, train_acc_sum
```

```
#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               devices=d2l.try_all_gpus()):
    timer, num_batches = d2l.Timer(), len(train_iter)
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        # Store training_loss, training_accuracy, num_examples, num_features
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, devices)
            metric.add(l, acc, labels.shape[0], labels.numel())
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[3],
```

(continues on next page)

(continued from previous page)

```
                          None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {metric[0] / metric[2]:.3f}, train acc '
          f'{metric[1] / metric[3]:.3f}, test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on '
          f'{str(devices)}')
```

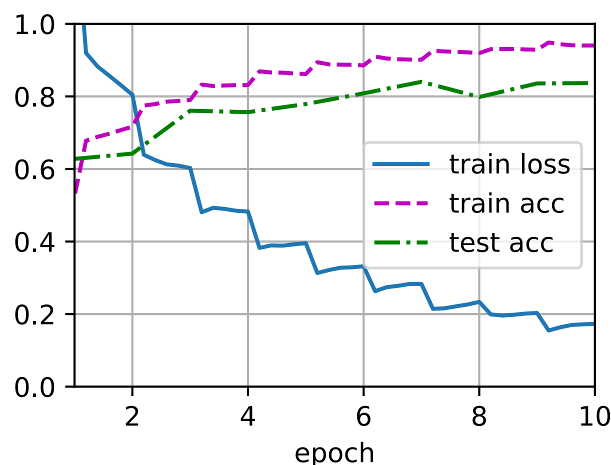
Agora, podemos definir a função `train_with_data_aug` para usar o aumento da imagem para treinar o modelo. Esta função obtém todas as GPUs disponíveis e usa Adam como o algoritmo de otimização para o treinamento. Em seguida, ele aplica o aumento da imagem ao conjunto de dados de treinamento e, finalmente, chama a função `train_ch13` definida para treinar e avaliar o modelo.

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10, 3)

def init_weights(m):
    if type(m) in [nn.Linear, nn.Conv2d]:
        nn.init.xavier_uniform_(m.weight)

net.apply(init_weights)

def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = nn.CrossEntropyLoss(reduction="none")
    trainer = torch.optim.Adam(net.parameters(), lr=lr)
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

Now we train the model using image augmentation of random flipping left and right.

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.173, train acc 0.940, test acc 0.837
5044.9 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



13.1.3 Resumo

- O aumento de imagem gera imagens aleatórias com base nos dados de treinamento existentes para lidar com o sobreajuste.
- Para obter resultados definitivos durante a previsão, geralmente aplicamos apenas o aumento da imagem ao exemplo de treinamento e não usamos o aumento da imagem com operações aleatórias durante a previsão.
- Podemos obter classes relacionadas ao aumento de imagem do módulo `transforma` do Gluon.

13.1.4 Exercícios

1. Treine o modelo sem usar aumento de imagem: `train_with_data_aug (no_aug, no_aug)`. Compare a precisão do treinamento e do teste ao usar e não usar o aumento de imagem. Este experimento comparativo pode apoiar o argumento de que o aumento da imagem pode mitigar o sobreajuste? Por quê?
2. Adicione diferentes métodos de aumento de imagem no treinamento do modelo com base no *dataset* CIFAR-10. Observe os resultados da implementação.
3. Com referência à documentação do MXNet, que outros métodos de aumento de imagem são fornecidos no módulo `transforms` do Gluon?

Discussões[155]

13.2 Ajustes

Nos capítulos anteriores, discutimos como treinar modelos no conjunto de dados de treinamento Fashion-MNIST, que tem apenas 60.000 imagens. Também descrevemos o ImageNet, o conjunto de dados de imagens em grande escala mais usado no mundo acadêmico, com mais de 10 milhões de imagens e objetos de mais de 1000 categorias. No entanto, o tamanho dos conjuntos de dados com os quais frequentemente lidamos é geralmente maior do que o primeiro, mas menor do que o segundo.

Suponha que queremos identificar diferentes tipos de cadeiras nas imagens e, em seguida, enviar o link de compra para o usuário. Um método possível é primeiro encontrar cem cadeiras comuns, obter mil imagens diferentes com diferentes ângulos para cada cadeira e, em seguida, treinar um modelo de classificação no conjunto de dados de imagens coletado. Embora esse conjunto de dados possa ser maior do que o Fashion-MNIST, o número de exemplos ainda é menor que um décimo do ImageNet. Isso pode resultar em sobreajuste do modelo complicado aplicável ao ImageNet neste conjunto de dados. Ao mesmo tempo, devido à quantidade limitada de dados, a precisão do modelo final treinado pode não atender aos requisitos práticos.

Para lidar com os problemas acima, uma solução óbvia é coletar mais dados. No entanto, coletar e rotular dados pode consumir muito tempo e dinheiro. Por exemplo, para coletar os conjuntos de dados ImageNet, os pesquisadores gastaram milhões de dólares em financiamento de pesquisa. Embora, recentemente, os custos de coleta de dados tenham caído significativamente, os custos ainda não podem ser ignorados.

[155] https://discuss.d2l.ai/t/1404

Outra solução é aplicar o aprendizado de transferência para migrar o conhecimento aprendido do conjunto de dados de origem para o conjunto de dados de destino. Por exemplo, embora as imagens no ImageNet não tenham relação com cadeiras, os modelos treinados neste conjunto de dados podem extrair recursos de imagem mais gerais que podem ajudar a identificar bordas, texturas, formas e composição de objetos. Esses recursos semelhantes podem ser igualmente eficazes para o reconhecimento de uma cadeira.

Nesta seção, apresentamos uma técnica comum no aprendizado por transferência: o ajuste fino. Conforme mostrado em :numref:`fig_finetune, o ajuste fino consiste nas quatro etapas a seguir:

1. Pré-treine um modelo de rede neural, ou seja, o modelo de origem, em um conjunto de dados de origem (por exemplo, o conjunto de dados ImageNet).
2. Crie um novo modelo de rede neural, ou seja, o modelo de destino. Isso replica todos os designs de modelo e seus parâmetros no modelo de origem, exceto a camada de saída. Assumimos que esses parâmetros do modelo contêm o conhecimento aprendido com o conjunto de dados de origem e esse conhecimento será igualmente aplicável ao conjunto de dados de destino. Também assumimos que a camada de saída do modelo de origem está intimamente relacionada aos rótulos do conjunto de dados de origem e, portanto, não é usada no modelo de destino.
3. Adicione uma camada de saída cujo tamanho de saída é o número de categorias de conjunto de dados de destino ao modelo de destino e inicialize aleatoriamente os parâmetros do modelo desta camada.
4. Treine o modelo de destino em um conjunto de dados de destino, como um conjunto de dados de cadeira. Vamos treinar a camada de saída do zero, enquanto os parâmetros de todas as camadas restantes são ajustados com base nos parâmetros do modelo de origem.

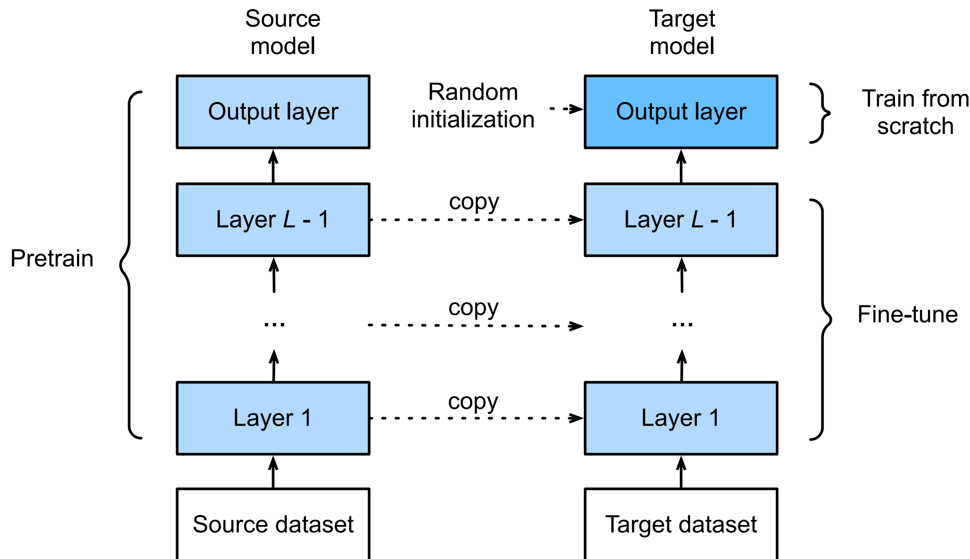


Fig. 13.2.1: Fine tuning.

13.2.1 Reconhecimento de Cachorro-quente

A seguir, usaremos um exemplo específico para prática: reconhecimento de cachorro-quente. Faremos o ajuste fino do modelo ResNet treinado no conjunto de dados ImageNet com base em um pequeno conjunto de dados. Este pequeno conjunto de dados contém milhares de imagens, algumas das quais contêm cachorros-quentes. Usaremos o modelo obtido pelo ajuste fino para identificar se uma imagem contém cachorro-quente.

Primeiro, importe os pacotes e módulos necessários para o experimento. O pacote `model_zoo` do Gluon fornece um modelo comum pré-treinado. Se você deseja obter mais modelos pré-treinados para visão computacional, você pode usar o GluonCV Toolkit[156].

```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

Obtendo o *Dataset*

O conjunto de dados de cachorro-quente que usamos foi obtido de imagens online e contém 1.400 imagens positivas contendo cachorros-quentes e o mesmo número de imagens negativas contendo outros alimentos. $1,000$ imagens de várias classes são usadas para treinamento e o resto é usado para teste.

Primeiro, baixamos o conjunto de dados compactado e obtemos duas pastas `hotdog/train` e `hotdog/test`. Ambas as pastas têm subpastas das categorias `hotdog` e `not-hotdog`, cada uma com arquivos de imagem correspondentes.

```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL+'hotdog.zip',
                         'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```

```
Downloading ../data/hotdog.zip from http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip...
```

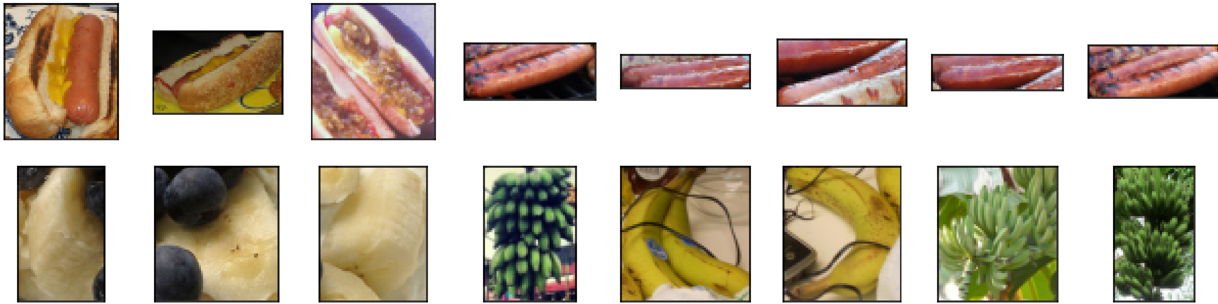
Criamos duas instâncias `ImageFolderDataset` para ler todos os arquivos de imagem no conjunto de dados de treinamento e teste, respectivamente.

```
train_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'train'))
test_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'test'))
```

Os primeiros 8 exemplos positivos e as últimas 8 imagens negativas são mostrados abaixo. Como você pode ver, as imagens variam em tamanho e proporção.

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

[156] https://gluon-cv.mxnet.io



Durante o treinamento, primeiro recortamos uma área aleatória com tamanho e proporção aleatória da imagem e, em seguida, dimensionamos a área para uma entrada com altura e largura de 224 pixels. Durante o teste, dimensionamos a altura e a largura das imagens para 256 pixels e, em seguida, recortamos a área central com altura e largura de 224 pixels para usar como entrada. Além disso, normalizamos os valores dos três canais de cores RGB (vermelho, verde e azul). A média de todos os valores do canal é subtraída de cada valor e o resultado é dividido pelo desvio padrão de todos os valores do canal para produzir a saída.

```
# We specify the mean and variance of the three RGB channels to normalize the
# image channel
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    normalize])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])
```

Definindo e Inicializando o Modelo

Usamos o ResNet-18, que foi pré-treinado no conjunto de dados ImageNet, como modelo de origem. Aqui, especificamos `pretrained = True` para baixar e carregar automaticamente os parâmetros do modelo pré-treinado. Na primeira vez em que são usados, os parâmetros do modelo precisam ser baixados da Internet.

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
```

A instância do modelo de origem pré-treinada contém duas variáveis de membro: `features` e `output`. O primeiro contém todas as camadas do modelo, exceto a camada de saída, e o último é a camada de saída do modelo. O principal objetivo desta divisão é facilitar o ajuste fino dos parâmetros do modelo de todas as camadas, exceto a camada de saída. A variável membro `output` do modelo de origem é fornecida abaixo. Como uma camada totalmente conectada, ele transforma a saída final da camada de agrupamento média global do ResNet em uma saída de 1000 classes no conjunto de dados ImageNet.

```
pretrained_net.fc
```

```
Linear(in_features=512, out_features=1000, bias=True)
```

Em seguida, construímos uma nova rede neural para usar como modelo-alvo. Ele é definido da mesma forma que o modelo de origem pré-treinado, mas o número final de saídas é igual ao número de categorias no conjunto de dados de destino. No código abaixo, os parâmetros do modelo na variável membro `features` da instância do modelo de destino `finetune_net` são inicializados para modelar os parâmetros da camada correspondente do modelo de origem. Como os parâmetros do modelo em `features` são obtidos por pré-treinamento no conjunto de dados ImageNet, é bom o suficiente. Portanto, geralmente só precisamos usar pequenas taxas de aprendizado para "ajustar" esses parâmetros. Em contraste, os parâmetros do modelo na variável membro `output` são inicializados aleatoriamente e geralmente requerem uma taxa de aprendizado maior para aprender do zero. Suponha que a taxa de aprendizado na instância `Trainer` seja η e use uma taxa de aprendizado de 10η para atualizar os parâmetros do modelo na variável membro `output`.

```
finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);
# If `param_group=True`, the model parameters in fc layer will be updated
# using a learning rate ten times greater, defined in the trainer.
```

Ajustando o Modelo

Primeiro definimos uma função de treinamento `train_fine_tuning` que usa ajuste fino para que possa ser chamada várias vezes.

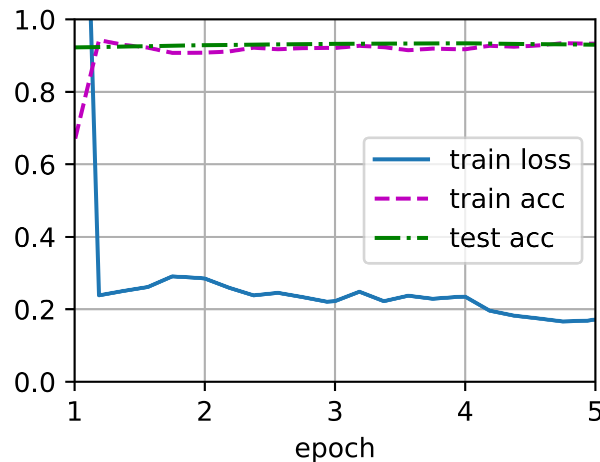
```
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5,
                      param_group=True):
    train_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'train'), transform=train_augs),
        batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'test'), transform=test_augs),
        batch_size=batch_size)
    devices = d2l.try_all_gpus()
    loss = nn.CrossEntropyLoss(reduction="none")
    if param_group:
        params_1x = [param for name, param in net.named_parameters()
             if name not in ["fc.weight", "fc.bias"]]
        trainer = torch.optim.SGD([{'params': params_1x},
                                   {'params': net.fc.parameters(),
                                    'lr': learning_rate * 10}],
                                lr=learning_rate, weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(), lr=learning_rate,
                                  weight_decay=0.001)
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                   devices)
```

Definimos a taxa de aprendizado na instância `Trainer` para um valor menor, como 0,01, a fim de ajustar os parâmetros do modelo obtidos no pré-treinamento. Com base nas configurações

anteriores, treinaremos os parâmetros da camada de saída do modelo de destino do zero, usando uma taxa de aprendizado dez vezes maior.

```
train_fine_tuning(finetune_net, 5e-5)
```

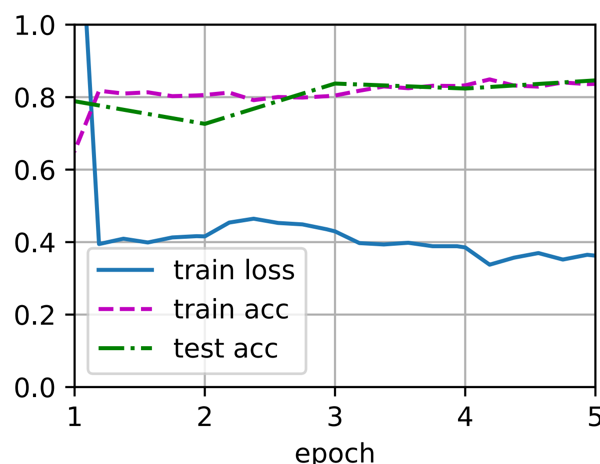
```
loss 0.172, train acc 0.933, test acc 0.930
827.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Para comparação, definimos um modelo idêntico, mas inicializamos todos os seus parâmetros de modelo para valores aleatórios. Como todo o modelo precisa ser treinado do zero, podemos usar uma taxa de aprendizado maior.

```
scratch_net = torchvision.models.resnet18()
scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.363, train acc 0.837, test acc 0.846
1613.8 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Como você pode ver, o modelo ajustado tende a obter maior precisão na mesma época porque os

valores iniciais dos parâmetros são melhores.

13.2.2 Resumo

- A aprendizagem de transferência migra o conhecimento aprendido do conjunto de dados de origem para o conjunto de dados de destino. O ajuste fino é uma técnica comum para a aprendizagem por transferência.
- O modelo de destino replica todos os designs de modelo e seus parâmetros no modelo de origem, exceto a camada de saída, e ajusta esses parâmetros com base no conjunto de dados de destino. Em contraste, a camada de saída do modelo de destino precisa ser treinada do zero.
- Geralmente, os parâmetros de ajuste fino usam uma taxa de aprendizado menor, enquanto o treinamento da camada de saída do zero pode usar uma taxa de aprendizado maior.

13.2.3 Exercícios

1. Continue aumentando a taxa de aprendizado de `finetune_net`. Como a precisão do modelo muda?
2. Ajuste ainda mais os hiperparâmetros de `finetune_net` e `scratch_net` no experimento comparativo. Eles ainda têm precisões diferentes?
3. Defina os parâmetros em `finetune_net.features` para os parâmetros do modelo de origem e não os atualize durante o treinamento. O que vai acontecer? Você pode usar o seguinte código.

```
for param in finetune_net.parameters():
    param.requires_grad = False
```

4. Na verdade, também existe uma classe "hotdog" no conjunto de dados `ImageNet`. Seu parâmetro de peso correspondente na camada de saída pode ser obtido usando o código a seguir. Como podemos usar este parâmetro?

```
weight = pretrained_net.fc.weight
hotdog_w = torch.split(weight.data, 1, dim=0)[713]
hotdog_w.shape
```

```
torch.Size([1, 512])
```

Discussões[157]

[157] https://discuss.d2l.ai/t/1439

13.3 Detecção de Objetos e Caixas Delimitadoras

Na seção anterior, apresentamos muitos modelos para classificação de imagens. Nas tarefas de classificação de imagens, presumimos que haja apenas uma característica principal na imagem e nos concentramos apenas em como identificar a categoria de destino. No entanto, em muitas situações, existem várias características na imagem que nos interessam. Não queremos apenas classificá-las, mas também queremos obter suas posições específicas na imagem. Na visão computacional, nos referimos a tarefas como detecção de objetos (ou reconhecimento de objetos).

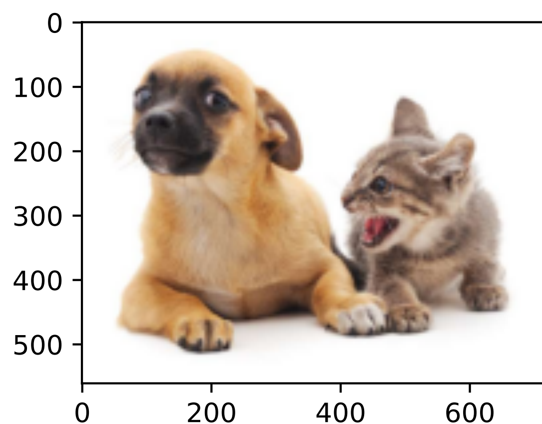
A detecção de objetos é amplamente usada em muitos campos. Por exemplo, na tecnologia de direção autônoma, precisamos planejar rotas identificando a localização de veículos, pedestres, estradas e obstáculos na imagem de vídeo capturada. Os robôs geralmente realizam esse tipo de tarefa para detectar alvos de interesse. Os sistemas no campo da segurança precisam detectar alvos anormais, como intrusos ou bombas.

Nas próximas seções, apresentaremos vários modelos de aprendizado profundo usados para detecção de objetos. Antes disso, devemos discutir o conceito de localização de destino. Primeiro, importe os pacotes e módulos necessários para o experimento.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

A seguir, carregaremos as imagens de amostra que serão usadas nesta seção. Podemos ver que há um cachorro no lado esquerdo da imagem e um gato no lado direito. Eles são os dois alvos principais desta imagem.

```
d2l.set_figsize()
img = d2l.plt.imread('../img/catdog.jpg')
d2l.plt.imshow(img);
```



13.3.1 Caixa Delimitadora

Na detecção de objetos, geralmente usamos uma caixa delimitadora para descrever o local de destino. A caixa delimitadora é uma caixa retangular que pode ser determinada pelas coordenadas dos eixos x e y no canto superior esquerdo e pelas coordenadas dos eixos x e y no canto inferior direito. Outra representação de caixa delimitadora comumente usada são as coordenadas dos eixos x e y do centro da caixa delimitadora e sua largura e altura. Aqui definimos funções para converter entre esses dois representações, `box_corner_to_center` converte da representação de dois cantos para a apresentação centro-largura-altura, e vice-versa `box_center_to_corner`. O argumento de entrada `boxes` pode ter um tensor de comprimento 4, ou um tensor $(N, 4)$ 2-dimensional.

```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper_left, bottom_right) to (center, width, height)"""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = torch.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper_left, bottom_right)"""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

Vamos definir as caixas delimitadoras do cão e do gato na imagem baseada nas informações de coordenadas. A origem das coordenadas na imagem é o canto superior esquerdo da imagem, e para a direita e para baixo estão as direções positivas do eixo x e do eixo y, respectivamente.

```
# bbox is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```

Podemos verificar a exatidão das funções de conversão da caixa convertendo duas vezes.

```
boxes = torch.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) - boxes
```

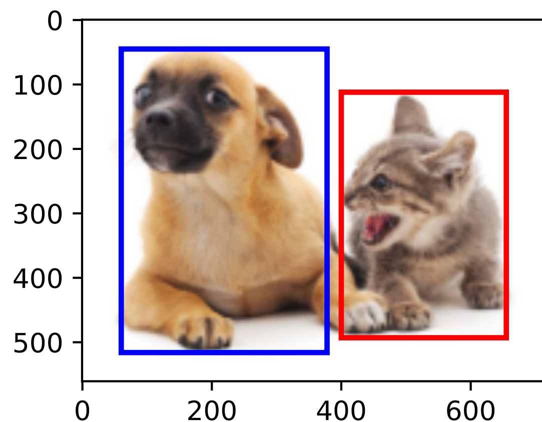
```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

Podemos desenhar a caixa delimitadora na imagem para verificar se ela é precisa. Antes de desenhar a caixa, definiremos uma função auxiliar `bbox_to_rect`. Ele representa a caixa delimitadora no formato de caixa delimitadora de `matplotlib`.

```python
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (top-left x, top-left y, bottom-right x,
    # bottom-right y) format to matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

Depois de carregar a caixa delimitadora na imagem, podemos ver que o contorno principal do alvo está basicamente dentro da caixa.

```python
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



13.3.2 Resumo

- Na detecção de objetos, não precisamos apenas identificar todos os objetos de interesse na imagem, mas também suas posições. As posições são geralmente representadas por uma caixa delimitadora retangular.

13.3.3 Exercícios

1. Encontre algumas imagens e tente rotular uma caixa delimitadora que contém o alvo. Compare a diferença entre o tempo que leva para rotular a caixa delimitadora e rotular a categoria.

Discussões[158]

[158] https://discuss.d2l.ai/t/1527

13.4 Caixas de Âncora

Os algoritmos de detecção de objetos geralmente amostram um grande número de regiões na imagem de entrada, determinam se essas regiões contêm objetos de interesse e ajustam as bordas das regiões de modo a prever a caixa delimitadora da verdade terrestre do alvo com mais precisão. Diferentes modelos podem usar diferentes métodos de amostragem de região. Aqui, apresentamos um desses métodos: ele gera várias caixas delimitadoras com diferentes tamanhos e proporções de aspecto, enquanto é centralizado em cada pixel. Essas caixas delimitadoras são chamadas de caixas de âncora. Praticaremos a detecção de objetos com base em caixas de âncora nas seções a seguir.

Primeiro, importe os pacotes ou módulos necessários para esta seção. Aqui, modificamos a precisão de impressão do PyTorch. Como os tensores de impressão, na verdade, chamam a função de impressão de PyTorch, os números de ponto flutuante nos tensores impressos nesta seção são mais concisos.

```
%matplotlib inline
import torch
from d2l import torch as d2l

torch.set_printoptions(2)
```

13.4.1 Gerando Várias Caixas de Âncora

Suponha que a imagem de entrada tenha uma altura de h e uma largura de w. Geramos caixas de âncora com diferentes formas centralizadas em cada pixel da imagem. Suponha que o tamanho seja $s \in (0, 1]$, a proporção da imagem é $r > 0$ e a largura e a altura da caixa de âncora são $ws\sqrt{r}$ e $hs/\sqrt{r}$, respectivamente. Quando a posição central é fornecida, uma caixa de âncora com largura e altura conhecidas é determinada.

Abaixo, definimos um conjunto de tamanhos $s_1, \ldots, s_n$ e um conjunto de relações de aspecto $r_1, \ldots, r_m$. Se usarmos uma combinação de todos os tamanhos e proporções com cada pixel como o centro, a imagem de entrada terá um total de $whnm$ caixas de âncora. Embora essas caixas de âncora possam abranger todas as caixas delimitadoras da verdade, a complexidade computacional costuma ser excessiva. Portanto, normalmente estamos interessados apenas em uma combinação contendo s_1 ou r_1 tamanhos e proporções, isto é:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1). \tag{13.4.1}$$

Ou seja, o número de caixas de âncora centradas no mesmo pixel é $n + m - 1$. Para toda a imagem de entrada, geraremos um total de $wh(n + m - 1)$ caixas de âncora.

O método acima para gerar caixas de âncora foi implementado na função `multibox_prior`. Especificamos a entrada, um conjunto de tamanhos e um conjunto de proporções, e esta função retornará todas as caixas de âncora inseridas.

```
#@save
def multibox_prior(data, sizes, ratios):
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
```

(continues on next page)

(continued from previous page)

```
    size_tensor = torch.tensor(sizes, device=device)
    ratio_tensor = torch.tensor(ratios, device=device)
    # Offsets are required to move the anchor to center of a pixel
    # Since pixel (height=1, width=1), we choose to offset our centers by 0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height  # Scaled steps in y axis
    steps_w = 1.0 / in_width  # Scaled steps in x axis

    # Generate all center points for the anchor boxes
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w)
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    # Generate boxes_per_pixel number of heights and widths which are later
    # used to create anchor box corner coordinates (xmin, xmax, ymin, ymax)
    # cat (various sizes, first ratio) and (first size, various ratios)
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:])))\
                   * in_height / in_width  # handle rectangular inputs
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    # Divide by 2 to get half height and half width
    anchor_manipulations = torch.stack((-w, -h, w, h)).T.repeat(
                                        in_height * in_width, 1) / 2

    # Each center point will have boxes_per_pixel number of anchor boxes, so
    # generate grid of all anchor box centers with boxes_per_pixel repeats
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                dim=1).repeat_interleave(boxes_per_pixel, dim=0)

    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)
```

Podemos ver que a forma da variável de caixa de âncora retornada y é (tamanho do lote, número de caixas de âncora, 4).

```
img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[0:2]

print(h, w)
X = torch.rand(size=(1, 3, h, w))  # Construct input data
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
```

```
torch.Size([1, 2042040, 4])
```

Depois de alterar a forma da variável da caixa de âncora y para (altura da imagem, largura da imagem, número de caixas de âncora centradas no mesmo pixel, 4), podemos obter todas as caixas de âncora centradas em uma posição de pixel especificada. No exemplo a seguir, acessamos a primeira caixa de âncora centrada em (250, 250). Ele tem quatro elementos: as coordenadas do

eixo x, y no canto superior esquerdo e as coordenadas do eixo x, y no canto inferior direito da caixa de âncora. Os valores das coordenadas dos eixos x e y são divididos pela largura e altura da imagem, respectivamente, portanto, o intervalo de valores está entre 0 e 1.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

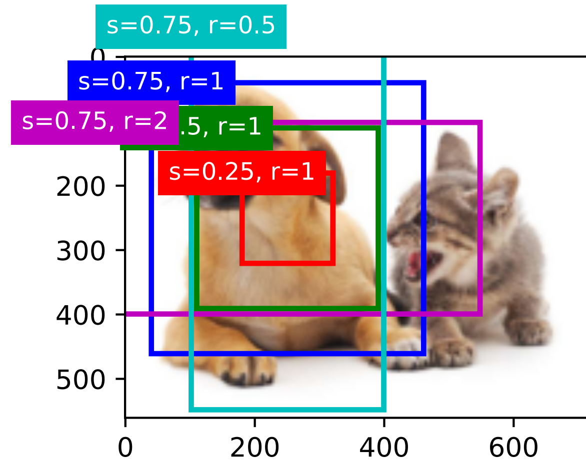
```
tensor([0.06, 0.07, 0.63, 0.82])
```

Para descrever todas as caixas de âncora centralizadas em um pixel na imagem, primeiro definimos a função `show_bboxes` para desenhar várias caixas delimitadoras na imagem.

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj
    labels = _make_list(labels)
    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.detach().numpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

Como acabamos de ver, os valores das coordenadas dos eixos x e y na variável `caixas` foram divididos pela largura e altura da imagem, respectivamente. Ao desenhar imagens, precisamos restaurar os valores das coordenadas originais das caixas de âncora e, portanto, definir a variável `bbox_scale`. Agora, podemos desenhar todas as caixas de âncora centralizadas em (250, 250) na imagem. Como você pode ver, a caixa de âncora azul com um tamanho de 0,75 e uma proporção de 1 cobre bem o cão na imagem.

```
d2l.set_figsize()
bbox_scale = torch.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```



13.4.2 Interseção sobre União

Acabamos de mencionar que a caixa de âncora cobre bem o cachorro na imagem. Se a caixa delimitadora da verdade básica do alvo é conhecida, como o "bem" pode ser quantificado aqui? Um método intuitivo é medir a semelhança entre as caixas de âncora e a caixa delimitadora da verdade absoluta. Sabemos que o índice de Jaccard pode medir a semelhança entre dois conjuntos. Dados os conjuntos $\mathcal{A}$ e $\mathcal{B}$, seu índice de Jaccard é o tamanho de sua interseção dividido pelo tamanho de sua união:

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}. \tag{13.4.2}$$

Na verdade, podemos considerar a área de pixels de uma caixa delimitadora como uma coleção de pixels. Dessa forma, podemos medir a similaridade das duas caixas delimitadoras pelo índice de Jaccard de seus conjuntos de pixels. Quando medimos a similaridade de duas caixas delimitadoras, geralmente nos referimos ao índice de Jaccard como interseção sobre união (IoU), que é a razão entre a área de interseção e a área de união das duas caixas delimitadoras, conforme mostrado em Fig. 13.4.1. O intervalo de valores de IoU está entre 0 e 1: 0 significa que não há pixels sobrepostos entre as duas caixas delimitadoras, enquanto 1 indica que as duas caixas delimitadoras são iguais.

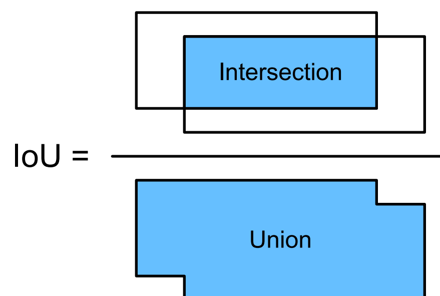


Fig. 13.4.1: IoU é a razão entre a área de interseção e a área de união de duas caixas delimitadoras.

Para o restante desta seção, usaremos IoU para medir a semelhança entre as caixas de âncora e as caixas delimitadoras de verdade terrestre e entre as diferentes caixas de âncora.

```python
#@save
def box_iou(boxes1, boxes2):
    """Compute IOU between two sets of boxes of shape (N,4) and (M,4)."""
    # Compute box areas
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    area1 = box_area(boxes1)
    area2 = box_area(boxes2)
    lt = torch.max(boxes1[:, None, :2], boxes2[:, :2])  # [N,M,2]
    rb = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])  # [N,M,2]
    wh = (rb - lt).clamp(min=0)  # [N,M,2]
    inter = wh[:, :, 0] * wh[:, :, 1]  # [N,M]
    unioun = area1[:, None] + area2 - inter
    return inter / unioun
```

13.4.3 Rotulagem de Treinamento para Definir Caixas de Âncora

No conjunto de treinamento, consideramos cada caixa de âncora como um exemplo de treinamento. Para treinar o modelo de detecção de objetos, precisamos marcar dois tipos de rótulos para cada caixa de âncora: primeiro, a categoria do alvo contido na caixa de âncora (categoria) e, em segundo lugar, o deslocamento da caixa delimitadora da verdade básica em relação à caixa de âncora (deslocamento). Na detecção de objetos, primeiro geramos várias caixas de âncora, predizemos as categorias e deslocamentos para cada caixa de âncora, ajustamos a posição da caixa de âncora de acordo com o deslocamento previsto para obter as caixas delimitadoras a serem usadas para previsão e, finalmente, filtramos as caixas delimitadoras de predição que precisam ser produzidos.

Sabemos que, no conjunto de treinamento de detecção de objetos, cada imagem é rotulada com a localização da caixa delimitadora da verdade terrestre e a categoria do alvo contido. Depois que as caixas de âncora são geradas, rotulamos principalmente as caixas de âncora com base na localização e nas informações de categoria das caixas delimitadoras de verdade terrestre semelhantes às caixas de âncora. Então, como atribuímos caixas delimitadoras de verdade terrestre a caixas de ancoragem semelhantes a elas?

Suponha que as caixas de ancoragem na imagem sejam $A_1, A_2, \ldots, A_{n_a}$ e as caixas delimitadoras de verdade são $B_1, B_2, \ldots, B_{n_b}$ e $n_a \geq n_b$. Defina a matriz $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$, onde o elemento x_{ij} na linha i^th e coluna j^th é a IoU da caixa de âncora A_i para a caixa delimitadora da verdade básica B_j. Primeiro, encontramos o maior elemento na matriz $\mathbf{X}$ e registramos o índice da linha e o índice da coluna do elemento como i_1, j_1. Atribuímos a caixa delimitadora da verdade básica B_{j_1} à caixa âncora A_{i_1}. Obviamente, a caixa de âncora A_{i_1} e a caixa delimitadora da verdade básica B_{j_1} têm a maior similaridade entre todos os pares "caixa de âncora - caixa delimitadora da verdade". A seguir, descarte todos os elementos da ${i_1}^\mathrm{a}$ linha e da ${j_1}^\mathrm{a}$ coluna da matriz $\mathbf{X}$. Encontre o maior elemento restante na matriz $\mathbf{X}$ e registre o índice da linha e o índice da coluna do elemento como i_2, j_2. Atribuímos a caixa delimitadora de verdade básica B_{j_2} à caixa de ancoragem A_{i_2} e, em seguida, descartamos todos os elementos na ${i_2}^\mathrm{a}$ linha e na ${j_2}^\mathrm{a}$ coluna na matriz $\mathbf{X}$. Neste ponto, os elementos em duas linhas e duas colunas na matriz $\mathbf{X}$ foram descartados.

Prosseguimos até que todos os elementos da coluna n_b da matriz $\mathbf{X}$ sejam descartados. Neste momento, atribuímos uma caixa delimitadora de verdade terrestre a cada uma das caixas de âncora n_b. Em seguida, percorremos apenas as caixas de âncora $n_a - n_b$ restantes. Dada a caixa de âncora A_i, encontre a caixa delimitadora B_j com o maior IoU com A_i de acordo com a ith linha da matriz $\mathbf{X}$, e apenas atribua o terreno -caixa delimitadora da verdade B_j para ancorar a caixa A_i quando o

IoU é maior do que o limite predeterminado.

Conforme mostrado em Fig. 13.4.2 (esquerda), assumindo que o valor máximo na matriz **X** é x_{23}, iremos atribuir a caixa delimitadora da verdade básica B_3 à caixa de âncora A_2.. Em seguida, descartamos todos os elementos na linha 2 e coluna 3 da matriz, encontramos o maior elemento x_{71} da área sombreada restante e atribuímos a caixa delimitadora de verdade básica B_1 à caixa de ancoragem A_7. Então, como mostrado em Fig. 13.4.2 (meio), descarte todos os elementos na linha 7 e coluna 1 da matriz, encontre o maior elemento x_{54} da área sombreada restante e atribua a caixa delimitadora de verdade fundamental B_4 para a caixa âncora A_5. Finalmente, como mostrado em Fig. 13.4.2 (direita), descarte todos os elementos na linha 5 e coluna 4 da matriz, encontre o maior elemento x_{92} da área sombreada restante e atribua a caixa delimitadora da verdade fundamental B_2 para a caixa âncora A_9. Depois disso, só precisamos atravessar as caixas de âncora restantes de A_1, A_3, A_4, A_6, A_8 e determinar se devemos atribuir caixas delimitadoras de verdade fundamental às caixas de âncora restantes de acordo com o limite.

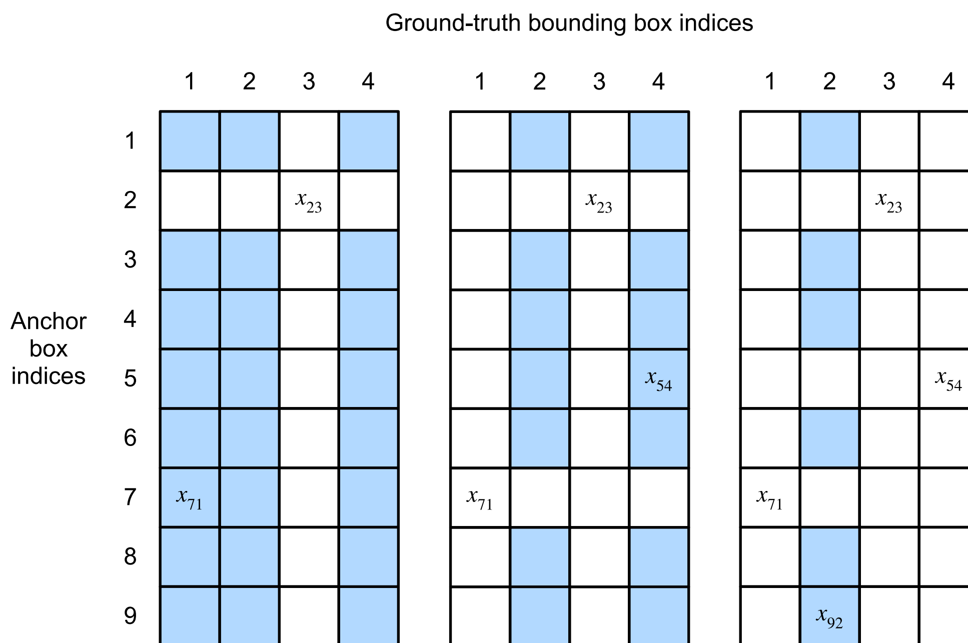


Fig. 13.4.2: Atribua caixas delimitadoras de base de verdade às caixas de ancoragem.

```
#@save
def match_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """Assign ground-truth bounding boxes to anchor boxes similar to them."""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # Element `x_ij` in the `i^th` row and `j^th` column is the IoU
    # of the anchor box `anc_i` to the ground-truth bounding box `box_j`
    jaccard = box_iou(anchors, ground_truth)
    # Initialize the tensor to hold assigned ground truth bbox for each anchor
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype=torch.long,
                                  device=device)
    # Assign ground truth bounding box according to the threshold
    max_ious, indices = torch.max(jaccard, dim=1)
    anc_i = torch.nonzero(max_ious >= 0.5).reshape(-1)
    box_j = indices[max_ious >= 0.5]
    anchors_bbox_map[anc_i] = box_j
    # Find the largest iou for each bbox
```

(continues on next page)

(continued from previous page)

```
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map
```

Agora podemos rotular as categorias e deslocamentos das caixas de âncora. Se uma caixa de âncora A for atribuída a uma caixa delimitadora de verdade fundamental B, a categoria da caixa de âncora A será definida como a categoria de B. E o deslocamento da caixa âncora A é definido de acordo com a posição relativa das coordenadas centrais de B e A e os tamanhos relativos das duas caixas. Como as posições e tamanhos de várias caixas no conjunto de dados podem variar, essas posições e tamanhos relativos geralmente requerem algumas transformações especiais para tornar a distribuição de deslocamento mais uniforme e fácil de ajustar. Suponha que as coordenadas centrais da caixa de âncora A e sua caixa delimitadora de verdade fundamental $ B $ sejam $(x_a, y_a), (x_b, y_b)$, as larguras de A e B são w_a, w_b, e suas alturas são h_a, h_b, respectivamente. Neste caso, uma técnica comum é rotular o deslocamento de A como

$$\left(\frac{ \frac{x_b - x_a}{w_a} - \mu_x }{\sigma_x}, \frac{ \frac{y_b - y_a}{h_a} - \mu_y }{\sigma_y}, \frac{ \log \frac{w_b}{w_a} - \mu_w }{\sigma_w}, \frac{ \log \frac{h_b}{h_a} - \mu_h }{\sigma_h}\right), \tag{13.4.3}$$

Os valores padrão da constante são $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1$, e $\sigma_w = \sigma_h = 0.2$. Esta transformação é implementada abaixo na função `offset_boxes`. Se uma caixa de âncora não for atribuída a uma caixa delimitadora de verdade, só precisamos definir a categoria da caixa de âncora como segundo plano. As caixas de âncora cuja categoria é o plano de fundo costumam ser chamadas de caixas de âncora negativas e o restante é chamado de caixas de âncora positivas.

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * torch.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = torch.cat([offset_xy, offset_wh], axis=1)
    return offset
```

```
#@save
def multibox_target(anchors, labels):
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = match_anchor_to_bbox(label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze(-1)).repeat(1, 4)
        # Initialize class_labels and assigned bbox coordinates with zeros
        class_labels = torch.zeros(num_anchors, dtype=torch.long,
                                   device=device)
```

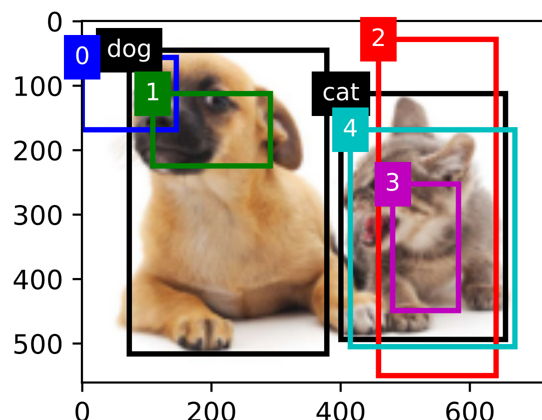
(continues on next page)

(continued from previous page)

```
        assigned_bb = torch.zeros((num_anchors, 4), dtype=torch.float32,
                                  device=device)
        # Assign class labels to the anchor boxes using matched gt bbox labels
        # If no gt bbox is assigned to an anchor box, then let the
        # class_labels and assigned_bb remain zero, i.e the background class
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # offset transformations
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

Abaixo, demonstramos um exemplo detalhado. Definimos caixas delimitadoras de verdade para o gato e o cachorro na imagem lida, onde o primeiro elemento é a categoria (0 para cachorro, 1 para gato) e os quatro elementos restantes são as coordenadas do eixo x, y no canto superior esquerdo canto e coordenadas do eixo x, y no canto inferior direito (o intervalo de valores está entre 0 e 1). Aqui, construímos cinco caixas de âncora para serem rotuladas pelas coordenadas do canto superior esquerdo e do canto inferior direito, que são registradas como $A_0, \ldots, A_4$, respectivamente (o índice no programa começa em 0) . Primeiro, desenhe as posições dessas caixas de âncora e das caixas delimitadoras da verdade fundamental na imagem.

```
ground_truth = torch.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                             [1, 0.55, 0.2, 0.9, 0.88]])
anchors = torch.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                        [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                        [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



Podemos rotular categorias e deslocamentos para caixas de âncora usando a função `multibox_target`. Esta função define a categoria de fundo para 0 e incrementa o índice inteiro da categoria de destino de zero por 1 (1 para cachorro e 2 para gato).

Adicionamos dimensões de exemplo às caixas de âncora e caixas delimitadoras de verdade e construímos resultados preditos aleatórios com uma forma de (tamanho do lote, número de categorias incluindo fundo, número de caixas de âncora) usando a função `unsqueeze`.

```
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

Existem três itens no resultado retornado, todos no formato tensor. O terceiro item é representado pela categoria rotulada para a caixa de âncora.

```
labels[2]
```

```
tensor([[0, 1, 2, 0, 2]])
```

Analisamos essas categorias rotuladas com base nas posições das caixas de âncora e das caixas delimitadoras de informações básicas na imagem. Em primeiro lugar, em todos os pares de "caixa de âncora - caixa delimitadora de verdade básica", a IoU da caixa de ancoragem A_4 para a caixa delimitadora de verdade básica do gato é a maior, então a categoria de caixa de ancoragem A_4 é rotulada como gato. Sem considerar a caixa de âncora A_4 ou a caixa delimitadora de verdade do solo do gato, nos pares restantes "caixa de âncora - caixa de ligação de verdade", o par com a maior IoU é a caixa de âncora A_1 e a a caixa delimitadora da verdade do cachorro, portanto, a categoria da caixa de âncora A_1 é rotulada como cachorro. Em seguida, atravesse as três caixas de âncora restantes sem etiqueta. A categoria da caixa delimitadora de verdade básica com o maior IoU com caixa de âncora A_0 é cahcorro, mas o IoU é menor que o limite (o padrão é 0,5), portanto, a categoria é rotulada como plano de fundo; a categoria da caixa delimitadora de verdade básica com a maior IoU com caixa de âncora A_2 é gato e a IoU é maior que o limite, portanto, a categoria é rotulada como gato; a categoria da caixa delimitadora de verdade básica com a maior IoU com caixa de âncora A_3 é cat, mas a IoU é menor que o limite, portanto, a categoria é rotulada como plano de fundo.

O segundo item do valor de retorno é uma variável de máscara, com a forma de (tamanho do lote, quatro vezes o número de caixas de âncora). Os elementos na variável de máscara correspondem um a um com os quatro valores de deslocamento de cada caixa de âncora. Como não nos importamos com a detecção de fundo, os deslocamentos da classe negativa não devem afetar a função de destino. Multiplicando por elemento, o 0 na variável de máscara pode filtrar os deslocamentos de classe negativos antes de calcular a função de destino.

```
labels[1]
```

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 1.,
         1., 1.]])
```

O primeiro item retornado são os quatro valores de deslocamento rotulados para cada caixa de âncora, com os deslocamentos das caixas de âncora de classe negativa rotulados como 0.

```
labels[0]
```

```
tensor([[-0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
          2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
          4.17e-06,  6.26e-01]])
```

13.4.4 Caixas Delimitadoras para Previsão

Durante a fase de previsão do modelo, primeiro geramos várias caixas de âncora para a imagem e, em seguida, predizemos categorias e deslocamentos para essas caixas de âncora, uma por uma. Em seguida, obtemos caixas delimitadoras de previsão com base nas caixas de âncora e seus deslocamentos previstos.

Abaixo, implementamos a função `offset_inverse` que leva âncoras e previsões de deslocamento como entradas e aplica transformações de deslocamento inversas para retornar as coordenadas da caixa delimitadora prevista.

```
#@save
def offset_inverse(anchors, offset_preds):
    c_anc = d2l.box_corner_to_center(anchors)
    c_pred_bb_xy = (offset_preds[:, :2] * c_anc[:, 2:] / 10) + c_anc[:, :2]
    c_pred_bb_wh = torch.exp(offset_preds[:, 2:] / 5) * c_anc[:, 2:]
    c_pred_bb = torch.cat((c_pred_bb_xy, c_pred_bb_wh), axis=1)
    predicted_bb = d2l.box_center_to_corner(c_pred_bb)
    return predicted_bb
```

Quando há muitas caixas de âncora, muitas caixas delimitadoras de predição semelhantes podem ser geradas para o mesmo alvo. Para simplificar os resultados, podemos remover caixas delimitadoras de predição semelhantes. Um método comumente usado é chamado de supressão não máxima (NMS).

Vamos dar uma olhada em como o NMS funciona. Para uma caixa delimitadora de previsão B, o modelo calcula a probabilidade prevista para cada categoria. Suponha que a maior probabilidade prevista seja p, a categoria correspondente a essa probabilidade é a categoria prevista de B. Também nos referimos a p como o nível de confiança da caixa delimitadora de predição B. Na mesma imagem, classificamos as caixas delimitadoras de previsão com categorias previstas diferentes do plano de fundo por nível de confiança de alto a baixo e obtemos a lista L. Selecionamos a caixa delimitadora de predição B_1 com o nível de confiança mais alto de L como linha de base e remova todas as caixas delimitadoras de predição não comparativas com um IoU com B_1 maior que um determinado limite de L. O limite aqui é um hiperparâmetro predefinido. Nesse ponto, L retém a caixa delimitadora de predição com o nível de confiança mais alto e remove outras caixas delimitadoras de predição semelhantes a ela. Em seguida, selecionamos a caixa delimitadora de predição B_2 com o segundo nível de confiança mais alto de L como linha de base e removemos todas as caixas delimitadoras de predição não comparativas com um IoU com B_2 maior que um determinado limite de L. Repetimos esse processo até que todas as caixas delimitadoras de previsão em L tenham sido usadas como linha de base. Neste momento, a IoU de qualquer par de caixas delimitadoras de predição em L é menor que o limite. Finalmente, produzimos todas as caixas delimitadoras de predição na lista L.

```
#@save
def nms(boxes, scores, iou_threshold):
```

(continues on next page)

(continued from previous page)

```
    # sorting scores by the descending order and return their indices
    B = torch.argsort(scores, dim=-1, descending=True)
    keep = []  # boxes indices that will be kept
    while B.numel() > 0:
        i = B[0]
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),
                      boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return torch.tensor(keep, device=boxes.device)

#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                       pos_threshold=0.00999999978):
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = torch.max(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, 0.5)
        # Find all non_keep indices and set the class_id to background
        all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
        combined = torch.cat((keep, all_idx))
        uniques, counts = combined.unique(return_counts=True)
        non_keep = uniques[counts == 1]
        all_id_sorted = torch.cat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted]
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        # threshold to be a positive prediction
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = torch.cat((class_id.unsqueeze(1),
                               conf.unsqueeze(1),
                               predicted_bb), dim=1)
        out.append(pred_info)
    return torch.stack(out)
```

A seguir, veremos um exemplo detalhado. Primeiro, construa quatro caixas de âncora. Para simplificar, assumimos que os deslocamentos previstos são todos 0. Isso significa que as caixas delimitadoras de previsão são caixas de âncora. Finalmente, construímos uma probabilidade prevista para cada categoria.

```
anchors = torch.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                      [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = torch.tensor([0] * anchors.numel())
cls_probs = torch.tensor([[0] * 4,  # Predicted probability for background
                        [0.9, 0.8, 0.7, 0.1],  # Predicted probability for dog
```

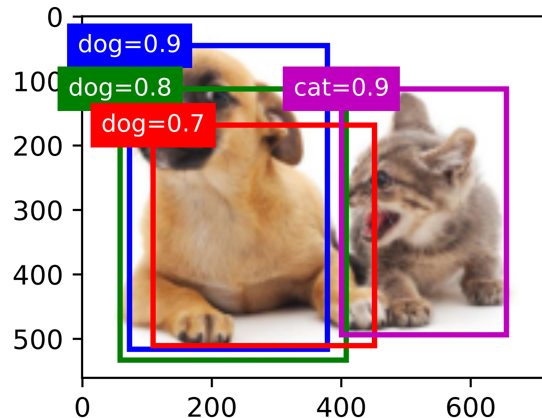
(continues on next page)

(continued from previous page)

```
                                    [0.1, 0.2, 0.3, 0.9]])  # Predicted probability for cat
```

Imprima caixas delimitadoras de previsão e seus níveis de confiança na imagem.

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



Usamos a função `multibox_detection` para executar NMS e definir o limite para 0,5. Isso adiciona uma dimensão de exemplo à entrada do tensor. Podemos ver que a forma do resultado retornado é (tamanho do lote, número de caixas de âncora, 6). Os 6 elementos de cada linha representam as informações de saída para a mesma caixa delimitadora de previsão. O primeiro elemento é o índice de categoria previsto, que começa em 0 (0 é cachorro, 1 é gato). O valor -1 indica fundo ou remoção no NMS. O segundo elemento é o nível de confiança da caixa delimitadora de previsão. Os quatro elementos restantes são as coordenadas do eixo x, y do canto superior esquerdo e as coordenadas do eixo x, y do canto inferior direito da caixa delimitadora de previsão (o intervalo de valores está entre 0 e 1).

```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0),
                            nms_threshold=0.5)
output
```

```
tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
         [ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
         [-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
         [-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

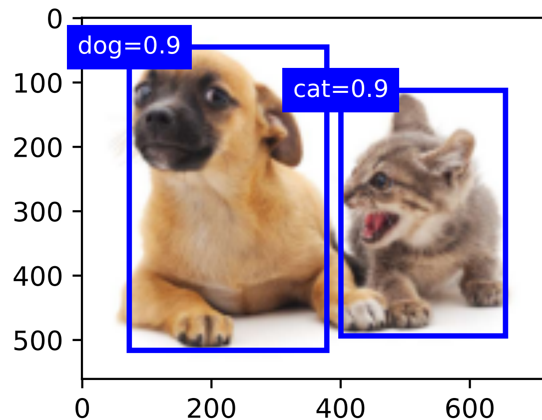
Removemos as caixas delimitadoras de predição da categoria -1 e visualizamos os resultados retidos pelo NMS.

```
fig = d2l.plt.imshow(img)
for i in output[0].detach().numpy():
    if i[0] == -1:
```

(continues on next page)

(continued from previous page)

```
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)
```



Na prática, podemos remover caixas delimitadoras de predição com níveis de confiança mais baixos antes de executar NMS, reduzindo assim a quantidade de computação para NMS. Também podemos filtrar a saída de NMS, por exemplo, retendo apenas os resultados com níveis de confiança mais altos como saída final.

13.4.5 Resumo

- Geramos várias caixas de âncora com diferentes tamanhos e proporções de aspecto, centralizadas em cada pixel.
- IoU, também chamado de índice de Jaccard, mede a similaridade de duas caixas delimitadoras. É a proporção entre a área de intersecção e a área de união de duas caixas delimitadoras.
- No conjunto de treinamento, marcamos dois tipos de rótulos para cada caixa de âncora: um é a categoria do alvo contido na caixa de âncora e o outro é o deslocamento da caixa delimitadora de verdade em relação à caixa de âncora.
- Ao prever, podemos usar supressão não máxima (NMS) para remover caixas delimitadoras de previsão semelhantes, simplificando assim os resultados.

13.4.6 Exercícios

1. Altere os valores de `sizes` e`ratios` na função `multibox_prior` e observe as alterações nas caixas de âncora geradas.
2. Construa duas caixas delimitadoras com uma IoU de 0,5 e observe sua coincidência.
3. Verifique a saída de offset `labels[0]` marcando os offsets da caixa de âncora conforme definido nesta seção (a constante é o valor padrão).
4. Modifique a variável `anchors` nas seções" Rotulando Caixas de Âncora de Conjunto de Treinamento "e" Caixas Limitadoras de Saída para Previsão ". Como os resultados mudam?

Discussões[159]

13.5 Detecção de Objetos Multiescala

Em Section 13.4, geramos várias caixas de âncora centralizadas em cada pixel da imagem de entrada. Essas caixas de âncora são usadas para amostrar diferentes regiões da imagem de entrada. No entanto, se as caixas de âncora forem geradas centralizadas em cada pixel da imagem, logo haverá muitas caixas de âncora para calcularmos. Por exemplo, assumimos que a imagem de entrada tem uma altura e uma largura de 561 e 728 pixels, respectivamente. Se cinco formas diferentes de caixas de âncora são geradas centralizadas em cada pixel, mais de dois milhões de caixas de âncora ($561 \times 728 \times 5$) precisam ser previstas e rotuladas na imagem.

Não é difícil reduzir o número de caixas de âncora. Uma maneira fácil é aplicar amostragem uniforme em uma pequena parte dos pixels da imagem de entrada e gerar caixas de âncora centralizadas nos pixels amostrados. Além disso, podemos gerar caixas de âncora de números e tamanhos variados em várias escalas. Observe que é mais provável que objetos menores sejam posicionados na imagem do que objetos maiores. Aqui, usaremos um exemplo simples: Objetos com formas de 1×1, 1×2, e 2×2 podem ter 4, 2 e 1 posição(ões) possível(is) em uma imagem com a forma 2×2.. Portanto, ao usar caixas de âncora menores para detectar objetos menores, podemos amostrar mais regiões; ao usar caixas de âncora maiores para detectar objetos maiores, podemos amostrar menos regiões.

Para demonstrar como gerar caixas de âncora em várias escalas, vamos ler uma imagem primeiro. Tem uma altura e largura de 561×728 pixels.

```
%matplotlib inline
import torch
from d2l import torch as d2l

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[0:2]
h, w
```

```
(561, 728)
```

Em Section 6.2, a saída da matriz 2D da rede neural convolucional (CNN) é chamada um mapa de recursos. Podemos determinar os pontos médios de caixas de âncora uniformemente amostradas em qualquer imagem, definindo a forma do mapa de feições.

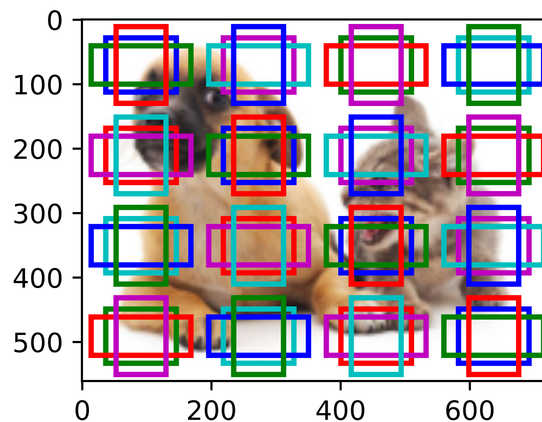
A função `display_anchors` é definida abaixo. Vamos gerar caixas de âncora `anchors` centradas em cada unidade (pixel) no mapa de feições `fmap`. Uma vez que as coordenadas dos eixos x e y nas caixas de âncora `anchors` foram divididas pela largura e altura do mapa de feições `fmap`, valores entre 0 e 1 podem ser usados para representar as posições relativas das caixas de âncora em o mapa de recursos. Uma vez que os pontos médios das "âncoras" das caixas de âncora se sobrepõem a todas as unidades no mapa de características "fmap", as posições espaciais relativas dos pontos médios das "âncoras" em qualquer imagem devem ter uma distribuição uniforme. Especificamente, quando a largura e a altura do mapa de feições são definidas para `fmap_w` e`fmap_h` respectivamente, a função irá conduzir uma amostragem uniforme para linhas `fmap_h` e colunas de pixels`fmap_w` e usá-los como pontos médios para gerar caixas de âncora com tamanho `s` (assumimos que o comprimento da lista `s` é 1) e diferentes proporções (`ratios`).

[159] https://discuss.d2l.ai/t/1603

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # The values from the first two dimensions will not affect the output
    fmap = torch.zeros((1, 10, fmap_h, fmap_w))
    anchors = d2l.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = torch.tensor((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img).axes,
                    anchors[0] * bbox_scale)
```

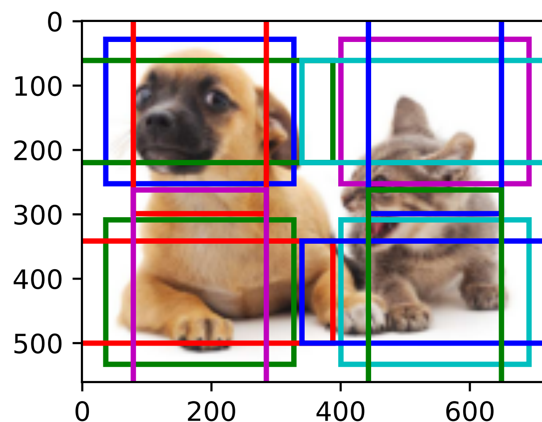
Primeiro, vamos nos concentrar na detecção de pequenos objetos. A fim de tornar mais fácil distinguir na exibição, as caixas de âncora com diferentes pontos médios aqui não se sobrepõem. Assumimos que o tamanho das caixas de âncora é 0,15 e a altura e largura do mapa de feições são 4. Podemos ver que os pontos médios das caixas de âncora das 4 linhas e 4 colunas da imagem estão uniformemente distribuídos.

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```



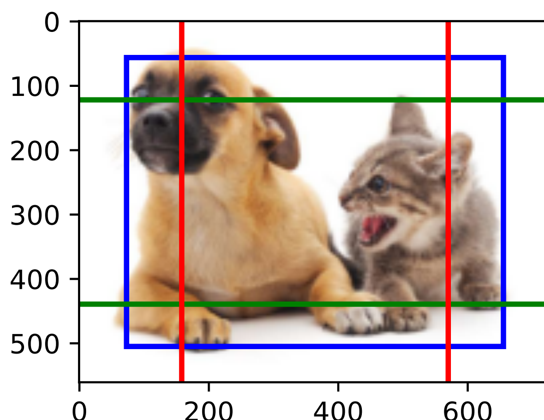
Vamos reduzir a altura e a largura do mapa de feições pela metade e usar uma caixa de âncora maior para detectar objetos maiores. Quando o tamanho é definido como 0,4, ocorrerão sobreposições entre as regiões de algumas caixas de âncora.

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



Finalmente, vamos reduzir a altura e a largura do mapa de feições pela metade e aumentar o tamanho da caixa de âncora para 0,8. Agora, o ponto médio da caixa de âncora é o centro da imagem.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



Como geramos caixas de âncora de tamanhos diferentes em escalas múltiplas, vamos usá-las para detectar objetos de vários tamanhos em escalas diferentes. Agora vamos apresentar um método baseado em redes neurais convolucionais (CNNs).

Em uma determinada escala, suponha que geramos $h \times w$ conjuntos de caixas de âncora com diferentes pontos médios baseados em c_i mapas de feições com a forma $h \times w$ e o número de caixas de âncora em cada conjunto é a . Por exemplo, para a primeira escala do experimento, geramos 16 conjuntos de caixas de âncora com diferentes pontos médios com base em 10 (número de canais) mapas de recursos com uma forma de 4×4, e cada conjunto contém 3 caixas de âncora. A seguir, cada caixa de âncora é rotulada com uma categoria e deslocamento com base na classificação e posição da caixa delimitadora de verdade. Na escala atual, o modelo de detecção de objeto precisa prever a categoria e o offset de $h \times w$ conjuntos de caixas de âncora com diferentes pontos médios com base na imagem de entrada.

Assumimos que os mapas de características c_i são a saída intermediária da CNN com base na imagem de entrada. Uma vez que cada mapa de características tem $h \times w$ posições espaciais diferentes, a mesma posição terá c_i unidades. De acordo com a definição de campo receptivo em Section 6.2, as unidades c_i do mapa de feições na mesma posição espacial têm o mesmo campo receptivo na imagem de entrada. Assim, elas representam a informação da imagem de entrada neste mesmo campo receptivo. Portanto, podemos transformar as unidades c_i do mapa de feições na mesma posição espacial nas categorias e deslocamentos das a caixas de âncora geradas usando essa posição como um ponto médio. Não é difícil perceber que, em essência, usamos as informações da imagem de entrada em um determinado campo receptivo para prever a categoria e deslocamento das caixas de âncora perto do campo na imagem de entrada.

Quando os mapas de recursos de camadas diferentes têm campos receptivos de tamanhos diferentes na imagem de entrada, eles são usados para detectar objetos de tamanhos diferentes. Por exemplo, podemos projetar uma rede para ter um campo receptivo mais amplo para cada unidade no mapa de recursos que está mais perto da camada de saída, para detectar objetos com tamanhos maiores na imagem de entrada.

Implementaremos um modelo de detecção de objetos multiescala na seção seguinte.

13.5.1 Resumo

- Podemos gerar caixas de âncora com diferentes números e tamanhos em várias escalas para detectar objetos de diferentes tamanhos em várias escalas.
- A forma do mapa de feições pode ser usada para determinar o ponto médio das caixas de âncora que amostram uniformemente qualquer imagem.
- Usamos as informações da imagem de entrada de um determinado campo receptivo para prever a categoria e o deslocamento das caixas de âncora próximas a esse campo na imagem.

13.5.2 Exercícios

1. Dada uma imagem de entrada, suponha que $1 \times c_i \times h \times w$ seja a forma do mapa de características, enquanto c_i, h, w são o número, altura e largura do mapa de características. Em quais métodos você pode pensar para converter essa variável na categoria e deslocamento da caixa de âncora? Qual é o formato da saída?

Discussões[160]

13.6 O *Dataset* de Detecção de Objetos

Não existem pequenos conjuntos de dados, como MNIST ou Fashion-MNIST, no campo de detecção de objetos. Para testar os modelos rapidamente, vamos montar um pequeno conjunto de dados. Primeiro, geramos 1000 imagens de banana de diferentes ângulos e tamanhos usando bananas grátis de nosso escritório. Em seguida, coletamos uma série de imagens de fundo e colocamos uma imagem de banana em uma posição aleatória em cada imagem.

13.6.1 Baixando Dataset

O conjunto de dados de detecção de banana com todas as imagens e arquivos de rótulo csv pode ser baixado diretamente da Internet.

```
%matplotlib inline
import os
import pandas as pd
import torch
import torchvision
from d2l import torch as d2l

#@save
d2l.DATA_HUB['banana-detection'] = (d2l.DATA_URL + 'banana-detection.zip',
                           '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

[160] https://discuss.d2l.ai/t/1607

13.6.2 Lendo o Dataset

Vamos ler o conjunto de dados de detecção de objetos na função `read_data_bananas`. O conjunto de dados inclui um arquivo csv para rótulos de classe de destino e coordenadas de caixa delimitadora de verdade fundamental no formato `corner`. Definimos `BananasDataset` para criar a instância do `Dataset` e finalmente definimos a função `load_data_bananas` para retornar os carregadores de dados. Não há necessidade de ler o conjunto de dados de teste em ordem aleatória.

```
#@save
def read_data_bananas(is_train=True):
    """Read the bananas dataset images and labels."""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(torchvision.io.read_image(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # Since all images have same object class i.e. category '0',
        # the `label` column corresponds to the only object i.e. banana
        # The target is as follows : (`label`, `xmin`, `ymin`, `xmax`, `ymax`)
        targets.append(list(target))
    return images, torch.tensor(targets).unsqueeze(1) / 256


#@save
class BananasDataset(torch.utils.data.Dataset):
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
              is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])

    def __len__(self):
        return len(self.features)


#@save
def load_data_bananas(batch_size):
    """Load the bananas dataset."""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                             batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return (train_iter, val_iter)
```

Abaixo, lemos um minibatch e imprimimos o formato da imagem e do *label*. A forma da imagem é a mesma da experiência anterior (tamanho do lote, número de canais, altura, largura). O formato do rótulo é (tamanho do lote, m, 5), onde m é igual ao número máximo de caixas delimitadoras contidas em uma única imagem no conjunto de dados. Embora o cálculo do minibatch seja muito eficiente, ele exige que cada imagem contenha o mesmo número de caixas delimitadoras para

que possam ser colocadas no mesmo lote. Como cada imagem pode ter um número diferente de caixas delimitadoras, podemos adicionar caixas delimitadoras ilegais às imagens que possuem menos de m caixas delimitadoras até que cada imagem contenha m caixas delimitadoras. Assim, podemos ler um minibatch de imagens a cada vez. O rótulo de cada caixa delimitadora na imagem é representado por um tensor de comprimento 5. O primeiro elemento no tensor é a categoria do objeto contido na caixa delimitadora. Quando o valor é -1, a caixa delimitadora é uma caixa delimitadora ilegal para fins de preenchimento. Os quatro elementos restantes da matriz representam as coordenadas do eixo x, y do canto superior esquerdo da caixa delimitadora e as coordenadas do eixo x, y do canto inferior direito da caixa delimitadora (o intervalo de valores é entre 0 e 1). O conjunto de dados da banana aqui tem apenas uma caixa delimitadora por imagem, então $m = 1$.

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

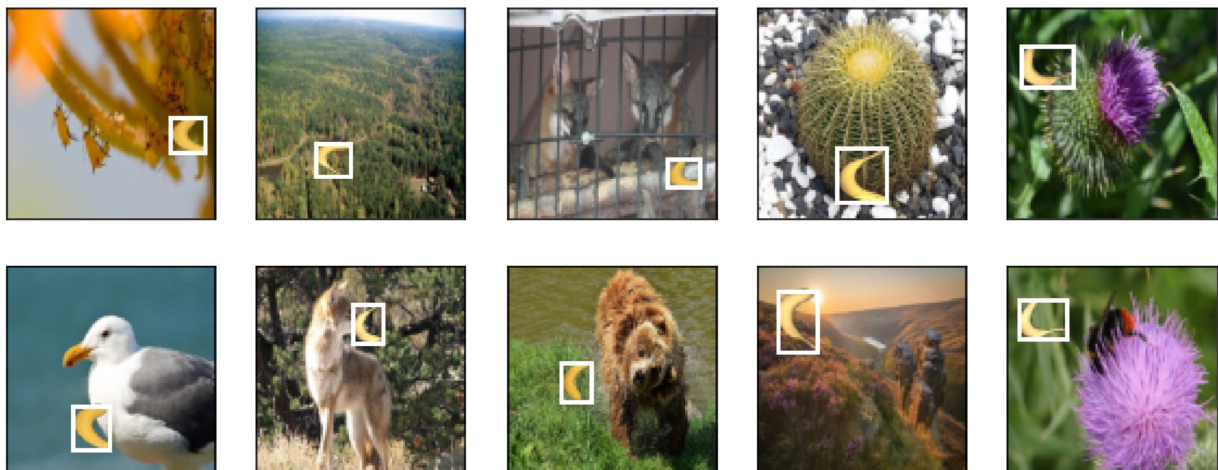
```
Downloading ../data/banana-detection.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪banana-detection.zip...
read 1000 training examples
read 100 validation examples
```

```
(torch.Size([32, 3, 256, 256]), torch.Size([32, 1, 5]))
```

13.6.3 Demonstração

Temos dez imagens com caixas delimitadoras. Podemos ver que o ângulo, o tamanho e a posição da banana são diferentes em cada imagem. Claro, este é um conjunto de dados artificial simples. Na prática, os dados geralmente são muito mais complicados.

```
imgs = (batch[0][0:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch[1][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



13.6.4 Resumo

- O conjunto de dados de detecção de banana que sintetizamos pode ser usado para testar modelos de detecção de objetos.
- A leitura de dados para detecção de objetos é semelhante àquela para classificação de imagens. No entanto, depois de introduzirmos as caixas delimitadoras, a forma do rótulo e o aumento da imagem (por exemplo, corte aleatório) são alterados.

13.6.5 Exercícios

1. Referindo-se à documentação do MXNet, quais são os parâmetros para os construtores das classes `image.ImageDetIter` e `image.CreateDetAugmenter`? Qual é o seu significado?

Discussões[161]

13.7 Detecção *Single Shot Multibox* (SSD)

Nas poucas seções anteriores, apresentamos caixas delimitadoras, caixas de âncora, detecção de objetos multiescala e conjuntos de dados. Agora, usaremos esse conhecimento prévio para construir um modelo de detecção de objetos: detecção multibox de disparo único [*Single Shot Multibox Detection*] (SSD) (Liu et al., 2016). Este modelo rápido e fácil já é amplamente utilizado. Alguns dos conceitos de design e detalhes de implementação deste modelo também são aplicáveis a outros modelos de detecção de objetos.

13.7.1 Modelo

Fig. 13.7.1 mostra o design de um modelo SSD. Os principais componentes do modelo são um bloco de rede básico e vários blocos de recursos multiescala conectados em série. Aqui, o bloco de rede de base é usado para as características extras de imagens originais e geralmente assumem a forma de uma rede neural convolucional profunda. O artigo sobre SSDs opta por colocar um VGG truncado antes do camada de classificação (Liu et al., 2016), mas agora é comumente substituído pelo ResNet. Podemos projetar uma rede de base para que ela produza alturas e larguras maiores. Desta forma, mais caixas de âncora são geradas com base neste mapa de características, permitindo-nos detectar objetos menores. Em seguida, cada bloco de feições multiescala reduz a altura e largura do mapa de feições fornecidas pela camada anterior (por exemplo, pode reduzir os tamanhos pela metade). Os blocos então usam cada elemento no mapa de recursos para expandir o campo receptivo na imagem de entrada. Desta forma, quanto mais próximo um bloco de feições multiescala estiver do topo de Fig. 13.7.1 menor será o mapa de feições de saída e menos caixas de âncora são geradas com base no mapa de feições. Além disso, quanto mais próximo um bloco de recursos estiver do topo, maior será o campo receptivo de cada elemento no mapa de recursos e mais adequado será para detectar objetos maiores. Como o SSD gera diferentes números de caixas de âncora de tamanhos diferentes com base no bloco de rede de base e cada bloco de recursos multiescala e, em seguida, prevê como categorias e deslocamentos (ou seja, caixas delimitadoras previsão) das caixas de âncora para detectar objetos de tamanhos diferentes, SSD é um modelo de detecção de objetos multiescala.

[161] https://discuss.d2l.ai/t/1608

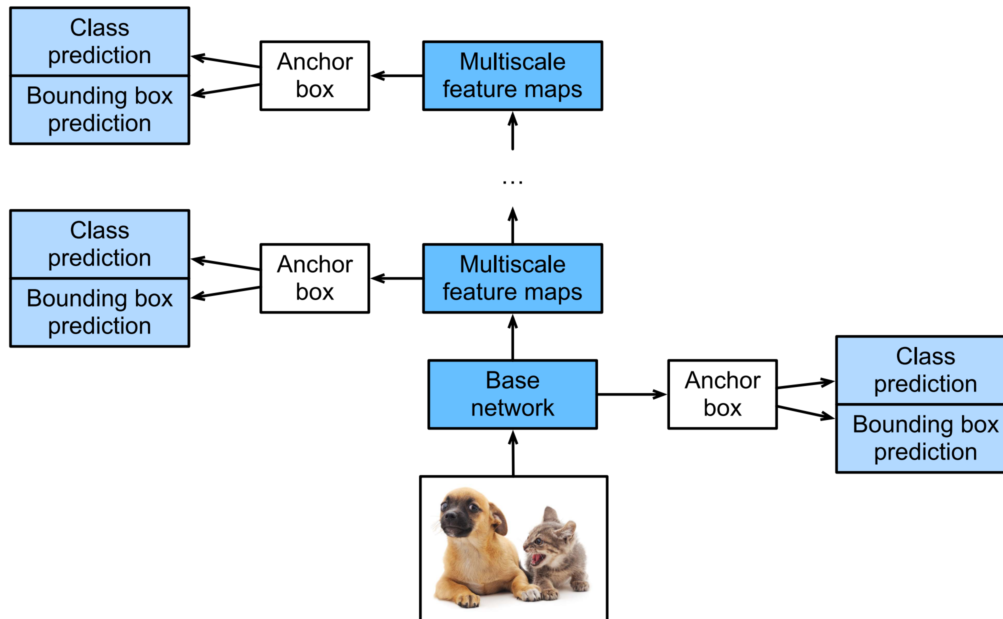


Fig. 13.7.1: O SSD é composto de um bloco de rede base e vários blocos de recursos multiescala conectados em série.

A seguir, descreveremos a implementação dos módulos em Fig. 13.7.1. Primeiro, precisamos discutir a implementação da previsão da categoria e da previsão da caixa delimitadora.

Camada de Previsão da Categoria

Defina o número de categorias de objeto como q. Nesse caso, o número de categorias de caixa de âncora é $q + 1$, com 0 indicando uma caixa de âncora que contém apenas o fundo. Para uma determinada escala, defina a altura e a largura do mapa de feições para h e w, respectivamente. Se usarmos cada elemento como o centro para gerar a caixas de âncora, precisamos classificar um total de hwa caixas de âncora. Se usarmos uma camada totalmente conectada (FCN) para a saída, isso provavelmente resultará em um número excessivo de parâmetros do modelo. Lembre-se de como usamos canais de camada convolucional para gerar previsões de categoria em Section 7.3. O SSD usa o mesmo método para reduzir a complexidade do modelo.

Especificamente, a camada de predição de categoria usa uma camada convolucional que mantém a altura e largura de entrada. Assim, a saída e a entrada têm uma correspondência de um para um com as coordenadas espaciais ao longo da largura e altura do mapa de características. Supondo que a saída e a entrada tenham as mesmas coordenadas (x, y), o canal para as coordenadas (x, y) no mapa de feição de saída contém as previsões de categoria para todas as caixas âncora geradas usando as coordenadas do mapa de feição de entrada (x, y) como o Centro. Portanto, existem $a(q+1)$ canais de saída, com os canais de saída indexados como $i(q + 1) + j$ ($0 \leq j \leq q$) representando as previsões do índice de categoria j para o índice de caixa de âncora i.

Agora, vamos definir uma camada de predição de categoria deste tipo. Depois de especificar os parâmetros a e q, ele usa uma camada convolucional 3×3 com um preenchimento de 1. As alturas e larguras de entrada e saída dessa camada convolucional permanecem inalteradas.

```
%matplotlib inline
import torch
```

(continues on next page)

(continued from previous page)

```
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

Camada de Previsão de Caixa Delimitadora

O design da camada de previsão da caixa delimitadora é semelhante ao da camada de previsão da categoria. A única diferença é que, aqui, precisamos prever 4 deslocamentos para cada caixa de âncora, em vez de categorias $q + 1$.

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

Concatenando Previsões para Múltiplas Escalas

Como mencionamos, o SSD usa mapas de recursos com base em várias escalas para gerar caixas de âncora e prever suas categorias e deslocamentos. Como as formas e o número de caixas de âncora centradas no mesmo elemento diferem para os mapas de recursos de escalas diferentes, as saídas de predição em escalas diferentes podem ter formas diferentes.

No exemplo a seguir, usamos o mesmo lote de dados para construir mapas de características de duas escalas diferentes, `Y1` e `Y2`. Aqui, `Y2` tem metade da altura e metade da largura de `Y1`. Usando a previsão de categoria como exemplo, assumimos que cada elemento nos mapas de características `Y1` e`Y2` gera cinco (Y1) ou três (Y2) caixas de âncora. Quando há 10 categorias de objeto, o número de canais de saída de predição de categoria é $5 \times (10 + 1) = 55$ ou $3 \times (10 + 1) = 33$. O formato da saída de previsão é (tamanho do lote, número de canais, altura, largura). Como você pode ver, exceto pelo tamanho do lote, os tamanhos das outras dimensões são diferentes. Portanto, devemos transformá-los em um formato consistente e concatenar as previsões das várias escalas para facilitar o cálculo subsequente.

```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
(Y1.shape, Y2.shape)
```

```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

A dimensão do canal contém as previsões para todas as caixas de âncora com o mesmo centro. Primeiro movemos a dimensão do canal para a dimensão final. Como o tamanho do lote é o mesmo para todas as escalas, podemos converter os resultados da previsão para o formato binário

(tamanho do lote, altura $\times$ largura $\times$ número de canais) para facilitar a concatenação subsequente no 1st dimensão.

```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

Assim, independentemente das diferentes formas de `Y1` e`Y2`, ainda podemos concatenar os resultados da previsão para as duas escalas diferentes do mesmo lote.

```
concat_preds([Y1, Y2]).shape
```

```
torch.Size([2, 25300])
```

Bloco de Redução de Amostragem de Altura e Largura

Para detecção de objetos multiescala, definimos o seguinte bloco `down_sample_blk`, que reduz a altura e largura em 50%. Este bloco consiste em duas camadas convolucionais 3×3 com um preenchimento de 1 e uma camada de *pooling* máximo 2×2 com uma distância de 2 conectadas em uma série. Como sabemos, 3×3 camadas convolucionais com um preenchimento de 1 não alteram a forma dos mapas de características. No entanto, a camada de agrupamento subsequente reduz diretamente o tamanho do mapa de feições pela metade. Como $1 \times 2 + (3-1) + (3-1) = 6$, cada elemento no mapa de recursos de saída tem um campo receptivo no mapa de recursos de entrada da forma 6×6. Como você pode ver, o bloco de redução de altura e largura aumenta o campo receptivo de cada elemento no mapa de recursos de saída.

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                             kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

Ao testar a computação direta no bloco de redução de altura e largura, podemos ver que ele altera o número de canais de entrada e divide a altura e a largura pela metade.

```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

```
torch.Size([2, 10, 10, 10])
```

Bloco de Rede Base

O bloco de rede básico é usado para extrair recursos das imagens originais. Para simplificar o cálculo, construiremos uma pequena rede de base. Essa rede consiste em três blocos de *downsample* de altura e largura conectados em série, portanto, dobra o número de canais em cada etapa. Quando inserimos uma imagem original com a forma 256×256, o bloco de rede base produz um mapa de características com a forma 32×32.

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

```
torch.Size([2, 64, 32, 32])
```

O Modelo Completo

O modelo SSD contém um total de cinco módulos. Cada módulo produz um mapa de recursos usado para gerar caixas de âncora e prever as categorias e deslocamentos dessas caixas de âncora. O primeiro módulo é o bloco de rede base, os módulos de dois a quatro são blocos de redução de amostragem de altura e largura e o quinto módulo é um bloco global camada de pooling máxima que reduz a altura e largura para 1. Portanto, os módulos dois a cinco são todos blocos de recursos multiescala mostrados em Fig. 13.7.1.

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

Agora, vamos definir o processo de computação progressiva para cada módulo. Em contraste com as redes neurais convolucionais descritas anteriormente, este módulo não só retorna a saída do mapa de características `Y` por computação convolucional, mas também as caixas de âncora da escala atual gerada a partir de`Y` e suas categorias e deslocamentos previstos.

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

Como mencionamos, quanto mais próximo um bloco de recursos multiescala está do topo em Fig. 13.7.1, maiores são os objetos que ele detecta e maiores são as caixas de âncora que deve gerar. Aqui, primeiro dividimos o intervalo de 0,2 a 1,05 em cinco partes iguais para determinar os tamanhos das caixas de âncora menores em escalas diferentes: 0,2, 0,37, 0,54, etc. Então, de acordo com $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$, e fórmulas semelhantes, determinamos os tamanhos de caixas de âncora maiores em escalas diferentes.

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

Agora, podemos definir o modelo completo, `TinySSD`.

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            # The assignment statement is self.blk_i = get_blk(i)
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                                    num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                                      num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # getattr(self, 'blk_%d' % i) accesses self.blk_i
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        # In the reshape function, 0 indicates that the batch size remains
        # unchanged
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

Agora criamos uma instância de modelo SSD e a usamos para realizar cálculos avançados no minibatch de imagem `X`, que tem uma altura e largura de 256 pixels. Como verificamos anteriormente, o primeiro módulo gera um mapa de recursos com a forma 32×32. Como os módulos dois a quatro são blocos de redução de altura e largura, o módulo cinco é uma camada de agrupamento global e cada elemento no mapa de recursos é usado como o centro para 4 caixas de âncora, um total de $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ caixas de âncora são geradas para cada imagem nas cinco escalas.

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)
```

(continues on next page)

(continued from previous page)

```
print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
```

13.7.2 Treinamento

Agora, vamos explicar, passo a passo, como treinar o modelo SSD para detecção de objetos.

Leitura e Inicialização de Dados

Lemos o conjunto de dados de detecção de banana que criamos na seção anterior.

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

Existe 1 categoria no conjunto de dados de detecção de banana. Depois de definir o módulo, precisamos inicializar os parâmetros do modelo e definir o algoritmo de otimização.

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

Definindo Funções de Perda e Avaliação

A detecção de objetos está sujeita a dois tipos de perdas. a primeira é a perda da categoria da caixa de âncora. Para isso, podemos simplesmente reutilizar a função de perda de entropia cruzada que usamos na classificação de imagens. A segunda perda é a perda de deslocamento da caixa de âncora positiva. A previsão de deslocamento é um problema de normalização. No entanto, aqui, não usamos a perda quadrática introduzida anteriormente. Em vez disso, usamos a perda de norma L_1, que é o valor absoluto da diferença entre o valor previsto e o valor verdadeiro. A variável de máscara `bbox_masks` remove caixas de âncora negativas e caixas de âncora de preenchimento do cálculo de perda. Finalmente, adicionamos a categoria de caixa de âncora e compensamos as perdas para encontrar a função de perda final para o modelo.

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
```

(continues on next page)

(continued from previous page)

```
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

Podemos usar a taxa de precisão para avaliar os resultados da classificação. Como usamos a perda de norma L_1, usaremos o erro absoluto médio para avaliar os resultados da previsão da caixa delimitadora.

```
def cls_eval(cls_preds, cls_labels):
    # Because the category prediction results are placed in the final
    # dimension, argmax must specify this dimension
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

Treinando o Modelo

Durante o treinamento do modelo, devemos gerar caixas de âncora multiescala (`âncoras`) no processo de computação direta do modelo e prever a categoria (`cls_preds`) e o deslocamento (`bbox_preds`) para cada caixa de âncora. Depois, rotulamos a categoria (`cls_labels`) e o deslocamento (`bbox_labels`) de cada caixa de âncora gerada com base nas informações do rótulo `Y`. Finalmente, calculamos a função de perda usando a categoria predita e rotulada e os valores de compensação. Para simplificar o código, não avaliamos o conjunto de dados de treinamento aqui.

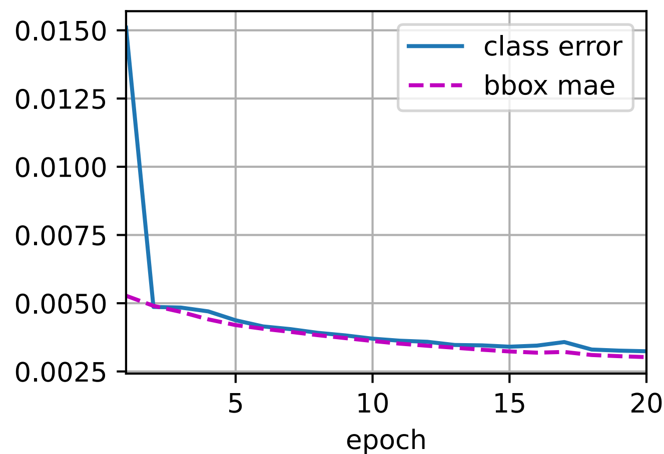
```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # accuracy_sum, mae_sum, num_examples, num_labels
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # Generate multiscale anchor boxes and predict the category and
        # offset of each
        anchors, cls_preds, bbox_preds = net(X)
        # Label the category and offset of each anchor box
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
        # Calculate the loss function using the predicted and labeled
        # category and offset values
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
```

(continues on next page)

(continued from previous page)

```
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.24e-03, bbox mae 3.03e-03
5130.7 examples/sec on cuda:0
```



13.7.3 Predição

Na fase de previsão, queremos detectar todos os objetos de interesse na imagem. Abaixo, lemos a imagem de teste e transformamos seu tamanho. Então, nós o convertemos para o formato quadridimensional exigido pela camada convolucional.

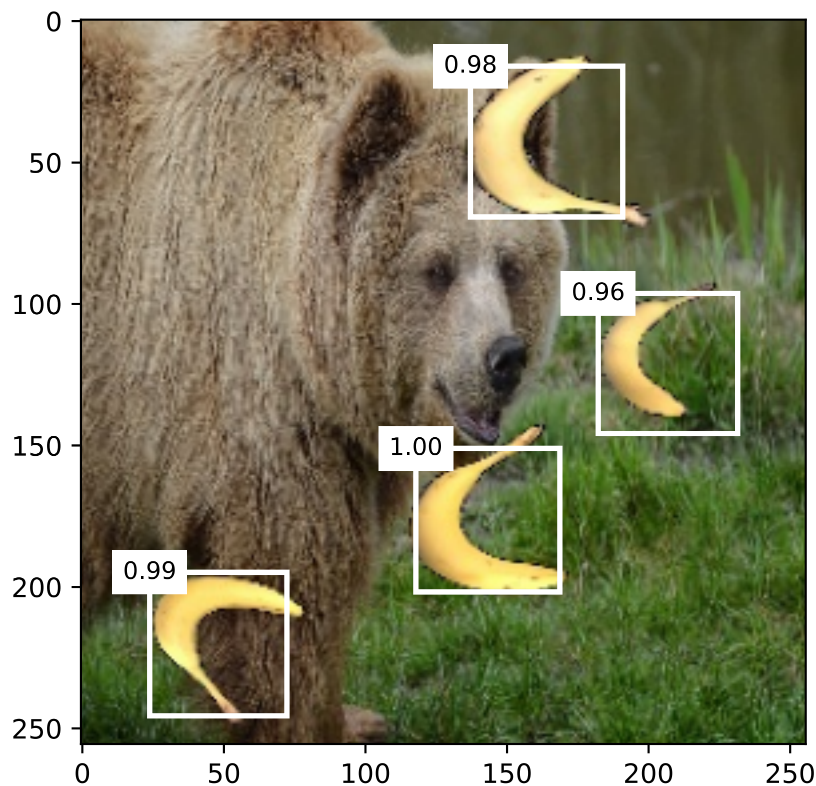
```
X = torchvision.io.read_image('../img/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1,2,0).long()
```

Usando a função `multibox_detection`, prevemos as caixas delimitadoras com base nas caixas de âncora e seus deslocamentos previstos. Em seguida, usamos a supressão não máxima para remover caixas delimitadoras semelhantes.

```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

Por fim, pegamos todas as caixas delimitadoras com um nível de confiança de pelo menos 0,9 e as exibimos como a saída final.

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```



13.7.4 Resumo

- SSD é um modelo de detecção de objetos multiescala. Este modelo gera diferentes números de caixas de âncora de tamanhos diferentes com base no bloco de rede de base e cada bloco de recursos multiescala e prevê as categorias e deslocamentos das caixas de âncora para detectar objetos de tamanhos diferentes.
- Durante o treinamento do modelo SSD, a função de perda é calculada usando a categoria prevista e rotulada e os valores de deslocamento.

13.7.5 Exercícios

1. Devido a limitações de espaço, ignoramos alguns dos detalhes de implementação do modelo SSD neste experimento. Você pode melhorar ainda mais o modelo nas seguintes áreas?

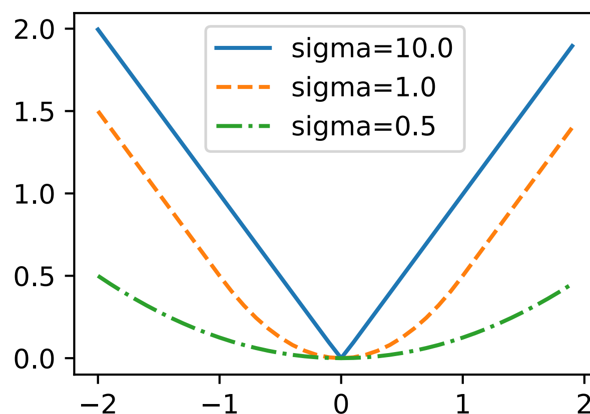
Função de Perda

A. Para as compensações previstas, substitua L_1 perda de norma por L_1 de perda de regularização. Esta função de perda usa uma função quadrada em torno de zero para maior suavidade. Esta é a área regularizada controlada pelo hiperparâmetro σ:

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if }|x| < 1/\sigma^2\\ |x|-0.5/\sigma^2, & \text{otherwise}\end{cases} \tag{13.7.1}$$

Quando σ é grande, essa perda é semelhante à perda normal de L_1. Quando o valor é pequeno, a função de perda é mais suave.

```
def smooth_l1(data, scalar):
    out = []
    for i in data:
        if abs(i) < 1 / (scalar ** 2):
            out.append(((scalar * i) ** 2) / 2)
        else:
            out.append(abs(i) - 0.5 / (scalar ** 2))
    return torch.tensor(out)

sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = torch.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = smooth_l1(x, scalar=s)
    d2l.plt.plot(x, y, l, label='sigma=%.1f' % s)
d2l.plt.legend();
```



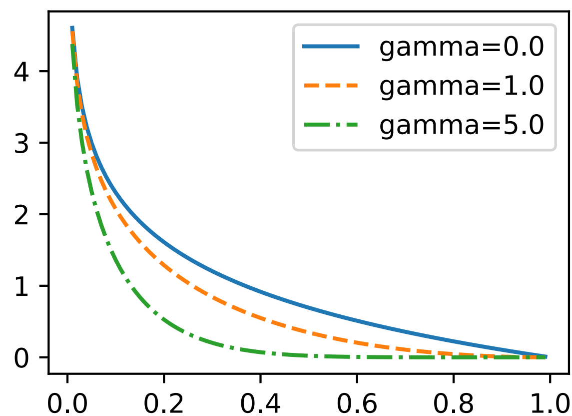
No experimento, usamos a perda de entropia cruzada para a previsão da categoria. Agora, assuma que a probabilidade de predição da categoria real j é p_j e a perda de entropia cruzada é $-\log p_j$.

Também podemos usar a perda focal (Lin et al., 2017a). Dados os hiperparâmetros positivos γ e α, essa perda é definida como:

$$-\alpha(1-p_j)^\gamma \log p_j. \tag{13.7.2}$$

Como você pode ver, ao aumentar γ, podemos efetivamente reduzir a perda quando a probabilidade de prever a categoria correta for alta.

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * torch.log(x)

x = torch.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x, focal_loss(gamma, x), l, label='gamma=%.1f' % gamma)
d2l.plt.legend();
```



Treinamento e Previsão

B. Quando um objeto é relativamente grande em comparação com a imagem, o modelo normalmente adota um tamanho de imagem de entrada maior.

C. Isso geralmente produz um grande número de caixas de âncora negativas ao rotular as categorias da caixa de âncora. Podemos amostrar as caixas de âncora negativas para equilibrar melhor as categorias de dados. Para fazer isso, podemos definir um parâmetro `negative_mining_ratio` na função `multibox_target`.

D. Atribuir hiperparâmetros com pesos diferentes para a perda de categoria da caixa de âncora e a perda de deslocamento da caixa de âncora positiva na função de perda.

E. Consulte o documento SSD. Quais métodos podem ser usados para avaliar a precisão dos modelos de detecção de objetos (Liu et al., 2016)?

Discussões[162]

[162] https://discuss.d2l.ai/t/1604

13.8 Region-based CNNs (R-CNNs)

Redes neurais convolucionais baseadas em regiões ou regiões com recursos CNN (R-CNNs) são uma abordagem pioneira que aplica modelos profundos para detecção de objetos (Girshick et al., 2014). Nesta seção, discutiremos R-CNNs e uma série de melhorias feitas a eles: Fast R-CNN (Girshick, 2015), Faster R-CNN (Ren et al., 2015) e Mask R-CNN (He et al., 2017). Devido às limitações de espaço, limitaremos nossa discussão aos designs desses modelos.

13.8.1 R-CNNs

Os modelos R-CNN primeiro selecionam várias regiões propostas de uma imagem (por exemplo, as caixas de âncora são um tipo de método de seleção) e, em seguida, rotulam suas categorias e caixas delimitadoras (por exemplo, deslocamentos). Em seguida, eles usam uma CNN para realizar cálculos avançados para extrair recursos de cada área proposta. Depois, usamos os recursos de cada região proposta para prever suas categorias e caixas delimitadoras. Fig. 13.8.1 mostra um modelo R-CNN.

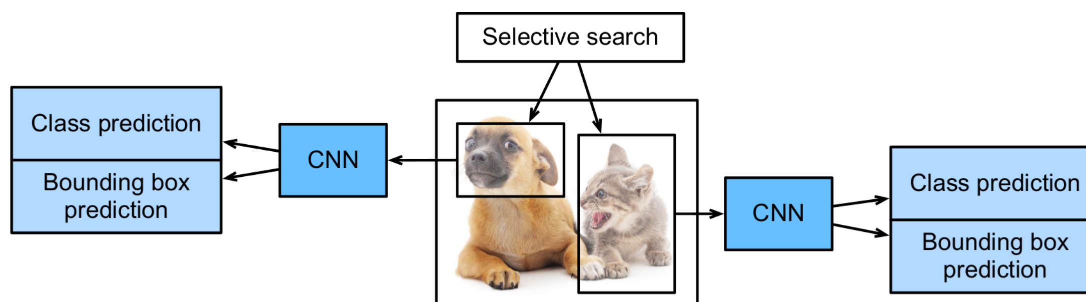


Fig. 13.8.1: Modelo R-CNN.

Especificamente, os R-CNNs são compostos por quatro partes principais:

1. A pesquisa seletiva é realizada na imagem de entrada para selecionar várias regiões propostas de alta qualidade (Uijlings et al., 2013). Essas regiões propostas são geralmente selecionados em várias escalas e têm diferentes formas e tamanhos. A categoria e a caixa delimitadora da verdade fundamental de cada região proposta é etiquetada.
2. Um CNN pré-treinado é selecionado e colocado, de forma truncada, antes da camada de saída. Ele transforma cada região proposta nas dimensões de entrada exigido pela rede e usa computação direta para gerar os recursos extraídos das regiões propostas.
3. Os recursos e a categoria rotulada de cada região proposta são combinados como um exemplo para treinar várias máquinas de vetores de suporte para classificação de objeto. Aqui, cada máquina de vetor de suporte é usada para determinar se um exemplo pertence a uma determinada categoria.
4. Os recursos e a caixa delimitadora rotulada de cada região proposta são combinados como um exemplo para treinar um modelo de regressão linear para a predição da caixa delimitadora de verdade básica.

Embora os modelos R-CNN usem CNNs pré-treinados para extrair recursos de imagem com eficácia, a principal desvantagem é a velocidade lenta. Como você pode imaginar, podemos selecionar

milhares de regiões propostas a partir de uma única imagem, exigindo milhares de cálculos diretos da CNN para realizar a detecção de objetos. Essa enorme carga de computação significa que os R-CNNs não são amplamente usados em aplicativos reais.

13.8.2 Fast R-CNN

O principal gargalo de desempenho de um modelo R-CNN é a necessidade de extrair recursos de forma independente para cada região proposta. Como essas regiões têm um alto grau de sobreposição, a extração de recursos independentes resulta em um alto volume de cálculos repetitivos. Fast R-CNN melhora o R-CNN por apenas executar CNN computação progressiva na imagem como um todo.

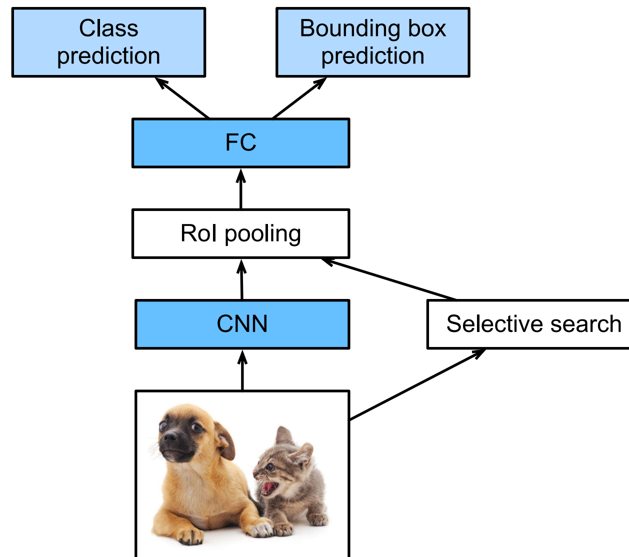


Fig. 13.8.2: Modelo Fast R-CNN.

Fig. 13.8.2 mostra um modelo Fast R-CNN. Suas principais etapas de computação são descritas abaixo:

1. Comparado a um modelo R-CNN, um modelo Fast R-CNN usa a imagem inteira como o Entrada da CNN para extração de recursos, em vez de cada região proposta. Além disso, esta rede é geralmente treinada para atualizar os parâmetros do modelo. Enquanto o a entrada é uma imagem inteira, a forma de saída do CNN é $1 \times c \times h_1 \times w_1$.
2. Supondo que a pesquisa seletiva gere n regiões propostas, seus diferentes formas indicam regiões de interesses (RoIs) de diferentes formas na CNN resultado. Características das mesmas formas devem ser extraídas dessas RoIs (aqui assumimos que a altura é h_2 e a largura é w_2). R-CNN rápido apresenta o pool de RoI, que usa a saída CNN e RoIs como entrada para saída uma concatenação dos recursos extraídos de cada região proposta com o forma $n \times c \times h_2 \times w_2$.
3. Uma camada totalmente conectada é usada para transformar a forma de saída em $n \times d$, onde d é determinado pelo design do modelo.
4. Durante a previsão da categoria, a forma da saída da camada totalmente conectada é novamente transformada em $n \times q$ e usamos a regressão softmax (q é o número de categorias). Durante a previsão da caixa delimitadora, a forma do a saída da camada conectada é nova-

mente transformada em $n \times 4$. Isso significa que prevemos a categoria e a caixa delimitadora para cada região proposta.

A camada de pooling de RoI no Fast R-CNN é um pouco diferente das camadas de pool que discutimos antes. Em uma camada de pooling normal, definimos a janela de pool, preenchimento e passo para controlar a forma de saída. Em uma camada de pooling de RoI, podemos especificar diretamente a forma de saída de cada região, como especificar a altura e a largura de cada região como h_2, w_2. Supondo que o altura e largura da janela RoI são h e w, esta janela é dividida em uma grade de subjanelas com a forma h_2 **:raw-latex:`\times `w_2$. O tamanho de cada subjanela é de cerca de :math:`(h/h_2) \times (w/w_2)`**. A altura e largura da subjanela devem ser sempre inteiros e o maior elemento é usado como saída para um determinada subjanela. Isso permite que a camada de pooling de RoI extraia recursos do mesmo formato de RoIs de formatos diferentes.

Em Fig. 13.8.3, selecionamos uma região 3×3 como um RoI da entrada 4×4. Para este RoI, usamos uma camada de pool de 2×2 RoI para obter uma única saída 2×2. Quando dividimos a região em quatro subjanelas, elas contêm respectivamente os elementos 0, 1, 4 e 5 (5 é o maior); 2 e 6 (6 é o maior); 8 e 9 (9 é o maior); e 10.

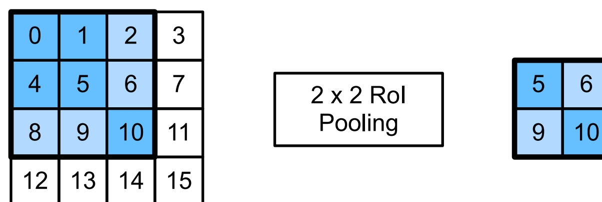


Fig. 13.8.3: Camada de *pooling* RoI 2×2 .

Usamos a função `roi_pool` de`torchvision` para demonstrar a computação da camada de pooling RoI. Suponha que a CNN extraia o elemento `X` com altura e largura 4 e apenas um único canal.

```
import torch
import torchvision

X = torch.arange(16.).reshape(1, 1, 4, 4)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Suponha que a altura e a largura da imagem sejam de 40 pixels e que a busca seletiva gere duas regiões propostas na imagem. Cada região é expressa como cinco elementos: a categoria de objeto da região e as coordenadas x, y de seus cantos superior esquerdo e inferior direito.

```
rois = torch.Tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Como a altura e largura de `X` são $1/10$ da altura e largura da imagem, as coordenadas das duas regiões propostas são multiplicadas por 0,1 de acordo com `escala_espacial`, e então os RoIs são rotulados em `X` como `X[:,:, 0: 3, 0: 3]` e `X[:,:, 1: 4, 0: 4]`, respectivamente. Por fim, dividimos os dois RoIs em uma grade de subjanela e extraímos recursos com altura e largura 2.

```
torchvision.ops.roi_pool(X, rois, output_size=(2, 2), spatial_scale=0.1)
```

```
tensor([[[[ 5.,  6.],
          [ 9., 10.]]],


        [[[ 9., 11.],
          [13., 15.]]]])
```

13.8.3 R-CNN Mais Rápido

A fim de obter resultados precisos de detecção de objeto, Fast R-CNN geralmente requer que muitas regiões propostas sejam geradas em busca seletiva. O Faster R-CNN substitui a pesquisa seletiva por uma rede de proposta regional. Isso reduz o número de regiões propostas geradas, garantindo a detecção precisa do objeto.

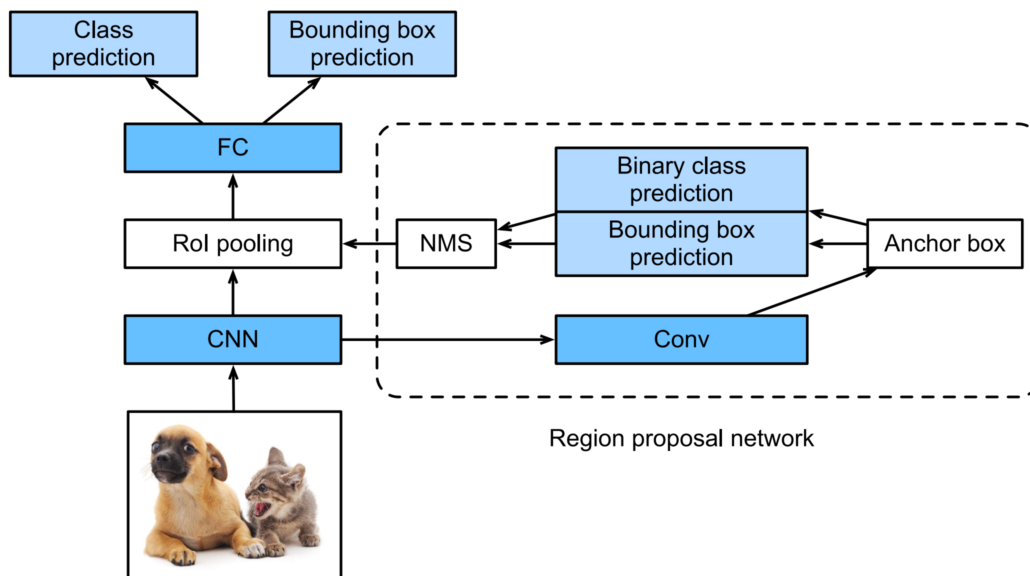


Fig. 13.8.4: Modelo R-CNN mais rápido.

Fig. 13.8.4 mostra um modelo Faster R-CNN. Comparado ao Fast R-CNN, o Faster R-CNN apenas muda o método para gerar regiões propostas de pesquisa seletiva para rede de proposta de região. As outras partes do modelo permanecem inalteradas. O processo de computação de rede de proposta de região detalhada é descrito abaixo:

1. Usamos uma camada convolucional 3×3 com um preenchimento de 1 para transformar a saída CNN e definir o número de canais de saída para c. Assim, cada elemento no mapa de recursos que a CNN extrai da imagem é um novo recurso com um comprimento de c.
2. Usamos cada elemento no mapa de recursos como um centro para gerar várias caixas de âncora de diferentes tamanhos e proporções de aspecto e, em seguida, etiquetá-las.
3. Usamos os recursos dos elementos de comprimento c no centro da âncora caixas para prever a categoria binária (objeto ou fundo) e caixa delimitadora para suas respectivas caixas de âncora.

4. Em seguida, usamos a supressão não máxima para remover resultados semelhantes da caixa delimitadora que correspondem às previsões da categoria de "objeto". Finalmente, nós produzimos as caixas delimitadoras previstas como as regiões propostas exigidas pelo *pooling* de RoI camada.

É importante notar que, como parte do modelo R-CNN mais rápido, a rede proposta da região é treinada em conjunto com o resto do modelo. Além disso, as funções de objeto do Faster R-CNN incluem as predições de categoria e caixa delimitadora na detecção de objetos, bem como a categoria binária e predições de caixa delimitadora para as caixas de âncora na rede de proposta da região. Finalmente, a rede proposta de região pode aprender como gerar regiões propostas de alta qualidade, o que reduz o número de regiões propostas, enquanto mantém a precisão da detecção de objetos.

13.8.4 Máscara R-CNN

Se os dados de treinamento forem rotulados com as posições de nível de pixel de cada objeto em uma imagem, um modelo Mask R-CNN pode usar efetivamente esses rótulos detalhados para melhorar ainda mais a precisão da detecção de objeto.

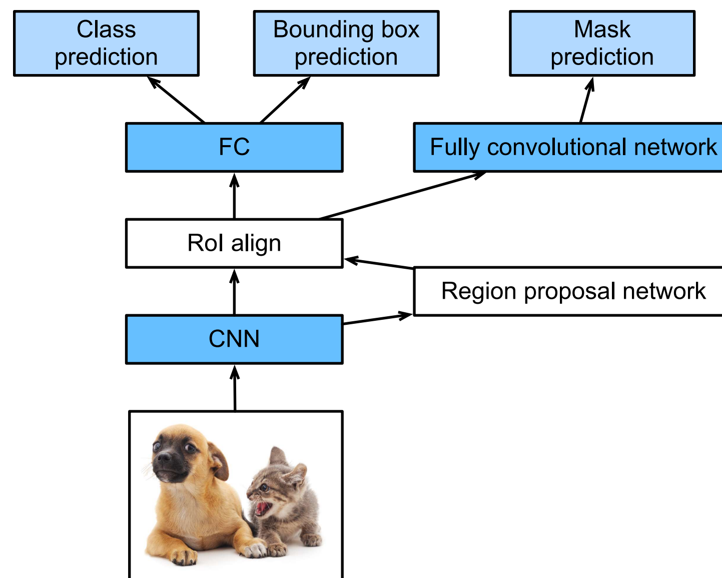


Fig. 13.8.5: Modelo de máscara R-CNN.

Conforme mostrado em Fig. 13.8.5, Mask R-CNN é uma modificação do modelo Faster R-CNN. Os modelos de máscara R-CNN substituem a camada de pooling RoI por uma camada de alinhamento RoI. Isso permite o uso de interpolação bilinear para reter informações espaciais em mapas de características, tornando o Mask R-CNN mais adequado para previsões em nível de pixel. A camada de alinhamento de RoI produz mapas de recursos do mesmo formato para todos os RoIs. Isso não apenas prevê as categorias e caixas delimitadoras de RoIs, mas nos permite usar uma rede totalmente convolucional adicional para prever as posições de objetos em nível de pixel. Descreveremos como usar redes totalmente convolucionais para prever a semântica em nível de pixel em imagens posteriormente neste capítulo.

13.8.5 Resumo

- Um modelo R-CNN seleciona várias regiões propostas e usa um CNN para realizar computação direta e extrair os recursos de cada região proposta. Em seguida, usa esses recursos para prever as categorias e caixas delimitadoras das regiões propostas.
- Fast R-CNN melhora o R-CNN realizando apenas cálculos de encaminhamento de CNN na imagem como um todo. Ele apresenta uma camada de pooling de RoI para extrair recursos do mesmo formato de RoIs de formatos diferentes.
- O R-CNN mais rápido substitui a pesquisa seletiva usada no Fast R-CNN por uma rede de proposta de região. Isso reduz o número de regiões propostas geradas, garantindo a detecção precisa do objeto.
- Mask R-CNN usa a mesma estrutura básica que R-CNN mais rápido, mas adiciona uma camada de convolução completa para ajudar a localizar objetos no nível de pixel e melhorar ainda mais a precisão da detecção de objetos.

13.8.6 Exercícios

1. Estude a implementação de cada modelo no kit de ferramentas GluonCV[163] relacionado a esta seção.

Discussões[164]

13.9 Segmentação Semântica e o *Dataset*

no discussãoobr os problemas de detecção de objetos nas seções anteriores, usamos apenas caixas delimitadoras retangulares para rotulares ola pre objets em imagens. Nesta seção, veremos a segmentação semão semntica, que tenta, tet segmentar imagens res m dierentes categorias semânticas. Essas regiões semânticas rotulam e prevêem objetos no nível do pixel. Fig. 13.9.1 ostra uma imagemeamente segmentada, com áreas marcadas como "cachorro" , "gato" e "fundo". Como você pode ver, em comparação com a detecção de objetos, a segmentação semântica rotula áreas com bordas em nível de pixel, para uma precisão significativamente maior.

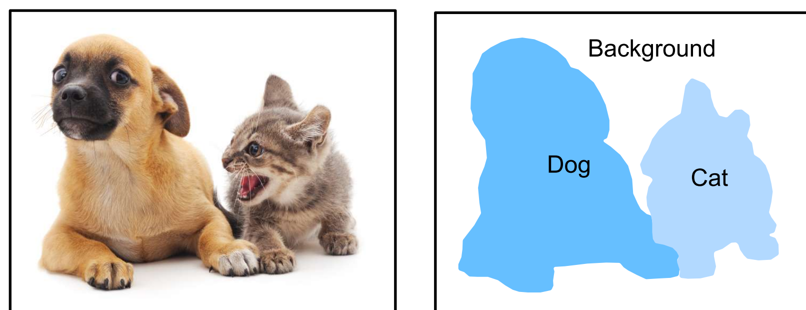


Fig. 13.9.1: Imagem segmentada semanticamente, com áreas rotuladas "cachorroSemantiatiamente mio", "gcato" e "plano de f"oundo".

[163] https://github.com/dmlc/gluon-cv/
[164] https://discuss.d2l.ai/t/1409

13.9.1 Segmentação de Imagem e Segmentação detion and Instâancia

No campo da visão computacional, existem dois métodos importantes relacionados à segmentaçãe Segmenntica: segmentação detion: imagens e segmentação detion and instâancias. Aqui, vamos distinguir esses conceitos da segmentação semântica da seguinte forma:

- A segmenta imagmgmen divide umas an imagem em várias regiões into sconstituente mét regioodo geramente usa as correlações entrel pixels ain an imagem. Durante oing treinamento, os rótulos não são necessários para pixels de imagem. No entanto, durante a previsão, esse método não pode garantir que as regiões segmentadas ain semntic imagem em .10, amage segmentação da imagem pode dividir o cão em duas regionsmaone cobverindo a boca do cão e os olhos onde o preto é a corth and eyes where black is the proeminente e a outra color and the other cobverindo o resto do cão onde o amarelo é a corg the rest of the dog where yellow is the proeminente.
- A segmentação da instância também é chamada de detecção e segmentação simultâneas. Este método tenta identificar as regiões de nível de pixel de cada color.
- Instance segmentation is also called simultaneous detection and segmentation. This method attempts to identify the pixel-level regions of each object instâancia de objeto em umae in an prominent corin prominent instnc imagem.mIn cont semnticraste com a segmentação semântica, a segmentação de instância não apenas sgmentation distinguhes semâanticas, ms o differetes instane objeto. Se umaes. I imagem contém dois cães, a segmentação de instância distingue quais pixels pertencem a cada cachorro.

13.9.2 O Conjunto de Dados de Segmentação Semântica Pascal VOC2012

No campo de segmentaçãoains two dogswill distinguish which pixels belong to which dog.

13.9.3 The Pascal VOC2012 Semantic Segmentation Dataset

In the semâantica, um conjunto de dados importante é segmentation field, one important dataset is Pascal VOC2012[165]. Para entender melhor este conjunto de dados, devemos primeirootkage our móoduloe necessário para oeded for t experimento.

```
%matplotlib inline
import os
import torch
import torchvision
from d2l import torch as d2l
```

O site original pode ser instável, portanto, baixamos os dados de um site espelho. O arquivo tem cerca de 2 GB, por isso levará algum tempo para fazer the arquchivoe, o conjunto de dados está localizado noo download. decomp mi `../data/VOCdevkit/VOC2012`.

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

[165] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

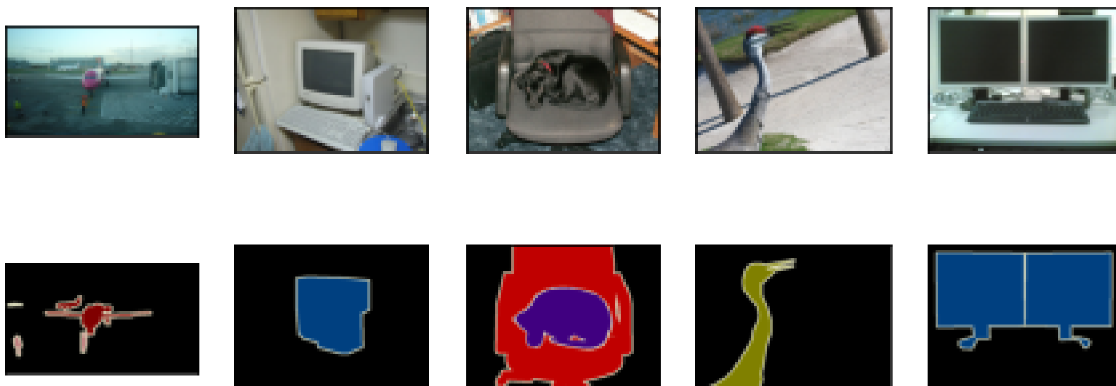
```
Downloading ../data/VOCtrainval_11-May-2012.tar from http://d2l-data.s3-accelerate.amazonaws.
↪com/VOCtrainval_11-May-2012.tar...
```

Vá para `../data/VOCdevkit/VOC2012` para ver as diferentes partes do conjunto de dados. O caminho `ImageSets/Segmentation` contém arquivos de texto que especificam os exemplos de treinamento e teste. Os cin `JPEGImages` e`SegmentationClass` contêm aspaths contain the example input imagens de entrada de exemplo e rótulos, respectivamente. Essas etiquetas também estão em formato de imagem, com as mesmasand labels, respectively. These labels are also in image format, with the same images dimensõei das put imagens de entrada às quais correspondem. Nos rótulos, os pixels com a mesma cor pertencem à mesma.

```python
#@save
def read_voc_images(voc_dir, is_train=True):
    """Read all VOC feature and label images."""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    mode = torchvision.io.image.ImageReadMode.RGB
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'SegmentationClass' ,f'{fname}.png'), mode))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

Desenhamos as primeiras cincoWe draw the first five input imagens de entrada e seus rótulos. Nas imagens do rótulo, o brancoand their labels. In the label images, white representa as bordas e o preto representa o fers and black represents the backgroundo. Outras corether colors correspondem a diferentes categorias.

```python
n = 5
imgs = train_features[0:n] + train_labels[0:n]
imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs, 2, n);
```



A seguir, listamos cada valor de cor RGB nos rótulos e as categoriaes que eles rl.

```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

Depois de definir as duasAfter defining the two constantes acima, podemos encontrar facilmente o índice de categoria para cada pixel nos rótulobove, we can easily find the category index for each pixel in the labels.

```
#@save
def build_colormap2label():
    """Build an RGB color to label mapping for segmentation."""
    colormap2label = torch.zeros(256 ** 3, dtype=torch.long)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[(colormap[0]*256 + colormap[1])*256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
    """Map an RGB color to a label."""
    colormap = colormap.permute(1,2,0).numpy().astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

PFor exeamploe, na primeira imagem dein the first exeamplo, o índice de categoria para a parte frontal do avião é 1 e o índice para o fundo ée image, the cat 0.

```
y = voc_label_indices(train_labels[0], build_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

```
(tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]]),
 'aeroplane')
```

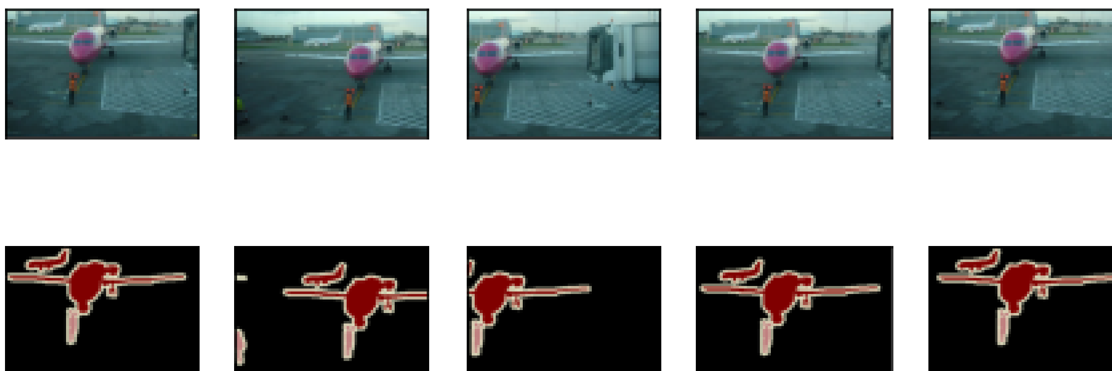
Pré-processamento de Dados

NosData Preprocessing

In the preceding chapítulos anteriores, dimensionamos asters, we scaled imagens para que se ajustassem à forma de entrada doto make them fit the inputmodelo. NaIn segmentação semântica, esse método exigiria que mapeamos novamente as categorias de pixels previstas de volta à imagem de entrada do tamanho original. Seria muito difícil fazer isso commantic segmentation, this method would require us to re-map the predicted pixel categories back to the original-size input image. It would be very difficult to do this precisãoely, especialmente em regiões segmentadas com semânticasly in segmented regions with differentes. Para evitar esse semantics. To avoid this problema, recortamos aswe crop the imagens para definir as dimensões e não as dimensionamos. Especificamente, usamos o método de corte aleatório usado no aumento da imagem para cortar a mesma região dasto set dimensions and do not scale them. Specifically, we use the random cropping method used in image augmentation to crop the same region from input imagens de entrada e seus rótuloand their labels.

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Randomly crop for both feature and label images."""
    rect = torchvision.transforms.RandomCrop.get_params(feature,
                                                         (height, width))
    feature = torchvision.transforms.functional.crop(feature, *rect)
    label = torchvision.transforms.functional.crop(label, *rect)
    return feature, label
```

```
imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)

imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```



Datasets para Segmentação Semântica Personalizada

Usamos a classe `Dataset` herdada fornecida pelo Gluon para personalizar a classe de conjunto de dados de segmentação semântica `VOCSegDataset`. IBy implementando a funçãoing the `__getitem__`, podemos acessar arbitrariamente a imagem de entrada com o índice `idx` e os índices de categoria para cada um de seus pixels do conjunto de dados. Como algumas function, we can arbitrarily access the input image with the index `idx` and the category indexes for each of its pixels from the dataset. As some imagens no conjunto de dados podem ser menores do que as dimensões de saída especificadas para corte aleatório, devemos remover essesin the dataset may be smaller than the output dimensions specified for random cropping, we must remove these exeamplos usando uma função `filter` personalizada. Além disso, definimos a função `normalize_image` para normalizar cada um dos três canais RGB das imagens de entradae by using a custom `filter` function. In addition, we define the `normalize_image` function to normalize each of the three RGB channels of the input images.

```
#@save
class VOCSegDataset(torch.utils.data.Dataset):
    """A customized dataset to load VOC dataset."""

    def __init__(self, is_train, crop_size, voc_dir):
        self.transform = torchvision.transforms.Normalize(
            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = build_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return self.transform(img.float())

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[1] >= self.crop_size[0] and
            img.shape[2] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature, voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

Lendo oReading the Dataset

Usando a classeing the custom `VOCSegDataset` personalizada, criamos o conjunto dclass, we create the treainamento e as instâncias do conjunto de teste. Assumimos que a operação de corte aleatório produzing set and testing set instances. We assume the random cropping operation output imagens no formatoin the shape 320×480. Abaixo, podemos ver o número deBelow, we can see the number of exeamploes retidos nos conjuntos dained in the treainamento e testeing and testing sets.

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

Definimos o tamanho do lote comWe set the batch size to 64 eand definimos ose the iteradtores para os conjuntos dfor the treainamento e teste. Imprimimos a forma do primeiroing and testing sets. Print the shape of the first minibatch. EmIn contraste com a to image classificação de imagens e o reconhecimento de objetos, os rótulos aqui são matrizes trition and object recognition, labels here are three-dimensionail arrays.

```
batch_size = 64
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
                                    drop_last=True,
                                    num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
torch.Size([64, 3, 320, 480])
torch.Size([64, 320, 480])
```

Juntando Tudo

Finalmente, definimos uma função `load_data_voc` que baixa e carrega este *Putting All Things Together

Finally, we define a function `load_data_voc` that downloads and loads this dataset*, e entãoand then retourna oss the data iteradtores de dados.

```
#@save
def load_data_voc(batch_size, crop_size):
    """Download and load the VOC2012 semantic dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = torch.utils.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, drop_last=True, num_workers=num_workers)
```

(continues on next page)

(continued from previous page)

```
    test_iter = torch.utils.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        drop_last=True, num_workers=num_workers)
    return train_iter, test_iter
```

13.9.4 Resumo

- A segmentação semântica analisa como asSummary
- Semantic segmentation looks at how imagens podem ser segmentadas em regiões comcan be segmented into regions with differentes categorias semânticas.
- No campo de segmentação semântica, um conjunto de dados importante é semantic categories.
- In the semantic segmentation field, one important dataset is Pascal VOC2012.
- Como asBecause the input imagens e rótulos de entrada na segmentação semântica têm umaand labels in semantic segmentation have a one-to-one correspondêencia um a um no nível do pixel, nós os cortamos aleatoriamente em um tamanho fixo, em vez de dimensionálose at the pixel level, we randomly crop them to a fixed size, rather than scaling them.

13.9.5 Exercícioises

1. Lembre-se do conteúdo que vimos emRecall the content we covered in Section 13.1. Qual dos métodos dWhich of the image augmento de imagem usados na classificação de imagens seria difícil de usar na segmentação semânticaation methods used in image classification would be hard to use in semantic segmentation?

Discussõeions[166]

13.10 Convolução Transposta

As camadas que apresentamos até agora para redes neurais convolucionais, incluindo camadas convolucionais (Section 6.2) e camadas de pooling (Section 6.5), geralmente reduzem a largura e altura de entrada ou as mantêm inalteradas. Aplicativos como segmentação semântica (`sec_semantic_segmentation`) e redes adversárias geradoras (Section 17.2), no entanto, exigem prever valores para cada pixel e, portanto, precisam aumentar a largura e altura de entrada. A convolução transposta, também chamada de convolução fracionada (Dumoulin & Visin, 2016) ou deconvolução (Long et al., 2015), serve a este propósito.

```
import torch
from torch import nn
from d2l import torch as d2l
```

[166] https://discuss.d2l.ai/t/1480

13.10.1 Convolução Transposta 2D Básica

Vamos considerar um caso básico em que os canais de entrada e saída são 1, com 0 preenchimento e 1 passo. Fig. 13.10.1 ilustra como a convolução transposta com um *kernel* 2×2 é calculada na matriz de entrada 2×2.

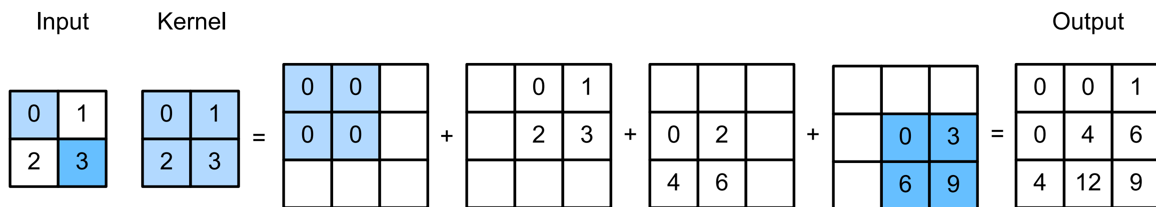


Fig. 13.10.1: Camada de convolução transposta com um *kernel* 2×2.

Podemos implementar essa operação fornecendo o *kernel* da matriz K e a entrada da matriz X.

```
def trans_conv(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

Lembre-se de que a convolução calcula os resultados por `Y[i, j] = (X[i: i + h, j: j + w] * K).sum()` (consulte `corr2d` em Section 6.2), que resume os valores de entrada por meio do *kernel*. Enquanto a convolução transposta transmite valores de entrada por meio do *kernel*, o que resulta em uma forma de saída maior.

Verifique os resultados em Fig. 13.10.1.

```
X = torch.tensor([[0., 1], [2, 3]])
K = torch.tensor([[0., 1], [2, 3]])
trans_conv(X, K)
```

```
tensor([[ 0.,  0.,  1.],
        [ 0.,  4.,  6.],
        [ 4., 12.,  9.]])
```

Ou podemos usar `nn.ConvTranspose2d` para obter os mesmos resultados. Como `nn.Conv2d`, tanto a entrada quanto o *kernel* devem ser tensores 4-D.

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[ 0.,  0.,  1.],
          [ 0.,  4.,  6.],
          [ 4., 12.,  9.]]]], grad_fn=<SlowConvTranspose2DBackward>)
```

13.10.2 Preenchimento, Passos e Canais

Aplicamos elementos de preenchimento à entrada em convolução, enquanto eles são aplicados à saída em convolução transposta. Um preenchimento 1×1 significa que primeiro calculamos a saída como normal e, em seguida, removemos as primeiras/últimas linhas e colunas.

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, padding=1, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[4.]]]], grad_fn=<SlowConvTranspose2DBackward>)
```

Da mesma forma, os avanços também são aplicados às saídas.

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, stride=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[0., 0., 0., 1.],
          [0., 0., 2., 3.],
          [0., 2., 0., 3.],
          [4., 6., 6., 9.]]]], grad_fn=<SlowConvTranspose2DBackward>)
```

A extensão multicanal da convolução transposta é igual à convolução. Quando a entrada tem vários canais, denotados por c_i, a convolução transposta atribui uma matriz de *kernel* $k_h \times k_w$ a cada canal de entrada. Se a saída tem um tamanho de canal c_o, então temos um *kernel* $c_i \times k_h \times k_w$ para cada canal de saída.

Como resultado, se alimentarmos X em uma camada convolucional f para calcular $Y = f(X)$ e criarmos uma camada de convolução transposta g com os mesmos hiperparâmetros de f, exceto para o conjunto de canais de saída para ter o tamanho do canal de X, então $g(Y)$ deve ter o mesmo formato que X. Deixe-nos verificar esta afirmação.

```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

```
True
```

13.10.3 Analogia à Transposição de Matriz

A convolução transposta leva o nome da transposição da matriz. Na verdade, as operações de convolução também podem ser realizadas por multiplicação de matrizes. No exemplo abaixo, definimos uma entrada X 3×3 com *kernel* K 2×2, e então usamos `corr2d` para calcular a saída da convolução.

```
X = torch.arange(9.0).reshape(3, 3)
K = torch.tensor([[0, 1], [2, 3]])
```

(continues on next page)

(continued from previous page)

```
Y = d2l.corr2d(X, K)
Y
```

```
tensor([[19., 25.],
        [37., 43.]])
```

A seguir, reescrevemos o *kernel* de convolução K como uma matriz W. Sua forma será $(4, 9)$, onde a linha i^th presente aplicando o *kernel* à entrada para gerar o i^th elemento de saída.

```
def kernel2matrix(K):
    k, W = torch.zeros(5), torch.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

```
tensor([[0., 1., 0., 2., 3., 0., 0., 0., 0.],
        [0., 0., 1., 0., 2., 3., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 2., 3., 0.],
        [0., 0., 0., 0., 0., 1., 0., 2., 3.]])
```

Então, o operador de convolução pode ser implementado por multiplicação de matriz com remodelagem adequada.

```
Y == torch.mv(W, X.reshape(-1)).reshape(2, 2)
```

```
tensor([[True, True],
        [True, True]])
```

Podemos implementar a convolução transposta como uma multiplicação de matriz reutilizando `kernel2matrix`. Para reutilizar o W gerado, construímos uma entrada 2×2, de modo que a matriz de peso correspondente terá uma forma $(9, 4)$, que é $W^\top$. Deixe-nos verificar os resultados.

```
X = torch.tensor([[0.0, 1], [2, 3]])
Y = trans_conv(X, K)
Y == torch.mv(W.T, X.reshape(-1)).reshape(3, 3)
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

13.10.4 Resumo

- Em comparação com as convoluções que reduzem as entradas por meio de *kernels*, as convoluções transpostas transmitem as entradas.
- Se uma camada de convolução reduz a largura e altura de entrada em n_w e h_h tempo, respectivamente. Então, uma camada de convolução transposta com os mesmos tamanhos de *kernel*, preenchimento e passos aumentará a largura e altura de entrada em n_w e h_h, respectivamente.
- Podemos implementar operações de convolução pela multiplicação da matriz, as convoluções transpostas correspondentes podem ser feitas pela multiplicação da matriz transposta.

13.10.5 Exercícios

1. É eficiente usar a multiplicação de matrizes para implementar operações de convolução? Por quê?

Discussões[167]

13.11 Redes Totalmente Convolucionais (*Fully Convolutional Networks*, FCN)

Discutimos anteriormente a segmentação semântica usando cada pixel em uma imagem para previsão de categoria. Uma rede totalmente convolucional (FCN) (Long et al., 2015) usa uma rede neural convolucional para transformar os pixels da imagem em categorias de pixels. Ao contrário das redes neurais convolucionais previamente introduzidas, uma FCN transforma a altura e largura do mapa de recurso da camada intermediária de volta ao tamanho da imagem de entrada por meio do camada de convolução transposta, de modo que as previsões tenham uma correspondência com a imagem de entrada em dimensão espacial (altura e largura). Dado uma posição na dimensão espacial, a saída da dimensão do canal será uma previsão de categoria do pixel correspondente ao local.

Primeiro importaremos o pacote ou módulo necessário para o experimento e depois explicaremos a camada de convolução transposta.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

[167] https://discuss.d2l.ai/t/1450

13.11.1 Construindo um Modelo

Aqui, demonstramos o projeto mais básico de um modelo de rede totalmente convolucional. Conforme mostrado em Fig. 13.11.1, a rede totalmente convolucional primeiro usa a rede neural convolucional para extrair características da imagem, então transforma o número de canais no número de categorias através da camada de convolução 1×1 e, finalmente, transforma a altura e largura do mapa de recursos para o tamanho da imagem de entrada usando a camada de convolução transposta Section 13.10. A saída do modelo tem a mesma altura e largura da imagem de entrada e uma correspondência de um para um nas posições espaciais. O canal de saída final contém a previsão da categoria do pixel da posição espacial correspondente.

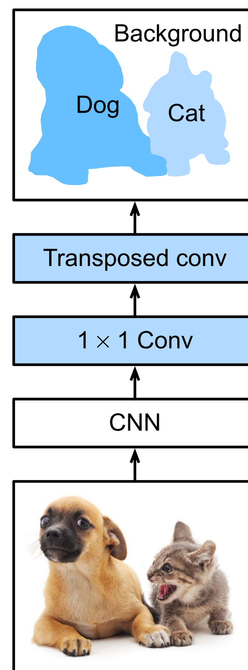


Fig. 13.11.1: Rede totalmente convolucional.

Abaixo, usamos um modelo ResNet-18 pré-treinado no conjunto de dados ImageNet para extrair recursos de imagem e registrar a instância de rede como `pretrained_net`. Como você pode ver, as duas últimas camadas da variável membro do modelo `features` são a camada de agrupamento global médio`GlobalAvgPool2D` e a camada de nivelamento de exemplo `Flatten` O módulo `output` contém a camada totalmente conectada usada para saída. Essas camadas não são necessárias para uma rede totalmente convolucional.

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
pretrained_net.layer4[1], pretrained_net.avgpool, pretrained_net.fc
```

```
(BasicBlock(
   (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
   (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   (relu): ReLU(inplace=True)
   (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
   (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 ),
```

(continues on next page)

(continued from previous page)

```
AdaptiveAvgPool2d(output_size=(1, 1)),
Linear(in_features=512, out_features=1000, bias=True))
```

Em seguida, criamos a instância de rede totalmente convolucional `net`. Ela duplica todas as camadas neurais, exceto as duas últimas camadas da variável membro de instância `features` de `pretrained_net` e os parâmetros do modelo obtidos após o pré-treinamento.

```
net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

Dada uma entrada de altura e largura de 320 e 480 respectivamente, o cálculo direto de `net` reduzirá a altura e largura da entrada para $1/32$ do original, ou seja, 10 e 15.

```
X = torch.rand(size=(1, 3, 320, 480))
net(X).shape
```

```
torch.Size([1, 512, 10, 15])
```

Em seguida, transformamos o número de canais de saída para o número de categorias de Pascal VOC2012 (21) por meio da camada de convolução 1×1. Finalmente, precisamos ampliar a altura e largura do mapa de feições por um fator de 32 para alterá-los de volta para a altura e largura da imagem de entrada. Lembre-se do cálculo método para a forma de saída da camada de convolução descrita em Section 6.3. Porque $(320-64+16\times2+32)/32 = 10$ e $(480-64+16\times2+32)/32 = 15$, construímos uma camada de convolução transposta com uma distância de 32 e definimos a altura e largura do *kernel* de convolução para 64 e o preenchimento para 16. Não é difícil ver que, se o passo for s, o preenchimento é $s/2$ (assumindo que $s/2$ é um inteiro), e a altura e largura do *kernel* de convolução são $2s$, o *kernel* de convolução transposto aumentará a altura e a largura da entrada por um fator de s.

```
num_classes = 21
net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
net.add_module('transpose_conv', nn.ConvTranspose2d(num_classes, num_classes,
                                    kernel_size=64, padding=16, stride=32))
```

13.11.2 Inicializando a Camada de Convolução Transposta

Já sabemos que a camada de convolução transposta pode ampliar um mapa de feições. No processamento de imagem, às vezes precisamos ampliar a imagem, ou seja, *upsampling*. Existem muitos métodos para aumentar a amostragem e um método comum é a interpolação bilinear. Simplesmente falando, para obter o pixel da imagem de saída nas coordenadas (x, y), as coordenadas são primeiro mapeadas para as coordenadas da imagem de entrada (x', y'). Isso pode ser feito com base na proporção do tamanho de três entradas em relação ao tamanho da saída. Os valores mapeados x' e y' são geralmente números reais. Então, encontramos os quatro pixels mais próximos da coordenada (x', y') na imagem de entrada. Finalmente, os pixels da imagem de saída nas coordenadas (x, y) são calculados com base nesses quatro pixels na imagem de entrada e suas distâncias relativas a (x', y'). O *upsampling* por interpolação bilinear pode ser implementado pela camada de convolução transposta do *kernel* de convolução construído usando a seguinte função `bilinear_kernel`. Devido a limitações de espaço, fornecemos apenas a implementação da função `bilinear_kernel` e não discutiremos os princípios do algoritmo.

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (torch.arange(kernel_size).reshape(-1, 1),
          torch.arange(kernel_size).reshape(1, -1))
    filt = (1 - torch.abs(og[0] - center) / factor) * \
           (1 - torch.abs(og[1] - center) / factor)
    weight = torch.zeros((in_channels, out_channels,
                          kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return weight
```

Agora, vamos experimentar com upsampling de interpolação bilinear implementado por camadas de convolução transpostas. Construa uma camada de convolução transposta que amplie a altura e a largura da entrada por um fator de 2 e inicialize seu kernel de convolução com a função `bilinear_kernel`.

```
conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4, padding=1, stride=2,
                                bias=False)
conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

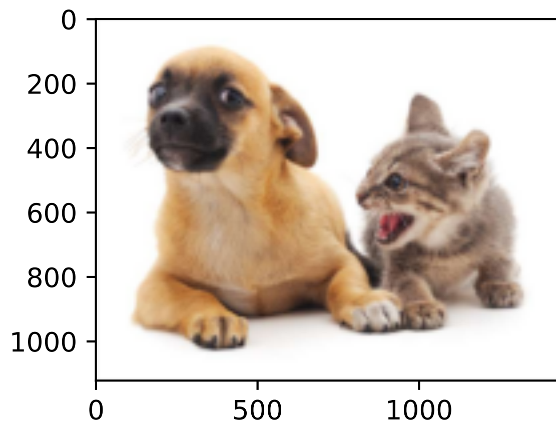
Leia a imagem `X` e registre o resultado do upsampling como`Y`. Para imprimir a imagem, precisamos ajustar a posição da dimensão do canal.

```
img = torchvision.transforms.ToTensor()(d2l.Image.open('../img/catdog.jpg'))
X = img.unsqueeze(0)
Y = conv_trans(X)
out_img = Y[0].permute(1, 2, 0).detach()
```

Como você pode ver, a camada de convolução transposta amplia a altura e largura da imagem em um fator de 2. Vale ressaltar que, além da diferença na escala de coordenadas, a imagem ampliada por interpolação bilinear e a imagem original impressa em Section 13.3 tem a mesma aparência.

```
d2l.set_figsize()
print('input image shape:', img.permute(1, 2, 0).shape)
d2l.plt.imshow(img.permute(1, 2, 0));
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img);
```

```
input image shape: torch.Size([561, 728, 3])
output image shape: torch.Size([1122, 1456, 3])
```



Em uma rede totalmente convolucional, inicializamos a camada de convolução transposta para interpolação bilinear com upsampled. Para uma camada de convolução 1×1, usamos o Xavier para inicialização aleatória.

```
W = bilinear_kernel(num_classes, num_classes, 64)
net.transpose_conv.weight.data.copy_(W);
```

13.11.3 Lendo o *Dataset*

Lemos o *dataset* usando o método descrito na seção anterior. Aqui, especificamos a forma da imagem de saída cortada aleatoriamente como 320×480, portanto, a altura e a largura são divisíveis por 32.

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

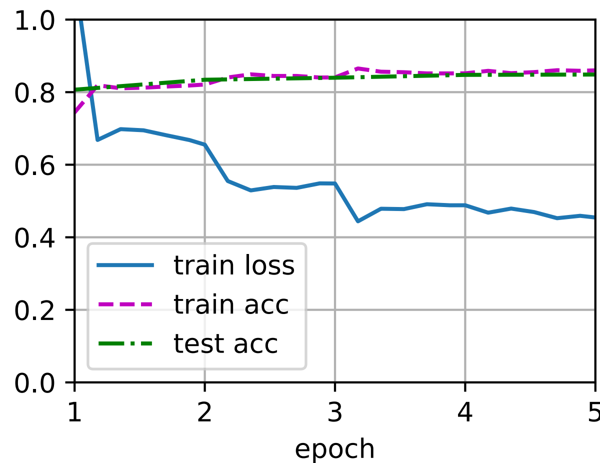
```
read 1114 examples
read 1078 examples
```

13.11.4 Treinamento

Agora podemos começar a treinar o modelo. A função de perda e o cálculo de precisão aqui não são substancialmente diferentes daqueles usados na classificação de imagens. Como usamos o canal da camada de convolução transposta para prever as categorias de pixels, a opção `axis = 1` (dimensão do canal) é especificada em `SoftmaxCrossEntropyLoss`. Além disso, o modelo calcula a precisão com base em se a categoria de previsão de cada pixel está correta.

```
def loss(inputs, targets):
    return F.cross_entropy(inputs, targets, reduction='none').mean(1).mean(1)

num_epochs, lr, wd, devices = 5, 0.001, 1e-3, d2l.try_all_gpus()
trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.455, train acc 0.860, test acc 0.849
222.9 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



13.11.5 Predição

Durante a previsão, precisamos padronizar a imagem de entrada em cada canal e transformá-los no formato de entrada quadridimensional exigido pela rede neural convolucional.

```
def predict(img):
    X = test_iter.dataset.normalize_image(img).unsqueeze(0)
    pred = net(X.to(devices[0])).argmax(dim=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

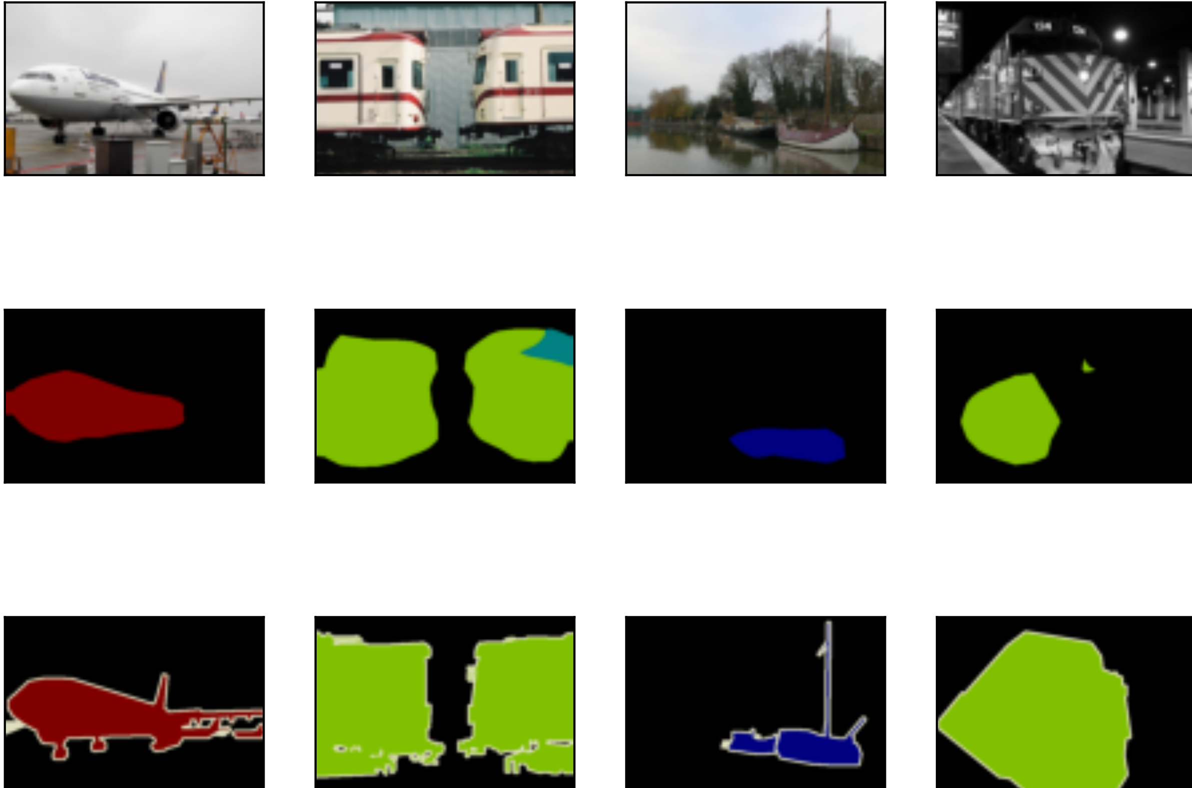
Para visualizar as categorias previstas para cada pixel, mapeamos as categorias previstas de volta às suas cores rotuladas no conjunto de dados.

```
def label2image(pred):
    colormap = torch.tensor(d2l.VOC_COLORMAP, device=devices[0])
    X = pred.long()
    return colormap[X, :]
```

O tamanho e a forma das imagens no conjunto de dados de teste variam. Como o modelo usa uma camada de convolução transposta com uma distância de 32, quando a altura ou largura da imagem de entrada não é divisível por 32, a altura ou largura da saída da camada de convolução transposta se desvia do tamanho da imagem de entrada. Para resolver esse problema, podemos recortar várias áreas retangulares na imagem com alturas e larguras como múltiplos inteiros de 32 e, em seguida, realizar cálculos para a frente nos pixels nessas áreas. Quando combinadas, essas áreas devem cobrir completamente a imagem de entrada. Quando um pixel é coberto por várias áreas, a média da saída da camada de convolução transposta no cálculo direto das diferentes áreas pode ser usada como uma entrada para a operação softmax para prever a categoria.

Para simplificar, lemos apenas algumas imagens de teste grandes e recortamos uma área com um formato de 320×480 no canto superior esquerdo da imagem. Apenas esta área é usada para previsão. Para a imagem de entrada, imprimimos primeiro a área cortada, depois imprimimos o resultado previsto e, por fim, imprimimos a categoria rotulada.

```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 320, 480)
    X = torchvision.transforms.functional.crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X.permute(1,2,0), pred.cpu(),
             torchvision.transforms.functional.crop(
                 test_labels[i], *crop_rect).permute(1,2,0)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



13.11.6 Resumo

- A rede totalmente convolucional primeiro usa a rede neural convolucional para extrair características da imagem, depois transforma o número de canais no número de categorias por meio da camada de convolução 1×1 e, finalmente, transforma a altura e largura do mapa de características para o tamanho da imagem de entrada usando a camada de convolução transposta para produzir a categoria de cada pixel.
- Em uma rede totalmente convolucional, inicializamos a camada de convolução transposta para interpolação bilinear com *upsampling*.

13.11.7 Exercícios

1. Se usarmos Xavier para inicializar aleatoriamente a camada de convolução transposta, o que acontecerá com o resultado?
2. Você pode melhorar ainda mais a precisão do modelo ajustando os hiperparâmetros?
3. Preveja as categorias de todos os pixels na imagem de teste.
4. As saídas de algumas camadas intermediárias da rede neural convolucional também são usadas no artigo sobre redes totalmente convolucionais (Long et al., 2015). Tente implementar essa ideia.

Discussões[168]

13.12 Transferência de Estilo Neural

Se você usa aplicativos de compartilhamento social ou é um fotógrafo amador, está familiarizado com os filtros. Os filtros podem alterar os estilos de cor das fotos para tornar o fundo mais nítido ou o rosto das pessoas mais branco. No entanto, um filtro geralmente só pode alterar um aspecto de uma foto. Para criar a foto ideal, muitas vezes você precisa tentar muitas combinações de filtros diferentes. Esse processo é tão complexo quanto ajustar os hiperparâmetros de um modelo.

Nesta seção, discutiremos como podemos usar redes neurais de convolução (CNNs) para aplicar automaticamente o estilo de uma imagem a outra imagem, uma operação conhecida como transferência de estilo (Gatys et al., 2016). Aqui, precisamos de duas imagens de entrada, uma imagem de conteúdo e uma imagem de estilo. Usamos uma rede neural para alterar a imagem do conteúdo de modo que seu estilo espelhe o da imagem do estilo. Em Fig. 13.12.1, a imagem do conteúdo é uma foto de paisagem que o autor tirou na Mount Rainier National Part perto de Seattle. A imagem do estilo é uma pintura a óleo de carvalhos no outono. A imagem composta de saída retém as formas gerais dos objetos na imagem de conteúdo, mas aplica a pincelada de pintura a óleo da imagem de estilo e torna a cor geral mais vívida.

[168] https://discuss.d2l.ai/t/1582

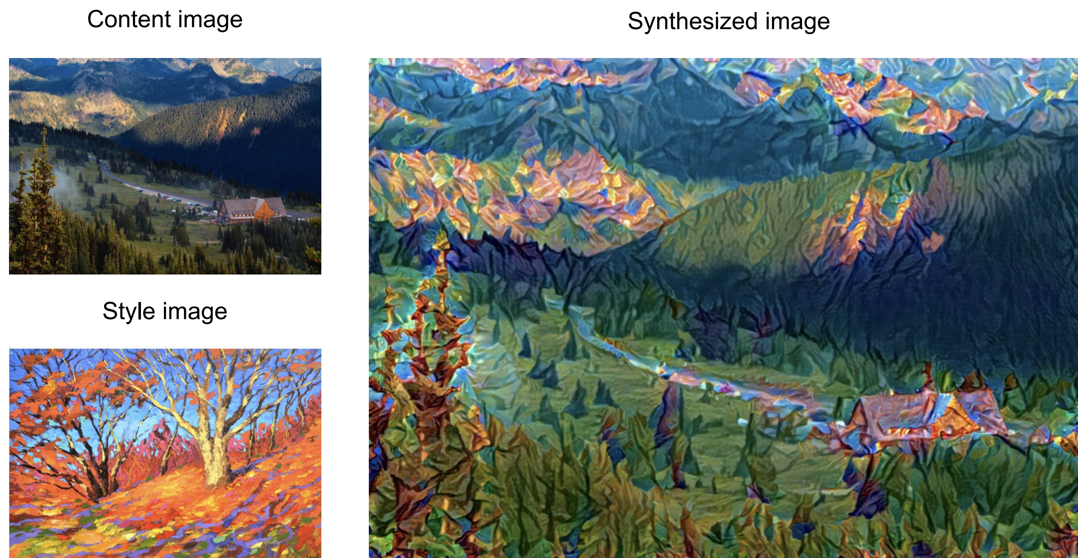


Fig. 13.12.1: Imagens de entrada de conteúdo e estilo e imagem composta produzida por transferência de estilo.

13.12.1 Técnica

O modelo de transferência de estilo baseado em CNN é mostrado em Fig. 13.12.2. Primeiro, inicializamos a imagem composta. Por exemplo, podemos inicializá-la como a imagem do conteúdo. Esta imagem composta é a única variável que precisa ser atualizada no processo de transferência de estilo, ou seja, o parâmetro do modelo a ser atualizado na transferência de estilo. Em seguida, selecionamos uma CNN pré-treinada para extrair recursos de imagem. Esses parâmetros do modelo não precisam ser atualizados durante o treinamento. O CNN profundo usa várias camadas neurais que extraem sucessivamente recursos de imagem. Podemos selecionar a saída de certas camadas para usar como recursos de conteúdo ou recursos de estilo. Se usarmos a estrutura em Fig. 13.12.2, a rede neural pré-treinada contém três camadas convolucionais. A segunda camada produz os recursos de conteúdo da imagem, enquanto as saídas da primeira e terceira camadas são usadas como recursos de estilo. Em seguida, usamos a propagação para a frente (na direção das linhas sólidas) para calcular a função de perda de transferência de estilo e a propagação para trás (na direção das linhas pontilhadas) para atualizar o parâmetro do modelo, atualizando constantemente a imagem composta. As funções de perda usadas na transferência de estilo geralmente têm três partes: 1. A perda de conteúdo é usada para fazer a imagem composta se aproximar da imagem de conteúdo no que diz respeito aos recursos de conteúdo. 2. A perda de estilo é usada para fazer com que a imagem composta se aproxime da imagem de estilo em termos de recursos de estilo. 3. A perda total de variação ajuda a reduzir o ruído na imagem composta. Finalmente, depois de terminar de treinar o modelo, produzimos os parâmetros do modelo de transferência de estilo para obter a imagem composta final.

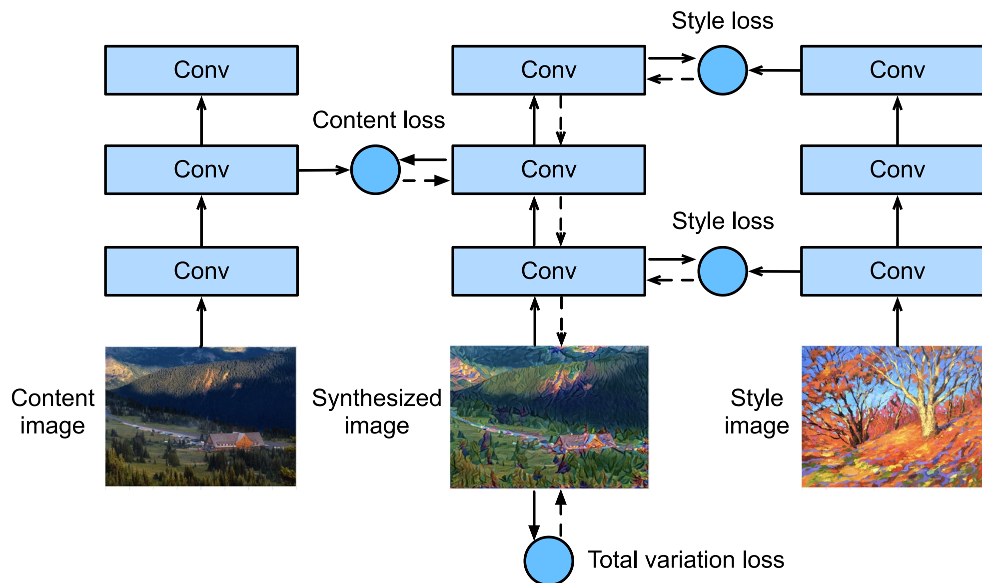


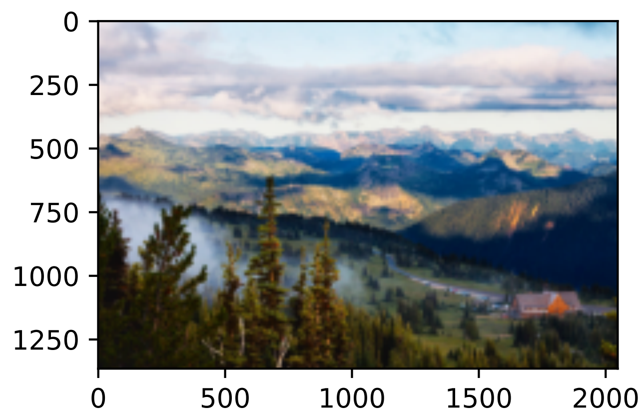
Fig. 13.12.2: Processo de transferência de estilo baseado em CNN. As linhas sólidas mostram a direção da propagação para a frente e as linhas pontilhadas mostram a propagação para trás.

A seguir, faremos um experimento para nos ajudar a entender melhor os detalhes técnicos da transferência de estilo.

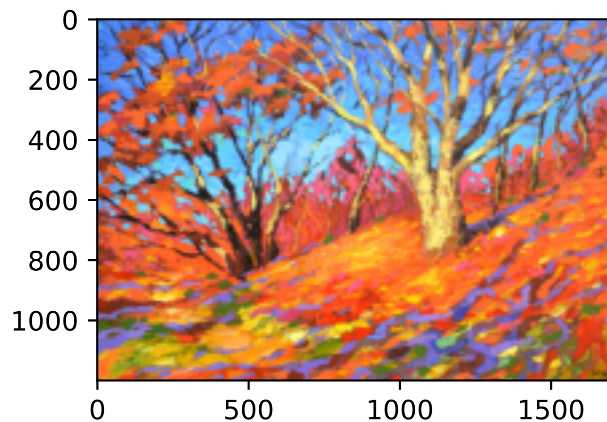
13.12.2 Lendo o Conteúdo e as Imagens de Estilo

Primeiro, lemos o conteúdo e as imagens de estilo. Ao imprimir os eixos de coordenadas da imagem, podemos ver que eles têm dimensões diferentes.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l

d2l.set_figsize()
content_img = d2l.Image.open('../img/rainier.jpg')
d2l.plt.imshow(content_img);
```



```
style_img = d2l.Image.open('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```



13.12.3 Pré-processamento e Pós-processamento

A seguir, definimos as funções de pré-processamento e pós-processamento de imagens. A função `pré-processamento` normaliza cada um dos três canais RGB das imagens de entrada e transforma os resultados em um formato que pode ser inserido na CNN. A função `postprocess` restaura os valores de pixel na imagem de saída para seus valores originais antes da normalização. Como a função de impressão de imagem requer que cada pixel tenha um valor de ponto flutuante de 0 a 1, usamos a função `clip` para substituir valores menores que 0 ou maiores que 1 por 0 ou 1, respectivamente.

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

13.12.4 Extraindo *Features*

Usamos o modelo VGG-19 pré-treinado no conjunto de dados ImageNet para extrair características da imagem [1].

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

Para extrair o conteúdo da imagem e os recursos de estilo, podemos selecionar as saídas de certas camadas na rede VGG. Em geral, quanto mais próxima uma saída estiver da camada de entrada, mais fácil será extrair informações detalhadas da imagem. Quanto mais longe uma saída estiver, mais fácil será extrair informações globais. Para evitar que a imagem composta retenha muitos detalhes da imagem de conteúdo, selecionamos uma camada de rede VGG próxima à camada de saída para produzir os recursos de conteúdo da imagem. Essa camada é chamada de camada de conteúdo. Também selecionamos as saídas de diferentes camadas da rede VGG para combinar os estilos local e global. Elas são chamadas de camadas de estilo. Como mencionamos em Section 7.2, as redes VGG têm cinco blocos convolucionais. Neste experimento, selecionamos a última camada convolucional do quarto bloco convolucional como a camada de conteúdo e a primeira camada de cada bloco como camadas de estilo. Podemos obter os índices para essas camadas imprimindo a instância `pretrained_net`.

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

Durante a extração de *features*, só precisamos usar todas as camadas VGG da camada de entrada até a camada de conteúdo ou estilo mais próxima da camada de saída. Abaixo, construímos uma nova rede, `net`, que retém apenas as camadas da rede VGG que precisamos usar. Em seguida, usamos `net` para extrair *features*.

```
net = nn.Sequential(*[pretrained_net.features[i] for i in
                      range(max(content_layers + style_layers) + 1)])
```

Dada a entrada `X`, se simplesmente chamarmos a computação direta de `net (X)`, só podemos obter a saída da última camada. Como também precisamos das saídas das camadas intermediárias, precisamos realizar computação camada por camada e reter o conteúdo e as saídas da camada de estilo.

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

A seguir, definimos duas funções: a função `get_contents` obtém as *features* de conteúdo extraídos da imagem do conteúdo, enquanto a função `get_styles` obtém os recursos de estilo extraídos da imagem de estilo. Como não precisamos alterar os parâmetros do modelo VGG pré-treinado durante o treinamento, podemos extrair as *features* de conteúdo da imagem de conteúdo e recursos de estilo da imagem de estilo antes do início do treinamento. Como a imagem composta é o parâmetro do modelo que deve ser atualizado durante a transferência do estilo, só podemos

chamar a função `extract_features` durante o treinamento para extrair o conteúdo e os recursos de estilo da imagem composta.

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

13.12.5 Definindo a Função de Perda

A seguir, veremos a função de perda usada para transferência de estilo. A função de perda inclui a perda de conteúdo, perda de estilo e perda total de variação.

Perda de Conteúdo

Semelhante à função de perda usada na regressão linear, a perda de conteúdo usa uma função de erro quadrado para medir a diferença nos recursos de conteúdo entre a imagem composta e a imagem de conteúdo. As duas entradas da função de erro quadrada são ambas saídas da camada de conteúdo obtidas da função `extract_features`.

```
def content_loss(Y_hat, Y):
    # we 'detach' the target content from the tree used
    # to dynamically compute the gradient: this is a stated value,
    # not a variable. Otherwise the loss will throw an error.
    return torch.square(Y_hat - Y.detach()).mean()
```

Perda de Estilo

A perda de estilo, semelhante à perda de conteúdo, usa uma função de erro quadrático para medir a diferença de estilo entre a imagem composta e a imagem de estilo. Para expressar a saída de estilos pelas camadas de estilo, primeiro usamos a função `extract_features` para calcular a saída da camada de estilo. Supondo que a saída tenha 1 exemplo, c canais, e uma altura e largura de h e w, podemos transformar a saída na matriz $\mathbf{X}$, que tem c linhas e $h \cdot w$ colunas. Você pode pensar na matriz $\mathbf{X}$ como a combinação dos vetores c e $\mathbf{x}_1, \ldots, \mathbf{x}_c$, que têm um comprimento de hw. Aqui, o vetor $\mathbf{x}_i$ representa a característica de estilo do canal i. Na matriz Gram desses vetores $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$, elemento x_{ij} na linha i coluna j é o produto interno dos vetores $\mathbf{x}_i$ e $\mathbf{x}_j$. Ele representa a correlação dos recursos de estilo dos canais i e j. Usamos esse tipo de matriz de Gram para representar a saída do estilo pelas camadas de estilo. Você deve notar que, quando o valor $h \cdot w$ é grande, isso geralmente leva a valores grandes na matriz de Gram. Além disso, a altura e a largura da matriz de Gram são o número de canais c. Para garantir que a perda de estilo não seja afetada pelo tamanho desses valores, definimos a função `gram` abaixo para dividir a matriz de Gram pelo número de seus elementos, ou seja, $c \cdot h \cdot w$.

```python
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

Naturalmente, as duas entradas de matriz de Gram da função de erro quadrado para perda de estilo são obtidas da imagem composta e das saídas da camada de estilo de imagem de estilo. Aqui, assumimos que a matriz de Gram da imagem de estilo, `gram_Y`, foi calculada antecipadamente.

```python
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

Perda de Variância Total

Às vezes, as imagens compostas que aprendemos têm muito ruído de alta frequência, principalmente pixels claros ou escuros. Um método comum de redução de ruído é a redução total de ruído na variação. Assumimos que $x_{i,j}$ representa o valor do pixel na coordenada (i, j), então a perda total de variância é:

$$\sum_{i,j} \left|x_{i,j} - x_{i+1,j}\right| + \left|x_{i,j} - x_{i,j+1}\right|. \tag{13.12.1}$$

Tentamos tornar os valores dos pixels vizinhos tão semelhantes quanto possível.

```python
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

Função de Perda

A função de perda para transferência de estilo é a soma ponderada da perda de conteúdo, perda de estilo e perda total de variância. Ajustando esses hiperparâmetros de peso, podemos equilibrar o conteúdo retido, o estilo transferido e a redução de ruído na imagem composta de acordo com sua importância relativa.

```python
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

13.12.6 Criação e inicialização da imagem composta

Na transferência de estilo, a imagem composta é a única variável que precisa ser atualizada. Portanto, podemos definir um modelo simples, `GeneratedImage`, e tratar a imagem composta como um parâmetro do modelo. No modelo, a computação direta retorna apenas o parâmetro do modelo.

```
class GeneratedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(GeneratedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

A seguir, definimos a função `get_inits`. Esta função cria uma instância de modelo de imagem composta e a inicializa na imagem `X`. A matriz de Gram para as várias camadas de estilo da imagem de estilo, `styles_Y_gram`, é calculada antes do treinamento.

```
def get_inits(X, device, lr, styles_Y):
    gen_img = GeneratedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

13.12.7 Treinamento

Durante o treinamento do modelo, extraímos constantemente o conteúdo e as *features* de estilo da imagem composta e calculamos a função de perda. Lembre-se de nossa discussão sobre como as funções de sincronização forçam o *front-end* a esperar pelos resultados de computação em Section 12.2. Como apenas chamamos a função de sincronização `asnumpy` a cada 10 épocas, o processo pode ocupar uma grande quantidade de memória. Portanto, chamamos a função de sincronização `waitall` durante cada época.

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
            X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
```

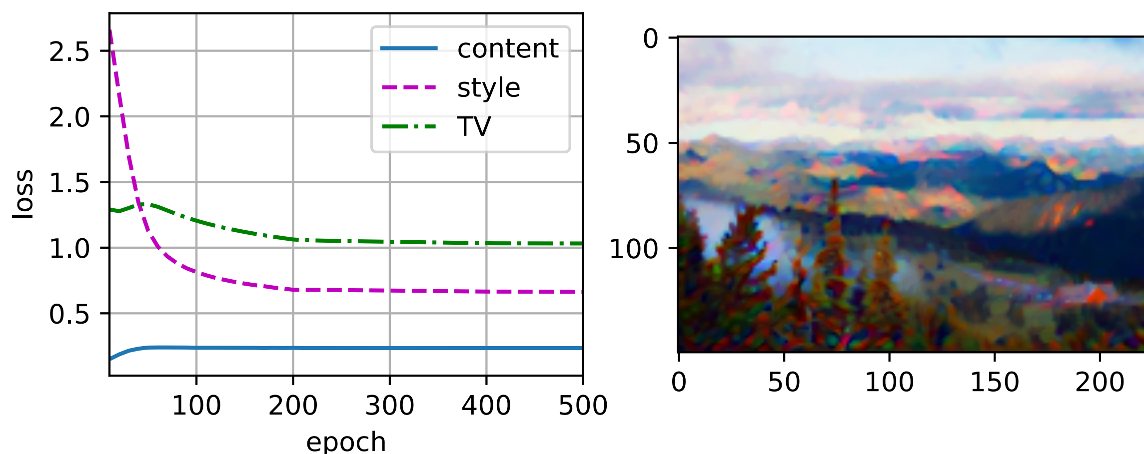
(continues on next page)

(continued from previous page)

```
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
                                     float(sum(styles_l)), float(tv_l)])
    return X
```

Em seguida, começamos a treinar o modelo. Primeiro, definimos a altura e a largura das imagens de conteúdo e estilo para 150 por 225 pixels. Usamos a imagem de conteúdo para inicializar a imagem composta.

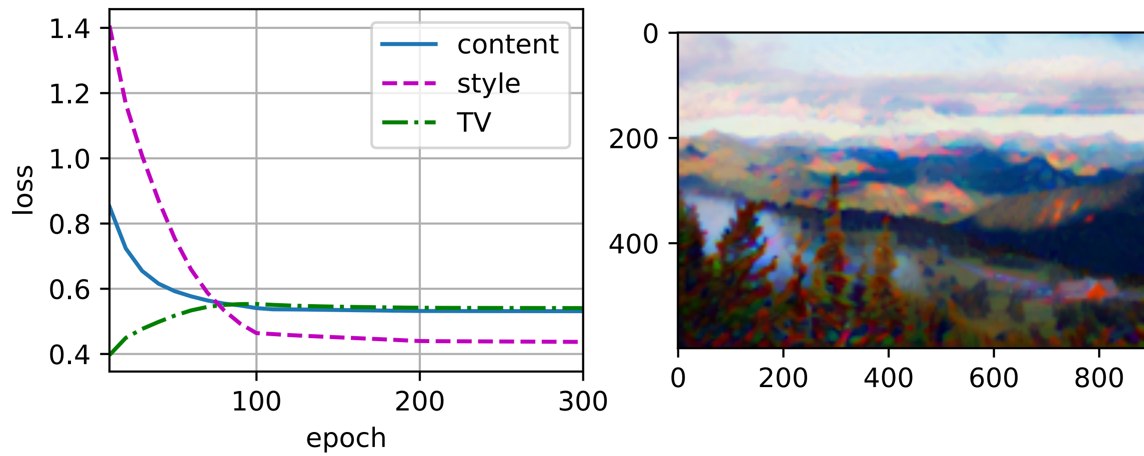
```
device, image_shape = d2l.try_gpu(), (150, 225) # PIL Image (h, w)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.01, 500, 200)
```



Como você pode ver, a imagem composta retém o cenário e os objetos da imagem de conteúdo, enquanto introduz a cor da imagem de estilo. Como a imagem é relativamente pequena, os detalhes são um pouco confusos.

Para obter uma imagem composta mais clara, treinamos o modelo usando um tamanho de imagem maior: 900×600. Aumentamos a altura e a largura da imagem usada antes por um fator de quatro e inicializamos uma imagem composta maior.

```
image_shape = (600, 900) # PIL Image (h, w)
_, content_Y = get_contents(image_shape, device)
_, style_Y = get_styles(image_shape, device)
X = preprocess(postprocess(output), image_shape).to(device)
output = train(X, content_Y, style_Y, device, 0.01, 300, 100)
d2l.plt.imsave('../img/neural-style.jpg', postprocess(output))
```



Como você pode ver, cada época leva mais tempo devido ao tamanho maior da imagem. Conforme mostrado em Fig. 13.12.3, a imagem composta produzida retém mais detalhes devido ao seu tamanho maior. A imagem composta não só tem grandes blocos de cores como a imagem de estilo, mas esses blocos têm até a textura sutil de pinceladas.

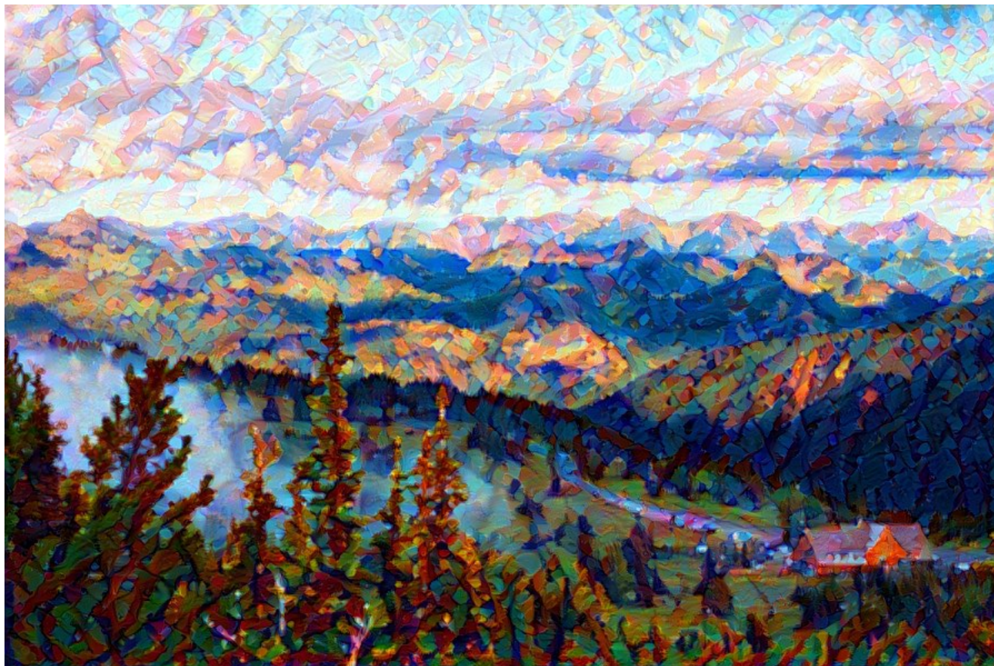


Fig. 13.12.3: Imagem 900×600 composta.

13.12.8 Resumo

- As funções de perda usadas na transferência de estilo geralmente têm três partes: 1. A perda de conteúdo é usada para fazer a imagem composta se aproximar da imagem de conteúdo no que diz respeito aos recursos de conteúdo. 2. A perda de estilo é usada para fazer com que a imagem composta se aproxime da imagem de estilo em termos de recursos de estilo. 3. A perda total de variação ajuda a reduzir o ruído na imagem composta.
- Podemos usar um CNN pré-treinado para extrair recursos de imagem e minimizar a função de perda para atualizar continuamente a imagem composta.
- Usamos uma matriz de Gram para representar a saída do estilo pelas camadas de estilo.

13.12.9 Exercícios

1. Como a saída muda quando você seleciona diferentes camadas de conteúdo e estilo?
2. Ajuste os hiperparâmetros de peso na função de perda. A saída retém mais conteúdo ou tem menos ruído?
3. Use imagens de conteúdo e estilo diferentes. Você pode criar imagens compostas mais interessantes?
4. Podemos aplicar transferência de estilo para texto? Dica: você pode consultar o documento de pesquisa de Hu et al. (Hu et al., 2020).

Discussões[169]

13.13 Classificação de Imagens (CIFAR-10) no Kaggle

Até agora, temos usado o pacote `data` do Gluon para obter diretamente conjuntos de dados de imagem no formato tensor. Na prática, entretanto, os conjuntos de dados de imagem geralmente existem no formato de arquivos de imagem. Nesta seção, começaremos com os arquivos de imagem originais e organizaremos, leremos e converteremos os arquivos para o formato tensor passo a passo.

Realizamos um experimento no conjunto de dados CIFAR-10 em Section 13.1. Este é um dado importante definido no campo de visão do computador. Agora, vamos aplicar o conhecimento que aprendemos em as seções anteriores para participar da competição Kaggle, que aborda problemas de classificação de imagens CIFAR-10. O endereço da competição na web é

https://www.kaggle.com/c/cifar-10

Fig. 13.13.1 mostra as informações na página da competição. Para enviar os resultados, primeiro registre uma conta no site do Kaggle.

[169] https://discuss.d2l.ai/t/1476

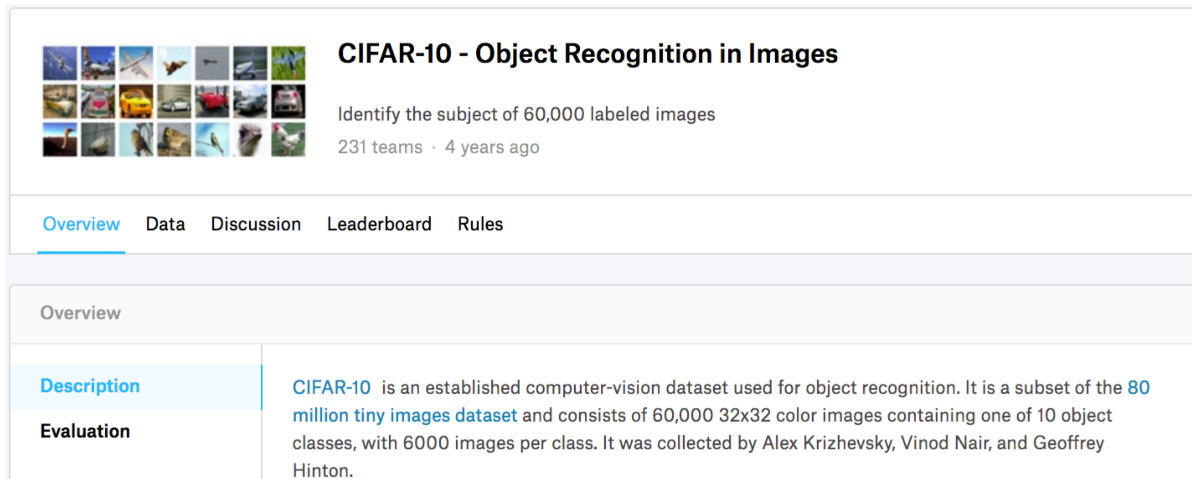


Fig. 13.13.1: Informações da página da web do concurso de classificação de imagens CIFAR-10. O conjunto de dados da competição pode ser acessado clicando na guia "Dados".

Primeiro, importe os pacotes ou módulos necessários para a competição.

```
import collections
import math
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.13.1 Obtendo e Organizando o *Dataset*

Os dados da competição são divididos em um conjunto de treinamento e um conjunto de teste. O conjunto de treinamento contém 50.000 imagens. O conjunto de teste contém 300.000 imagens, das quais 10.000 imagens são usadas para pontuação, enquanto as outras 290.000 imagens sem pontuação são incluídas para evitar a rotulagem manual do conjunto de teste e o envio dos resultados da rotulagem. Os formatos de imagem em ambos os conjuntos de dados são PNG, com alturas e larguras de 32 pixels e três canais de cores (RGB). As imagens cobrem categorias de 10: aviões, carros, pássaros, gatos, veados, cães, sapos, cavalos, barcos e caminhões. O canto superior esquerdo de Fig. 13.13.1 mostra algumas imagens de aviões, carros e pássaros no conjunto de dados.

Baixando o Dataset

Após fazer o login no Kaggle, podemos clicar na guia "Dados" na página da competição de classificação de imagens CIFAR-10 mostrada em Fig. 13.13.1 e baixar o conjunto de dados clicando no botão "*Download All*". Após descompactar o arquivo baixado em `../data` e descompactar `train.7z` e `test.7z` dentro dele, você encontrará o conjunto de dados inteiro nos seguintes caminhos:

- ../data/cifar-10/train/[1-50000].png
- ../data/cifar-10/test/[1-300000].png
- ../data/cifar-10/trainLabels.csv
- ../data/cifar-10/sampleSubmission.csv

Aqui, as pastas `train` e`test` contêm as imagens de treinamento e teste, respectivamente, `trainLabels.csv` tem rótulos para as imagens de treinamento e `sample_submission.csv` é um exemplo de envio.

Para facilitar o início, fornecemos uma amostra em pequena escala do conjunto de dados: ele contém as primeiras 1000 de imagens de treinamento e 5 de imagens de teste aleatórias. Para usar o conjunto de dados completo da competição Kaggle, você precisa definir a seguinte variável demo como `False`.

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
Downloading ../data/kaggle_cifar10_tiny.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪kaggle_cifar10_tiny.zip...
```

Organizando o Dataset

Precisamos organizar conjuntos de dados para facilitar o treinamento e teste do modelo. Vamos primeiro ler os rótulos do arquivo csv. A função a seguir retorna um dicionário que mapeia o nome do arquivo sem extensão para seu rótulo.

```
#@save
def read_csv_labels(fname):
    """Read fname to return a name to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))
```

(continues on next page)

(continued from previous page)

```
labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
# training examples: 1000
# classes: 10
```

Em seguida, definimos a função `reorg_train_valid` para segmentar o conjunto de validação do conjunto de treinamento original. O argumento `valid_ratio` nesta função é a razão entre o número de exemplos no conjunto de validação e o número de exemplos no conjunto de treinamento original. Em particular, seja n o número de imagens da classe com menos exemplos e r a proporção, então usaremos $\max(\lfloor nr\rfloor,1)$ imagens para cada classe como o conjunto de validação. Vamos usar `valid_ratio = 0.1` como exemplo. Como o conjunto de treinamento original tem 50.000 imagens, haverá 45.000 imagens usadas para treinamento e armazenadas no caminho "`train_valid_test/train`" ao ajustar hiperparâmetros, enquanto as outras 5.000 imagens serão armazenadas como conjunto de validação no caminho "`train_valid_test / valid`". Depois de organizar os dados, as imagens da mesma turma serão colocadas na mesma pasta para que possamos lê-las posteriormente.

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    # The number of examples of the class with the least examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        # Copy to train_valid_test/train_valid with a subfolder per class
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            # Copy to train_valid_test/valid
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            # Copy to train_valid_test/train
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

A função `reorg_test` abaixo é usada para organizar o conjunto de testes para facilitar a leitura durante a previsão.

```
#@save
def reorg_test(data_dir):
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

Finalmente, usamos uma função para chamar as funções `read_csv_labels`, `reorg_train_valid` e `reorg_test` previamente definidas.

```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```

Definimos apenas o tamanho do lote em 4 para o conjunto de dados de demonstração. Durante o treinamento e os testes reais, o conjunto de dados completo da competição Kaggle deve ser usado e `batch_size` deve ser definido como um número inteiro maior, como 128. Usamos 10% dos exemplos de treinamento como o conjunto de validação para ajustar os hiperparâmetros.

```
batch_size = 4 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

13.13.2 Aumento de Imagem

Para lidar com o *overfitting*, usamos o aumento da imagem. Por exemplo, adicionando `transforms.RandomFlipLeftRight()`, as imagens podem ser invertidas aleatoriamente. Também podemos realizar a normalização para os três canais RGB de imagens coloridas usando `transforma.Normalize()`. Abaixo, listamos algumas dessas operações que você pode escolher para usar ou modificar, dependendo dos requisitos.

```
transform_train = torchvision.transforms.Compose([
    # Magnify the image to a square of 40 pixels in both height and width
    torchvision.transforms.Resize(40),
    # Randomly crop a square image of 40 pixels in both height and width to
    # produce a small square of 0.64 to 1 times the area of the original
    # image, and then shrink it to a square of 32 pixels in both height and
    # width
    torchvision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                              ratio=(1.0, 1.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    # Normalize each channel of the image
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

Para garantir a certeza da saída durante o teste, realizamos apenas normalização na imagem.

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

13.13.3 Lendo o *Dataset*

Em seguida, podemos criar a instância `ImageFolderDataset` para ler o conjunto de dados organizado contendo os arquivos de imagem originais, onde cada exemplo inclui a imagem e o rótulo.

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

Especificamos a operação de aumento de imagem definida em `DataLoader`. Durante o treinamento, usamos apenas o conjunto de validação para avaliar o modelo, portanto, precisamos garantir a certeza do resultado. Durante a previsão, treinaremos o modelo no conjunto de treinamento combinado e no conjunto de validação para fazer uso completo de todos os dados rotulados.

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

13.13.4 Definindo o Modelo

Aqui, construímos os blocos residuais com base na classe `HybridBlock`, que é ligeiramente diferente da implementação descrita em Section 7.6. Isso é feito para melhorar a eficiência de execução.

A seguir, definimos o modelo ResNet-18.

O desafio de classificação de imagens CIFAR-10 usa 10 categorias. Vamos realizar a inicialização aleatória de Xavier no modelo antes do início do treinamento.

```
def get_net():
    num_classes = 10
    # PyTorch doesn't have the notion of hybrid model
    net = d2l.resnet18(num_classes, 3)
    return net

loss = nn.CrossEntropyLoss(reduction="none")
```

13.13.5 Definindo as Funções de Treinamento

Selecionaremos o modelo e ajustaremos os hiperparâmetros de acordo com o desempenho do modelo no conjunto de validação. Em seguida, definimos a função de treinamento do modelo `treinar`. Registramos o tempo de treinamento de cada época, o que nos ajuda a comparar os custos de tempo de diferentes modelos.

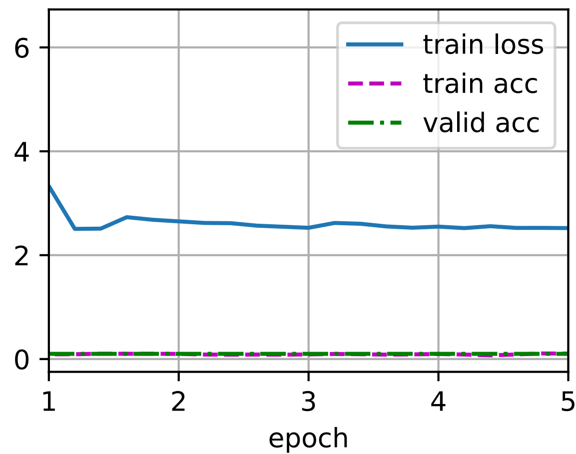
```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9,
                              weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'valid acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        net.train()
        metric = d2l.Accumulator(3)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(net, features, labels,
                                          loss, trainer, devices)
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                              None))
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpu(net, valid_iter)
            animator.add(epoch + 1, (None, None, valid_acc))
        scheduler.step()
    if valid_iter is not None:
        print(f'loss {metric[0] / metric[2]:.3f}, '
              f'train acc {metric[1] / metric[2]:.3f}, '
              f'valid acc {valid_acc:.3f}')
    else:
        print(f'loss {metric[0] / metric[2]:.3f}, '
              f'train acc {metric[1] / metric[2]:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)}')
```

13.13.6 Treinamento e Validação do Modelo

Agora podemos treinar e validar o modelo. Os hiperparâmetros a seguir podem ser ajustados. Por exemplo, podemos aumentar o número de épocas. Como `lr_period` e `lr_decay` são definidos como 50 e 0,1 respectivamente, a taxa de aprendizado do algoritmo de otimização será multiplicada por 0,1 a cada 50 épocas. Para simplificar, treinamos apenas uma época aqui.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 5, 0.1, 5e-4
lr_period, lr_decay, net = 50, 0.1, get_net()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
loss 2.522, train acc 0.105, valid acc 0.100
116.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```

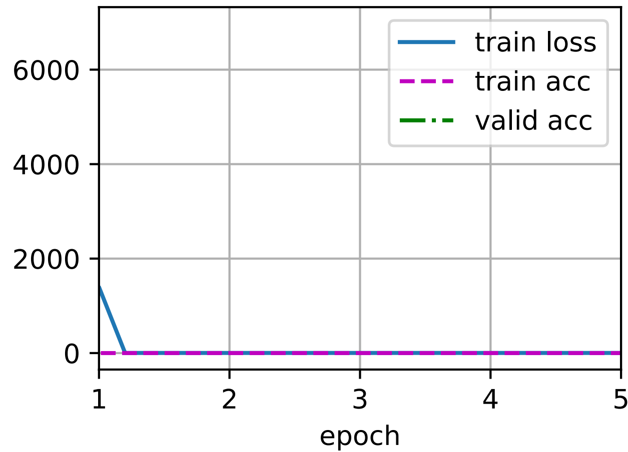


13.13.7 Classificando o Conjunto de Testes e Enviando Resultados no Kaggle

Depois de obter um design de modelo satisfatório e hiperparâmetros, usamos todos os conjuntos de dados de treinamento (incluindo conjuntos de validação) para treinar novamente o modelo e classificar o conjunto de teste.

```
net, preds = get_net(), []
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.to(devices[0]))
    preds.extend(y_hat.argmax(dim=1).type(torch.int32).cpu().numpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.classes[x])
df.to_csv('submission.csv', index=False)
```

```
loss 2.501, train acc 0.097
133.7 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Após executar o código acima, obteremos um arquivo "submit.csv". O formato deste arquivo é consistente com os requisitos da competição Kaggle. O método para enviar resultados é semelhante ao método em Section 4.10.

13.13.8 Resumo

- Podemos criar uma instância `ImageFolderDataset` para ler o conjunto de dados contendo os arquivos de imagem originais.
- Podemos usar redes neurais convolucionais, aumento de imagens e programação híbrida para participar de uma competição de classificação de imagens.

13.13.9 Exercícios

1. Use o conjunto de dados CIFAR-10 completo para a competição Kaggle. Altere o `batch_size` e o número de épocas `num_epochs` para 128 e 100, respectivamente. Veja qual precisão e classificação você pode alcançar nesta competição.
2. Que precisão você pode alcançar quando não está usando o aumento de imagem?
3. Digitalize o código QR para acessar as discussões relevantes e trocar ideias sobre os métodos usados e os resultados obtidos com a comunidade. Você pode sugerir técnicas melhores?

Discussões[170]

13.14 Identificação de Raça de Cachorro (*ImageNet Dogs*) no Kaggle

Nesta seção, abordaremos o desafio da identificação de raças de cães na Competição Kaggle. O endereço da competição na web é

https://www.kaggle.com/c/dog-breed-identification

[170] https://discuss.d2l.ai/t/1479

Nesta competição, tentamos identificar 120 raças diferentes de cães. O conjunto de dados usado nesta competição é, na verdade, um subconjunto do famoso conjunto de dados ImageNet. Diferente das imagens no conjunto de dados CIFAR-10 usado na seção anterior, as imagens no conjunto de dados ImageNet são mais altas e mais largas e suas dimensões são inconsistentes.

Fig. 13.14.1 mostra as informações na página da competição. Para enviar os resultados, primeiro registre uma conta no site do Kaggle.

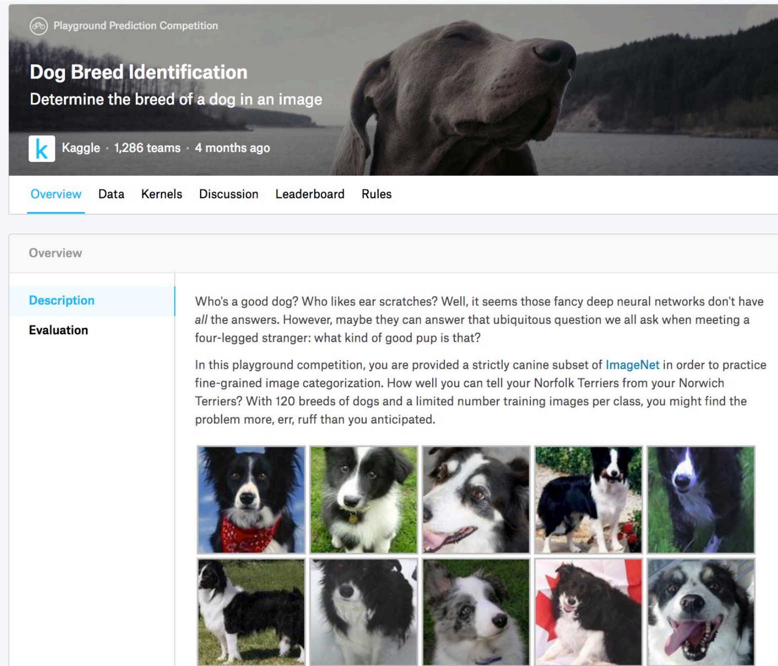


Fig. 13.14.1: Site de competição de identificação de raças de cães. O conjunto de dados da competição pode ser acessado clicando na guia "Data".

Primeiro, importe os pacotes ou módulos necessários para a competição.

```
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.14.1 Obtenção e organização do *Dataset*

Os dados da competição são divididos em um conjunto de treinamento e um conjunto de teste. O conjunto de treinamento contém 10.222 imagens e o conjunto de teste contém 10.357 imagens. As imagens em ambos os conjuntos estão no formato JPEG. Essas imagens contêm três canais RGB (cores) e diferentes alturas e larguras. Existem 120 raças de cães no conjunto de treinamento, incluindo Labradores, Poodles, Dachshunds, Samoyeds, Huskies, Chihuahuas e Yorkshire Terriers.

Baixando o Dataset

Depois de fazer o login no Kaggle, podemos clicar na guia "Data" na página da competição de identificação de raças de cães mostrada em Fig. 13.14.1 e baixar o conjunto de dados clicando no botão" Baixar tudo ". Após descompactar o arquivo baixado em `../data`, você encontrará todo o conjunto de dados nos seguintes caminhos:

- ../data/dog-breed-identification/labels.csv
- ../data/dog-breed-identification/sample_submission.csv
- ../data/dog-breed-identification/train
- ../data/dog-breed-identification/test

Você deve ter notado que a estrutura acima é bastante semelhante à da competição CIFAR-10 em Section 13.13, onde as pastas `train/` e `test/` contêm imagens de treinamento e teste de cães, respectivamente, e `rótulos. csv` tem os rótulos das imagens de treinamento.

Da mesma forma, para facilitar o início, fornecemos uma amostra em pequena escala do conjunto de dados mencionado acima, "train_valid_test_tiny.zip". Se você for usar o conjunto de dados completo para a competição Kaggle, você também precisará alterar a variável demo abaixo para `False`.

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to False
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

Organizando o Dataset

Podemos organizar o conjunto de dados de forma semelhante ao que fizemos em Section 13.13, nomeadamente separando um conjunto de validação do conjunto de treinamento e movendo imagens em subpastas agrupadas por rótulos.

A função `reorg_dog_data` abaixo é usada para ler os rótulos dos dados de treinamento, segmentar o conjunto de validação e organizar o conjunto de treinamento.

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 4 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

13.14.2 Aumento de Imagem

O tamanho das imagens nesta seção é maior do que as imagens na seção anterior. Aqui estão mais algumas operações de aumento de imagem que podem ser úteis.

```
transform_train = torchvision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height to width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new image with a height and width of 224
    # pixels each
    torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                             ratio=(3.0/4.0, 4.0/3.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    # Randomly change the brightness, contrast, and saturation
    torchvision.transforms.ColorJitter(brightness=0.4,
                                       contrast=0.4,
                                       saturation=0.4),
    # Add random noise
    torchvision.transforms.ToTensor(),
    # Standardize each channel of the image
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

Durante o teste, usamos apenas operações de pré-processamento de imagens definidas.

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    # Crop a square of 224 by 224 from the center of the image
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

13.14.3 Lendo o Dataset

Como na seção anterior, podemos criar uma instância `ImageFolderDataset` para ler o conjunto de dados contendo os arquivos de imagem originais.

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

Aqui, criamos instâncias de `DataLoader`, assim como em Section 13.13.

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
```

(continues on next page)

(continued from previous page)

```
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

13.14.4 Definindo o Modelo

O conjunto de dados para esta competição é um subconjunto dos dados ImageNet. Portanto, podemos usar a abordagem discutida em Section 13.2 para selecionar um modelo pré-treinado em todo o conjunto de dados ImageNet e usá-lo para extrair recursos de imagem a serem inseridos na rede de saída de pequena escala personalizada. A Gluon oferece uma ampla gama de modelos pré-treinados. Aqui, usaremos o modelo ResNet-34 pré-treinado. Como o conjunto de dados da competição é um subconjunto do conjunto de dados de pré-treinamento, simplesmente reutilizamos a entrada da camada de saída do modelo pré-treinado, ou seja, os recursos extraídos. Então, podemos substituir a camada de saída original por uma pequena camada personalizada de rede de saída que pode ser treinada, como duas camadas totalmente conectadas em uma série. Diferente da experiência em Section 13.2, aqui, não retreinamos o modelo pré-treinado usado para extração de características. Isso reduz o tempo de treinamento e a memória necessária para armazenar gradientes de parâmetro do modelo.

Você deve notar que, durante o aumento da imagem, usamos os valores médios e desvios padrão dos três canais RGB para todo o conjunto de dados ImageNet para normalização. Isso é consistente com a normalização do modelo pré-treinado.

```
def get_net(devices):
    finetune_net = nn.Sequential()
    finetune_net.features = torchvision.models.resnet34(pretrained=True)
    # Define a new output network
    # There are 120 output categories
    finetune_net.output_new = nn.Sequential(nn.Linear(1000, 256),
                                            nn.ReLU(),
                                            nn.Linear(256, 120))
    # Move model to device
    finetune_net = finetune_net.to(devices[0])
    # Freeze feature layer params
    for param in finetune_net.features.parameters():
        param.requires_grad = False
    return finetune_net
```

Ao calcular a perda, primeiro usamos a variável-membro `features` para obter a entrada da camada de saída do modelo pré-treinado, ou seja, a característica extraída. Em seguida, usamos essa característica como a entrada para nossa pequena rede de saída personalizada e calculamos a saída.

```
loss = nn.CrossEntropyLoss(reduction='none')

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        features, labels = features.to(devices[0]), labels.to(devices[0])
```

(continues on next page)

(continued from previous page)

```
        outputs = net(features)
        l = loss(outputs, labels)
        l_sum = l.sum()
        n += labels.numel()
    return l_sum / n
```

13.14.5 Definindo as Funções de Treinamento

Selecionaremos o modelo e ajustaremos os hiperparâmetros de acordo com o desempenho do modelo no conjunto de validação. A função de treinamento do modelo `treinar` apenas treina a pequena rede de saída personalizada.

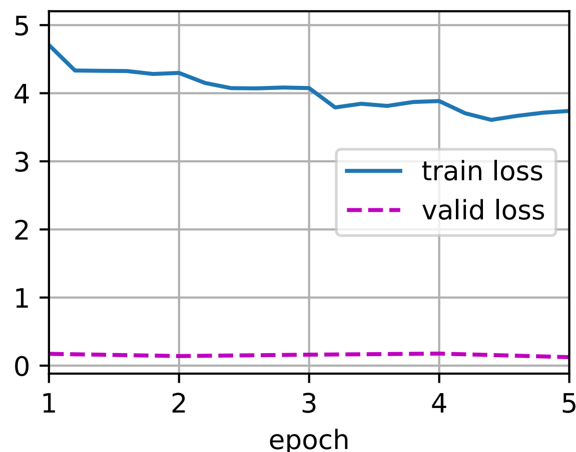
```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Only train the small custom output network
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.SGD((param for param in net.parameters()
                               if param.requires_grad), lr=lr,
                              momentum=0.9, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'valid loss'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            features, labels = features.to(devices[0]), labels.to(devices[0])
            trainer.zero_grad()
            output = net(features)
            l = loss(output, labels).sum()
            l.backward()
            trainer.step()
            metric.add(l, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1], None))
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss))
        scheduler.step()
    if valid_iter is not None:
        print(f'train loss {metric[0] / metric[1]:.3f}, '
              f'valid loss {valid_loss:.3f}')
    else:
        print(f'train loss {metric[0] / metric[1]:.3f}')
    print(f'{metric[1] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)}')
```

13.14.6 Treinamento e Validação do Modelo

Agora podemos treinar e validar o modelo. Os hiperparâmetros a seguir podem ser ajustados. Por exemplo, podemos aumentar o número de épocas. Como `lr_period` e `lr_decay` são definidos como 10 e 0,1 respectivamente, a taxa de aprendizado do algoritmo de otimização será multiplicada por 0,1 a cada 10 épocas.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 5, 0.001, 1e-4
lr_period, lr_decay, net = 10, 0.1, get_net(devices)
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 3.740, valid loss 0.125
107.6 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```

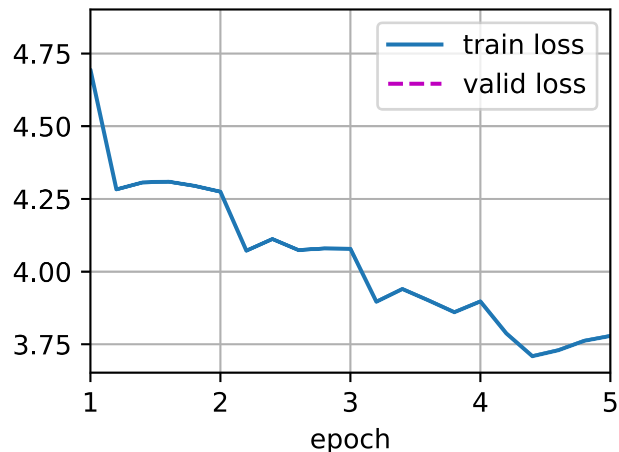


13.14.7 Classificando o Conjunto de Testes e Enviando Resultados no Kaggle

Depois de obter um design de modelo satisfatório e hiperparâmetros, usamos todos os conjuntos de dados de treinamento (incluindo conjuntos de validação) para treinar novamente o modelo e, em seguida, classificar o conjunto de teste. Observe que as previsões são feitas pela rede de saída que acabamos de treinar.

```
net = get_net(devices)
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output = torch.nn.functional.softmax(net(data.to(devices[0])), dim=0)
    preds.extend(output.cpu().detach().numpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.classes) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 3.779
126.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Após executar o código acima, geraremos um arquivo "submit.csv". O formato deste arquivo é consistente com os requisitos da competição Kaggle. O método para enviar resultados é semelhante ao método em Section 4.10.

13.14.8 Resumo

- Podemos usar um modelo pré-treinado no conjunto de dados ImageNet para extrair recursos e treinar apenas uma pequena rede de saída personalizada. Isso nos permitirá classificar um subconjunto do conjunto de dados ImageNet com menor sobrecarga de computação e armazenamento.

13.14.9 Exercícios

1. Ao usar todo o conjunto de dados Kaggle, que tipo de resultados você obtém ao aumentar `batch_size` (tamanho do lote) `enum_epochs` (número de épocas)?
2. Você obtém melhores resultados se usar um modelo pré-treinado mais profundo?
3. Digitalize o código QR para acessar as discussões relevantes e trocar ideias sobre os métodos usados e os resultados obtidos com a comunidade. Você pode sugerir técnicas melhores?

Discussões[171]

[171] https://discuss.d2l.ai/t/1481

14 Processamento de linguagem natural: Pré-treinamento

Os humanos precisam se comunicar. A partir dessa necessidade básica da condição humana, uma vasta quantidade de texto escrito tem sido gerada diariamente. Dado o texto rico em mídia social, aplicativos de chat, e-mails, análises de produtos, artigos de notícias, artigos de pesquisa e livros, torna-se vital permitir que os computadores os entendam para oferecer assistência ou tomar decisões com base em linguagens humanas.

O processamento de linguagem natural estuda as interações entre computadores e humanos usando linguagens naturais. Na prática, é muito comum usar técnicas de processamento de linguagem natural para processar e analisar dados de texto (linguagem natural humana), como modelos de linguagem em Section 8.3 e modelos de tradução automática em Section 9.5.

Para entender o texto, podemos começar com sua representação, como tratar cada palavra ou subpalavra como um token de texto individual. Como veremos neste capítulo, a representação de cada token pode ser pré-treinada em um grande corpus, usando word2vec, GloVe ou modelos de incorporação de subpalavra. Após o pré-treinamento, a representação de cada token pode ser um vetor, no entanto, permanece o mesmo, independentemente do contexto. Por exemplo, a representação vetorial de "banco" é a mesma em ambos "vá ao banco para depositar algum dinheiro" e "vá ao banco para se sentar". Assim, muitos modelos de pré-treinamento mais recentes adaptam a representação do mesmo token para contextos diferentes. Entre eles está o BERT, um modelo muito mais profundo baseado no codificador do transformador. Neste capítulo, vamos nos concentrar em como pré-treinar tais representações para texto, como destacado em Fig. 14.1.

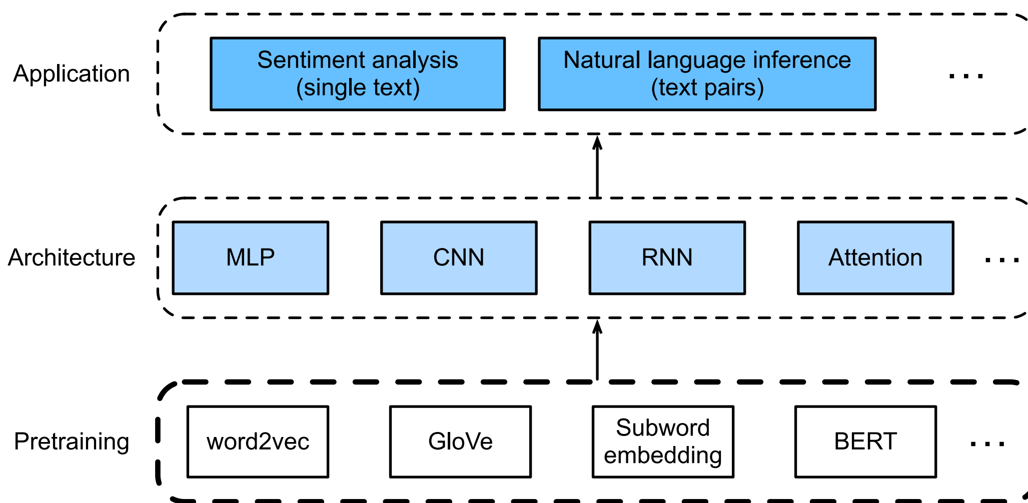


Fig. 14.1: As representações de texto pré-treinadas podem ser alimentadas para várias arquiteturas de aprendizado profundo para diferentes aplicativos de processamento de linguagem natural downstream. Este capítulo enfoca o pré-treinamento de representação de texto upstream.

Conforme mostrado em Fig. 14.1, as representações de texto pré-treinadas podem ser alimentadas para uma variedade de arquiteturas de aprendizado profundo para diferentes aplicativos de processamento de linguagem natural downstream. Iremos cobri-los em Chapter 15.

14.1 Incorporação de Palavras (word2vec)

Uma linguagem natural é um sistema complexo que usamos para expressar significados. Nesse sistema, as palavras são a unidade básica do significado linguístico. Como o próprio nome indica, um vetor de palavras é um vetor usado para representar uma palavra. Também pode ser considerado o vetor de características de uma palavra. A técnica de mapear palavras em vetores de números reais também é conhecida como incorporação de palavras. Nos últimos anos, a incorporação de palavras tornou-se gradualmente um conhecimento básico no processamento de linguagem natural.

14.1.1 Por que não usar vetores one-hot?

Usamos vetores one-hot para representar palavras (caracteres são palavras) em Section 8.5. Lembre-se de que quando assumimos o número de palavras diferentes em um dicionário (o tamanho do dicionário) é N, cada palavra pode corresponder uma a uma com inteiros consecutivos de 0 a $N-1$. Esses inteiros que correspondem a as palavras são chamadas de índices das palavras. Assumimos que o índice de uma palavra é i. A fim de obter a representação vetorial one-hot da palavra, criamos um vetor de 0s com comprimento de N e defina o elemento i como 1. Desta forma, cada palavra é representada como um vetor de comprimento N que pode ser usado diretamente por a rede neural.

Embora os vetores de uma palavra quente sejam fáceis de construir, eles geralmente não são uma boa escolha. Uma das principais razões é que os vetores de uma palavra quente não podem expressar com precisão a semelhança entre palavras diferentes, como a semelhança de cosseno que usamos comumente. Para os vetores $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, suas semelhanças de cosseno são os cossenos dos

ângulos entre eles:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \tag{14.1.1}$$

Uma vez que a similaridade de cosseno entre os vetores one-hot de quaisquer duas palavras diferentes é 0, é difícil usar o vetor one-hot para representar com precisão a similaridade entre várias palavras diferentes.

Word2vec[172] é uma ferramenta que viemos para resolver o problema acima. Ele representa cada palavra com um vetor de comprimento fixo e usa esses vetores para melhor indicar a similaridade e relações de analogia entre palavras diferentes. A ferramenta Word2vec contém dois modelos: skip-gram (Mikolov et al., 2013b) e bolsa contínua de words (CBOW) (Mikolov et al., 2013a). Em seguida, vamos dar um observe os dois modelos e seus métodos de treinamento.

14.1.2 O Modelo Skip-Gram

O modelo skip-gram assume que uma palavra pode ser usada para gerar as palavras que a cercam em uma sequência de texto. Por exemplo, assumimos que a sequência de texto é "o", "homem", "ama", "seu" e "filho". Usamos "amores" como palavra-alvo central e definimos o tamanho da janela de contexto para 2. Conforme mostrado em Fig. 14.1.1, dada a palavra-alvo central "amores", o modelo de grama de salto está preocupado com a probabilidade condicional para gerando as palavras de contexto, "o", "homem", "seu" e "filho", que estão a uma distância de no máximo 2 palavras, que é

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}). \tag{14.1.2}$$

Assumimos que, dada a palavra-alvo central, as palavras de contexto são geradas independentemente umas das outras. Neste caso, a fórmula acima pode ser reescrita como

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}). \tag{14.1.3}$$

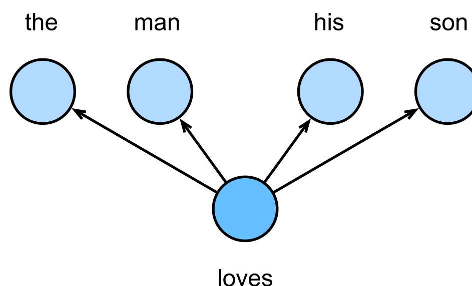


Fig. 14.1.1: O modelo skip-gram se preocupa com a probabilidade condicional de gerar palavras de contexto para uma determinada palavra-alvo central.

No modelo skip-gram, cada palavra é representada como dois vetores de dimensão d, que são usados para calcular a probabilidade condicional. Assumimos que a palavra está indexada como i no dicionário, seu vetor é representado como $\mathbf{v}_i \in \mathbb{R}^d$ quando é a palavra alvo central, e $\mathbf{u}_i \in \mathbb{R}^d$ quando é uma palavra de contexto. Deixe a palavra alvo central w_c e a palavra de contexto

[172] https://code.google.com/archive/p/word2vec/

w_o serem indexadas como c e o respectivamente no dicionário. A probabilidade condicional de gerar a palavra de contexto para a palavra alvo central fornecida pode ser obtida executando uma operação softmax no produto interno do vetor:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{14.1.4}$$

onde o índice de vocabulário definido $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$. Suponha que uma sequência de texto de comprimento T seja fornecida, onde a palavra no passo de tempo t é denotada como $w^{(t)}$. Suponha que as palavras de contexto sejam geradas independentemente, dadas as palavras centrais. Quando o tamanho da janela de contexto é m, a função de verossimilhança do modelo skip-gram é a probabilidade conjunta de gerar todas as palavras de contexto dadas qualquer palavra central

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.1.5}$$

Aqui, qualquer intervalo de tempo menor que 1 ou maior que T pode ser ignorado.

Treinamento do modelo Skip-Gram

Os parâmetros do modelo skip-gram são o vetor da palavra alvo central e o vetor da palavra de contexto para cada palavra individual. No processo de treinamento, aprenderemos os parâmetros do modelo maximizando a função de verossimilhança, também conhecida como estimativa de máxima verossimilhança. Isso é equivalente a minimizar a seguinte função de perda:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m,\ j \neq 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{14.1.6}$$

Se usarmos o SGD, em cada iteração vamos escolher uma subsequência mais curta por meio de amostragem aleatória para calcular a perda para essa subsequência e, em seguida, calcular o gradiente para atualizar os parâmetros do modelo. A chave do cálculo de gradiente é calcular o gradiente da probabilidade condicional logarítmica para o vetor de palavras central e o vetor de palavras de contexto. Por definição, primeiro temos

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \tag{14.1.7}$$

Por meio da diferenciação, podemos obter o gradiente $\mathbf{v}_c$ da fórmula acima.

$$\begin{aligned}
\frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\
&= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\
&= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) \mathbf{u}_j.
\end{aligned} \tag{14.1.8}$$

Seu cálculo obtém a probabilidade condicional para todas as palavras no dicionário dada a palavra alvo central w_c. Em seguida, usamos o mesmo método para obter os gradientes para outros vetores de palavras.

Após o treinamento, para qualquer palavra do dicionário com índice i, vamos obter seus conjuntos de vetores de duas palavras $\mathbf{v}_i$ e $\mathbf{u}_i$. Em aplicações de processamento de linguagem natural, o vetor de palavra-alvo central no modelo skip-gram é geralmente usado como o vetor de representação de uma palavra.

14.1.3 O modelo do conjunto contínuo de palavras (CBOW)

O modelo de conjunto contínuo de palavras (CBOW) é semelhante ao modelo skip-gram. A maior diferença é que o modelo CBOW assume que a palavra-alvo central é gerada com base nas palavras do contexto antes e depois dela na sequência de texto. Com a mesma sequência de texto "o", "homem", "ama", "seu" e "filho", em que "ama" é a palavra alvo central, dado um tamanho de janela de contexto de 2, o modelo CBOW se preocupa com a probabilidade condicional de gerar a palavra de destino "ama" com base nas palavras de contexto "o", "homem", "seu" e "filho" (conforme mostrado em Fig. 14.1.2), como

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \tag{14.1.9}$$

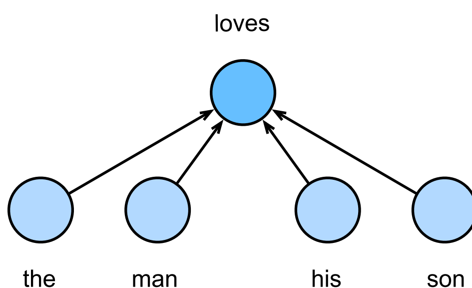


Fig. 14.1.2: O modelo CBOW se preocupa com a probabilidade condicional de gerar a palavra-alvo central a partir de determinadas palavras de contexto.

Como há várias palavras de contexto no modelo CBOW, calcularemos a média de seus vetores de palavras e usaremos o mesmo método do modelo skip-gram para calcular a probabilidade condicional. Assumimos que $\mathbf{v_i} \in \mathbb{R}^d$ e $\mathbf{u_i} \in \mathbb{R}^d$ são o vetor de palavra de contexto e vetor de palavra-alvo central da palavra com index i no dicionário (observe que os símbolos são opostos aos do modelo skip-gram). Deixe a palavra alvo central w_c ser indexada como c, e as palavras de contexto $w_{o_1}, \ldots, w_{o_{2m}}$ sejam indexadas como $o_1, \ldots, o_{2m}$ no dicionário. Assim, a probabilidade condicional de gerar uma palavra-alvo central a partir da palavra de contexto fornecida é

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\text{exp}\left(\frac{1}{2m}\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots, + \mathbf{v}_{o_{2m}}) \right)}{ \sum_{i \in \mathcal{V}} \text{exp}\left(\frac{1}{2m}\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots, + \mathbf{v}_{o_{2m}}) \right)}. \tag{14.1.10}$$

Para resumir, denote $\mathcal{W}_o= \{w_{o_1}, \ldots, w_{o_{2m}}\}$, e $\bar{\mathbf{v}}_o = \left(\mathbf{v}_{o_1} + \ldots, + \mathbf{v}_{o_{2m}} \right)/(2m)$. A equação acima pode ser simplificada como

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{14.1.11}$$

Dada uma sequência de texto de comprimento T, assumimos que a palavra no passo de tempo t é $w^{(t)}$, e o tamanho da janela de contexto é m. A função de verossimilhança do modelo CBOW é a probabilidade de gerar qualquer palavra-alvo central a partir das palavras de contexto.

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.12}$$

Treinamento de modelo CBOW

O treinamento do modelo CBOW é bastante semelhante ao treinamento do modelo skip-gram. A estimativa de máxima verossimilhança do modelo CBOW é equivalente a minimizar a função de perda.

$$-\sum_{t=1}^T \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.13}$$

Note que

$$\log\, P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log\,\left(\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)\right). \tag{14.1.14}$$

Por meio da diferenciação, podemos calcular o logaritmo da probabilidade condicional do gradiente de qualquer vetor de palavra de contexto $\mathbf{v}_{o_i}(i = 1, \ldots, 2m)$ na fórmula acima.

$$\frac{\partial \log\, P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)\mathbf{u}_j}{ \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j \right). \tag{14.1.15}$$

Em seguida, usamos o mesmo método para obter os gradientes para outros vetores de palavras. Ao contrário do modelo skip-gram, geralmente usamos o vetor de palavras de contexto como o vetor de representação de uma palavra no modelo CBOW.

14.1.4 Sumário

- Um vetor de palavras é um vetor usado para representar uma palavra. A técnica de mapear palavras em vetores de números reais também é conhecida como incorporação de palavras.
- Word2vec inclui o saco contínuo de palavras (CBOW) e modelos de grama de salto. O modelo skip-gram assume que as palavras de contexto são geradas com base na palavra-alvo central. O modelo CBOW assume que a palavra-alvo central é gerada com base nas palavras do contexto.

14.1.5 Exercícios

1. Qual é a complexidade computacional de cada gradiente? Se o dicionário contiver um grande volume de palavras, que problemas isso causará?
2. Existem algumas frases fixas no idioma inglês que consistem em várias palavras, como "nova york". Como você pode treinar seus vetores de palavras? Dica: Veja a seção 4 do artigo Word2vec (Mikolov et al., 2013b).
3. Use o modelo skip-gram como um exemplo para pensar sobre o design de um modelo word2vec. Qual é a relação entre o produto interno de dois vetores de palavras e a semelhança de cosseno no modelo de grama de salto? Para um par de palavras com significado semântico próximo, por que é provável que sua similaridade de cosseno de vetor de palavras seja alta?

Discussão[173]

[173] https://discuss.d2l.ai/t/381

14.2 Treinamento Aproximado

Lembre-se do conteúdo da última seção. O principal recurso do modelo skip-gram é o uso de operações softmax para calcular a probabilidade condicional de gerar a palavra de contexto w_o com base na palavra-alvo central fornecida w_c.

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}. \tag{14.2.1}$$

A perda logarítmica correspondente à probabilidade condicional é dada como

$$-\log P(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \tag{14.2.2}$$

Como a operação softmax considerou que a palavra de contexto poderia ser qualquer palavra no dicionário $\mathcal{V}$, a perda mencionada acima na verdade inclui a soma do número de itens no tamanho do dicionário. Na última seção, sabemos que para o modelo skip-gram e o modelo CBOW, porque ambos obtêm a probabilidade condicional usando uma operação softmax, o cálculo do gradiente para cada etapa contém a soma do número de itens no tamanho do dicionário. Para dicionários maiores com centenas de milhares ou até milhões de palavras, a sobrecarga para calcular cada gradiente pode ser muito alta. Para reduzir essa complexidade computacional, apresentaremos dois métodos de treinamento aproximados nesta seção: amostragem negativa e softmax hierárquico. Como não há grande diferença entre o modelo skip-gram e o modelo CBOW, usaremos apenas o modelo skip-gram como um exemplo para apresentar esses dois métodos de treinamento nesta seção.

14.2.1 Amostragem Negativa

A amostragem negativa modifica a função objetivo original. Dada uma janela de contexto para a palavra alvo central w_c, vamos tratá-la como um evento para a palavra de contexto w_o aparecer na janela de contexto e calcular a probabilidade deste evento a partir de

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \tag{14.2.3}$$

Aqui, a função σ tem a mesma definição que a função de ativação sigmóide:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{14.2.4}$$

Consideraremos primeiro o treinamento do vetor de palavras maximizando a probabilidade conjunta de todos os eventos na sequência de texto. Dada uma sequência de texto de comprimento T, assumimos que a palavra no passo de tempo t é $w^{(t)}$ e o tamanho da janela de contexto é m. Agora consideramos maximizar a probabilidade conjunta

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \tag{14.2.5}$$

No entanto, os eventos incluídos no modelo consideram apenas exemplos positivos. Nesse caso, apenas quando todos os vetores de palavras são iguais e seus valores se aproximam do infinito, a probabilidade conjunta acima pode ser maximizada para 1. Obviamente, esses vetores de palavras não têm sentido. A amostragem negativa torna a função objetivo mais significativa, amostrando

com uma adição de exemplos negativos. Assuma que o evento P ocorre quando a palavra de contexto w_o aparece na janela de contexto da palavra alvo central w_c, e nós amostramos K palavras que não aparecem na janela de contexto de acordo com a distribuição $P(w)$ para atuar como palavras de ruído. Assumimos que o evento para a palavra de ruído w_k ($k = 1, \ldots, K$) não aparecer na janela de contexto da palavra de destino central w_c é N_k. Suponha que os eventos P e $N_1, \ldots, N_K$ para os exemplos positivos e negativos sejam independentes um do outro. Ao considerar a amostragem negativa, podemos reescrever a probabilidade conjunta acima, que considera apenas os exemplos positivos, como

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.2.6}$$

Aqui, a probabilidade condicional é aproximada a ser

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim P(w)}^{K} P(D = 0 \mid w^{(t)}, w_k). \tag{14.2.7}$$

Seja o índice de sequência de texto da palavra $w^{(t)}$ no passo de tempo t i_t e h_k para a palavra de ruído w_k no dicionário. A perda logarítmica para a probabilidade condicional acima é

$$\begin{aligned}
-\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log P(D = 0 \mid w^{(t)}, w_k) \\
&= -\log \sigma \left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t} \right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \left(1 - \sigma \left(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t} \right) \right) \\
&= -\log \sigma \left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t} \right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma \left(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t} \right).
\end{aligned} \tag{14.2.8}$$

Aqui, o cálculo do gradiente em cada etapa do treinamento não está mais relacionado ao tamanho do dicionário, mas linearmente relacionado a K. Quando K assume uma constante menor, a amostragem negativa tem uma sobrecarga computacional menor para cada etapa.

14.2.2 Hierárquico Softmax

O softmax hierárquico é outro tipo de método de treinamento aproximado. Ele usa uma árvore binária para a estrutura de dados conforme ilustrado em Fig. 14.2.1, com os nós folha da árvore representando cada palavra no dicionário $\mathcal{V}$.

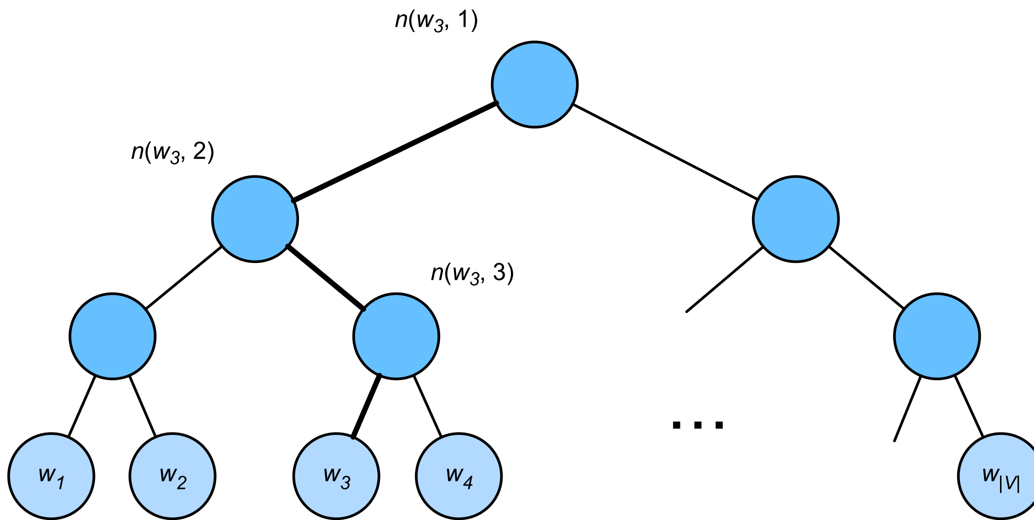


Fig. 14.2.1: Hierarchical Softmax. Each leaf node of the tree represents a word in the dictionary.

Assumimos que $L(w)$ é o número de nós no caminho (incluindo os nós raiz e folha) do nó raiz da árvore binária ao nó folha da palavra w. Seja $n(w, j)$ o nó j^{th} neste caminho, com o vetor de palavra de contexto $\mathbf{u}_{n(w,j)}$. Usamos Fig. 14.2.1 como exemplo, então $L(w_3) = 4$. O softmax hierárquico aproximará a probabilidade condicional no modelo skip-gram como

$$P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c \right), \tag{14.2.9}$$

Aqui, a função σ tem a mesma definição que a função de ativação sigmóide, e leftChild(n) é o nó filho esquerdo do nó n. Se x for verdadeiro, $[\![x]\!] = 1$; caso contrário, $[\![x]\!] = -1$. Agora, vamos calcular a probabilidade condicional de gerar a palavra w_3 com base na palavra w_c dada em Fig. 14.2.1. Precisamos encontrar o produto interno do vetor palavra $\mathbf{v}_c$ (para a palavra w_c) e cada vetor de nó não folha no caminho do nó raiz para w_3. Porque, na árvore binária, o caminho do nó raiz ao nó folha w_3 precisa ser percorrido para a esquerda, direita e esquerda novamente (o caminho com a linha em negrito em Fig. 14.2.1), obtemos

$$P(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c). \tag{14.2.10}$$

Porque $\sigma(x) + \sigma(-x) = 1$, a condição de que a soma da probabilidade condicional de qualquer palavra gerada com base na palavra alvo central fornecida w_c no dicionário $\mathcal{V}$ ser 1 também será suficiente:

$$\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1. \tag{14.2.11}$$

Além disso, porque a ordem de magnitude para $L(w_o) - 1$ é $\mathcal{O}(\log_2 |\mathcal{V}|)$, quando o tamanho do dicionário $\mathcal{V}$ for grande, a sobrecarga computacional para cada etapa do treinamento hierárquico softmax é muito reduzida em comparação com situações em que não usamos treinamento aproximado.

14.2.3 Sumário

- A amostragem negativa constrói a função de perda considerando eventos independentes que contêm exemplos positivos e negativos. O gradiente de sobrecarga computacional para cada etapa do processo de treinamento está linearmente relacionado ao número de palavras de ruído que amostramos.
- O softmax hierárquico usa uma árvore binária e constrói a função de perda com base no caminho do nó raiz ao nó folha. A sobrecarga computacional gradiente para cada etapa do processo de treinamento está relacionada ao logaritmo do tamanho do dicionário.

14.2.4 Exercícios

1. Antes de ler a próxima seção, pense em como devemos amostrar palavras de ruído na amostragem negativa.
2. O que faz com que a última fórmula desta seção seja válida?
3. Como podemos aplicar a amostragem negativa e softmax hierárquico no modelo skip-gram?

Discussão[174]

14.3 O conjunto de dados para incorporação de palavras com pré-treinamento

Nesta seção, apresentaremos como pré-processar um conjunto de dados com amostragem negativa Section 14.2 e carregar em minibatches para treinamento word2vec. O conjunto de dados que usamos é Penn Tree Bank (PTB)[175], que é um corpus pequeno, mas comumente usado. Ele pega amostras de artigos do Wall Street Journal e inclui conjuntos de treinamento, conjuntos de validação e conjuntos de teste.

Primeiro, importe os pacotes e módulos necessários para o experimento.

```
import math
import os
import random
import torch
from d2l import torch as d2l
```

[174] https://discuss.d2l.ai/t/382
[175] https://catalog.ldc.upenn.edu/LDC99T42

14.3.1 Leitura e pré-processamento do conjunto de dados

Este conjunto de dados já foi pré-processado. Cada linha do conjunto de dados atua como uma frase. Todas as palavras em uma frase são separadas por espaços. Na tarefa de incorporação de palavras, cada palavra é um token.

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    data_dir = d2l.download_extract('ptb')
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

```
Downloading ../data/ptb.zip from http://d2l-data.s3-accelerate.amazonaws.com/ptb.zip...
```

```
'# sentences: 42069'
```

Em seguida, construímos um vocabulário com palavras que aparecem no máximo 10 vezes mapeadas em um “<unk>” símbolo. Observe que os dados PTB pré-processados também contêm “<unk>” tokens apresentando palavras raras.

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

```
'vocab size: 6719'
```

14.3.2 Subamostragem

Em dados de texto, geralmente há algumas palavras que aparecem em altas frequências, como “the”, “a” e “in” em inglês. De modo geral, em uma janela de contexto, é melhor treinar o modelo de incorporação de palavras quando uma palavra (como “chip”) e uma palavra de frequência mais baixa (como “microprocessador”) aparecem ao mesmo tempo, em vez de quando um palavra aparece com uma palavra de frequência mais alta (como “o”). Portanto, ao treinar o modelo de incorporação de palavras, podemos realizar subamostragem nas palavras (Mikolov et al., 2013b). Especificamente, cada palavra indexada w_i no conjunto de dados desaparecerá com uma certa probabilidade. A probabilidade de abandono é dada como:

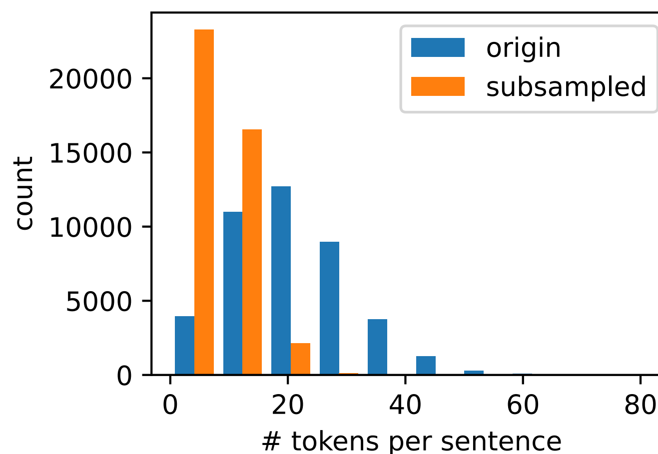
$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right), \tag{14.3.1}$$

Aqui, $f(w_i)$ é a proporção das instâncias da palavra w_i para o número total de palavras no conjunto de dados, e a constante t é um hiperparâmetro (definido como 10^{-4} neste experimentar). Como podemos ver, só é possível eliminar a palavra w_i na subamostragem quando $f(w_i) > t$. Quanto mais alta a frequência da palavra, maior sua probabilidade de abandono.

```
#@save
def subsampling(sentences, vocab):
    # Map low frequency words into <unk>
    sentences = [[vocab.idx_to_token[vocab[tk]] for tk in line]
                 for line in sentences]
    # Count the frequency for each word
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # Return True if to keep this token during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    # Now do the subsampling
    return [[tk for tk in line if keep(tk)] for line in sentences]

subsampled = subsampling(sentences, vocab)
```

Compare os comprimentos da sequência antes e depois da amostragem, podemos ver que a subamostragem reduziu significativamente o comprimento da sequência.

```
d2l.set_figsize()
d2l.plt.hist([[len(line) for line in sentences],
              [len(line) for line in subsampled]])
d2l.plt.xlabel('# tokens per sentence')
d2l.plt.ylabel('count')
d2l.plt.legend(['origin', 'subsampled']);
```



Para tokens individuais, a taxa de amostragem da palavra de alta frequência "the" é menor que 1/20.

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([line.count(token) for line in sentences])}, '
            f'after={sum([line.count(token) for line in subsampled])}')

compare_counts('the')
```

```
'# of "the": before=50770, after=2105'
```

Mas a palavra de baixa frequência "juntar" é completamente preservada.

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

Por último, mapeamos cada token em um índice para construir o corpus.

```
corpus = [vocab[line] for line in subsampled]
corpus[0:3]
```

```
[[0, 0], [392, 32, 2115, 5, 274, 406], [5277, 3054, 1580, 95]]
```

14.3.3 Carregando o conjunto de dados

Em seguida, lemos o corpus com índices de token em lotes de dados para treinamento.

Extração de palavras-alvo centrais e palavras de contexto

Usamos palavras com uma distância da palavra alvo central não excedendo o tamanho da janela de contexto como as palavras de contexto da palavra alvo central fornecida. A função de definição a seguir extrai todas as palavras-alvo centrais e suas palavras de contexto. Ele mostra de maneira uniforme e aleatória um inteiro para ser usado como o tamanho da janela de contexto entre o inteiro 1 e o max_window_size (janela de contexto máxima).

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    centers, contexts = [], []
    for line in corpus:
        # Each sentence needs at least 2 words to form a "central target word
        # - context word" pair
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at i
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the central target word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

A seguir, criamos um conjunto de dados artificial contendo duas sentenças de 7 e 3 palavras, respectivamente. Suponha que a janela de contexto máxima seja 2 e imprima todas as palavras-alvo centrais e suas palavras de contexto.

```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2, 3]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [2, 4]
center 4 has contexts [2, 3, 5, 6]
center 5 has contexts [3, 4, 6]
center 6 has contexts [4, 5]
center 7 has contexts [8, 9]
center 8 has contexts [7, 9]
center 9 has contexts [8]
```

Definimos o tamanho máximo da janela de contexto como 5. O seguinte extrai todas as palavras-alvo centrais e suas palavras de contexto no conjunto de dados.

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {len(all_centers)}'
```

```
'# center-context pairs: 352636'
```

Amostragem Negativa

Usamos amostragem negativa para treinamento aproximado. Para um par de palavras centrais e de contexto, amostramos aleatoriamente K palavras de ruído ($K = 5$ no experimento). De acordo com a sugestão do artigo Word2vec, a probabilidade de amostragem de palavras de ruído $P(w)$ é a razão entre a frequência de palavras de w e a frequência total de palavras elevada à potência de 0,75 [Mikolov.Sutskever. Chen.ea.2013].

Primeiro definimos uma classe para desenhar um candidato de acordo com os pesos amostrais. Ele armazena em cache um banco de números aleatórios de tamanho 10.000 em vez de chamar `random.choices` todas as vezes.

```
#@save
class RandomGenerator:
    """Draw a random int in [0, n] according to n sampling weights."""
    def __init__(self, sampling_weights):
        self.population = list(range(len(sampling_weights)))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
```

(continues on next page)

(continued from previous page)

```
        self.i += 1
        return self.candidates[self.i-1]

generator = RandomGenerator([2, 3, 4])
[generator.draw() for _ in range(10)]
```

```
[1, 0, 1, 2, 2, 0, 2, 2, 2, 2]
```

```
#@save
def get_negatives(all_contexts, corpus, K):
    counter = d2l.count_corpus(corpus)
    sampling_weights = [counter[i]**0.75 for i in range(len(counter))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, corpus, 5)
```

Lendo em lotes

Extraímos todas as palavras-alvo centrais `all_centers`, e as palavras de contexto `all_contexts` e palavras de ruído `all_negatives` de cada palavra-alvo central do conjunto de dados. Vamos lê-los em minibatches aleatórios.

Em um minibatch de dados, o exemplo i^{th} inclui uma palavra central e suas correspondentes n_i palavras de contexto e m_i palavras de ruído. Visto que o tamanho da janela de contexto de cada exemplo pode ser diferente, a soma das palavras de contexto e palavras de ruído, $n_i + m_i$, será diferente. Ao construir um minibatch, concatenamos as palavras de contexto e as palavras de ruído de cada exemplo e adicionamos 0s para preenchimento até que o comprimento das concatenações sejam iguais, ou seja, o comprimento de todas as concatenações é $\max_i n_i + m_i$ (`max_len`). Para evitar o efeito do preenchimento no cálculo da função de perda, construímos a variável de máscara `masks`, cada elemento correspondendo a um elemento na concatenação de palavras de contexto e ruído,`contexts_negatives`. Quando um elemento na variável `contexts_negatives` é um preenchimento, o elemento na variável de máscara `masks` na mesma posição será 0. Caso contrário, ele assume o valor 1. Para distinguir entre exemplos positivos e negativos, nós também precisa distinguir as palavras de contexto das palavras de ruído na variável `contexts_negatives`. Com base na construção da variável de máscara, só precisamos criar uma variável de rótulo `labels` com a mesma forma da variável `contexts_negatives` e definir os elementos correspondentes às palavras de contexto (exemplos positivos) para 1, e o resto para 0 .

A seguir, implementaremos a função de leitura de minibatch `batchify`. Sua entrada de minibatch, `data`, é uma lista cujo comprimento é o tamanho do lote, cada elemento contendo palavras-alvo centrais `center`, palavras de contexto `context` e palavras de ruído `negative`. Os dados de minibatch

retornados por esta função estão de acordo com o formato de que precisamos, por exemplo, inclui a variável de máscara.

```
#@save
def batchify(data):
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (torch.tensor(centers).reshape((-1, 1)), torch.tensor(contexts_negatives),
            torch.tensor(masks), torch.tensor(labels))
```

Construa dois exemplos simples:

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
centers = tensor([[1],
        [1]])
contexts_negatives = tensor([[2, 2, 3, 3, 3, 3],
        [2, 2, 2, 3, 3, 0]])
masks = tensor([[1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 0]])
labels = tensor([[1, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0]])
```

Usamos a função `batchify` definida apenas para especificar o método de leitura de minibatch na instância `DataLoader`.

14.3.4 Juntando todas as coisas

Por último, definimos a função `load_data_ptb` que lê o conjunto de dados PTB e retorna o iterador de dados.

```
#@save
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    num_workers = d2l.get_dataloader_workers()
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled = subsampling(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centers, all_contexts = get_centers_and_contexts(
        corpus, max_window_size)
    all_negatives = get_negatives(all_contexts, corpus, num_noise_words)
```

(continues on next page)

(continued from previous page)

```
    class PTBDataset(torch.utils.data.Dataset):
        def __init__(self, centers, contexts, negatives):
            assert len(centers) == len(contexts) == len(negatives)
            self.centers = centers
            self.contexts = contexts
            self.negatives = negatives

        def __getitem__(self, index):
            return (self.centers[index], self.contexts[index], self.negatives[index])

        def __len__(self):
            return len(self.centers)

    dataset = PTBDataset(
        all_centers, all_contexts, all_negatives)

    data_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True,
                                      collate_fn=batchify,
                                      num_workers=num_workers)
    return data_iter, vocab
```

Vamos imprimir o primeiro minibatch do iterador de dados.

```
data_iter, vocab = load_data_ptb(512, 5, 5)
for batch in data_iter:
    for name, data in zip(names, batch):
        print(name, 'shape:', data.shape)
    break
```

```
centers shape: torch.Size([512, 1])
contexts_negatives shape: torch.Size([512, 60])
masks shape: torch.Size([512, 60])
labels shape: torch.Size([512, 60])
```

14.3.5 Sumário

- A subamostragem tenta minimizar o impacto de palavras de alta frequência no treinamento de um modelo de incorporação de palavras.
- Podemos preencher exemplos de comprimentos diferentes para criar minibatches com exemplos de todos os mesmos comprimentos e usar variáveis de máscara para distinguir entre elementos de preenchimento e não preenchimento, de modo que apenas elementos não preenchidos participem do cálculo da função de perda.

14.3.6 Exercícios

1. Usamos a função `batchify` para especificar o método de leitura do minibatch na instância do `DataLoader` e imprimir a forma de cada variável na primeira leitura do lote. Como essas formas devem ser calculadas?

Discussão[176]

14.4 Pré-treinamento do word2vec

Nesta seção, treinaremos um modelo skip-gram definido em Section 14.1.

Primeiro, importe os pacotes e módulos necessários para o experimento e carregue o conjunto de dados PTB.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size,
                                     num_noise_words)
```

14.4.1 O Modelo Skip-Gram

Implementaremos o modelo skip-gram usando camadas de incorporação e multiplicação de minibatch. Esses métodos também são frequentemente usados para implementar outros aplicativos de processamento de linguagem natural.

Camada de incorporação

Conforme descrito em Section 9.7, A camada na qual a palavra obtida é incorporada é chamada de camada de incorporação, que pode ser obtida criando uma instância `nn.Embedding` em APIs de alto nível. O peso da camada de incorporação é uma matriz cujo número de linhas é o tamanho do dicionário (`input_dim`) e cujo número de colunas é a dimensão de cada vetor de palavra (`output_dim`). Definimos o tamanho do dicionário como 20 e a dimensão do vetor da palavra como 4.

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
print(f'Parameter embedding_weight ({embed.weight.shape}, '
      'dtype={embed.weight.dtype})')
```

```
Parameter embedding_weight (torch.Size([20, 4]), dtype={embed.weight.dtype})
```

A entrada da camada de incorporação é o índice da palavra. Quando inserimos o índice i de uma palavra, a camada de incorporação retorna a linha i^{th} da matriz de peso como seu vetor de palavra. Abaixo, inserimos um índice de forma $(2, 3)$ na camada de incorporação. Como a dimensão do vetor de palavras é 4, obtemos um vetor de palavras de forma $(2, 3, 4)$.

[176] https://discuss.d2l.ai/t/1330

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
tensor([[[-1.1053,  0.1184,  0.0823, -0.2267],
         [ 2.4417, -1.0708,  0.1795, -0.3554],
         [ 0.3089, -0.4537,  1.1768,  1.3481]],

        [[-1.8293, -1.1143, -0.4053,  1.0142],
         [-0.4587, -0.8997,  1.1724, -0.9287],
         [-0.0265,  0.8163,  0.7636, -2.3677]]], grad_fn=<EmbeddingBackward>)
```

Cálculo de avanço do modelo Skip-Gram

No cálculo progressivo, a entrada do modelo skip-gram contém o índice central da palavra-alvo `center` e o contexto concatenado e o índice da palavra de ruído `contexts_and_negatives`. Em que, a variável `center` tem a forma (tamanho do lote, 1), enquanto a variável `contexts_and_negatives` tem a forma (tamanho do lote, `max_len`). Essas duas variáveis são primeiro transformadas de índices de palavras em vetores de palavras pela camada de incorporação de palavras e, em seguida, a saída da forma (tamanho do lote, 1, `max_len`) é obtida pela multiplicação de minibatch. Cada elemento na saída é o produto interno do vetor de palavra de destino central e do vetor de palavra de contexto ou vetor de palavra de ruído.

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

Verifique se a forma de saída deve ser (tamanho do lote, 1, `max_len`).

```
skip_gram(torch.ones((2, 1), dtype=torch.long),
          torch.ones((2, 4), dtype=torch.long), embed, embed).shape
```

```
torch.Size([2, 1, 4])
```

14.4.2 Treinamento

Antes de treinar o modelo de incorporação de palavras, precisamos definir a função de perda do modelo.

Função de perda de entropia cruzada binária

De acordo com a definição da função de perda na amostragem negativa, podemos usar diretamente a função de perda de entropia cruzada binária de APIs de alto nível.

```
class SigmoidBCELoss(nn.Module):
    "BCEWithLogitLoss with masking on call."
    def __init__(self):
        super().__init__()

    def forward(self, inputs, target, mask=None):
        out = nn.functional.binary_cross_entropy_with_logits(
            inputs, target, weight=mask, reduction="none")
        return out.mean(dim=1)

loss = SigmoidBCELoss()
```

Vale ressaltar que podemos usar a variável máscara para especificar o valor predito parcial e o rótulo que participam do cálculo da função de perda no minibatch: quando a máscara for 1, o valor predito e o rótulo da posição correspondente participarão do cálculo de a função de perda; Quando a máscara é 0, eles não participam. Como mencionamos anteriormente, as variáveis de máscara podem ser usadas para evitar o efeito do preenchimento nos cálculos da função de perda.

Dados dois exemplos idênticos, máscaras diferentes levam a valores de perda diferentes.

```
pred = torch.tensor([[.5]*4]*2)
label = torch.tensor([[1., 0., 1., 0.]]*2)
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask)
```

```
tensor([0.7241, 0.3620])
```

Podemos normalizar a perda em cada exemplo devido a vários comprimentos em cada exemplo.

```
loss(pred, label, mask) / mask.sum(axis=1) * mask.shape[1]
```

```
tensor([0.7241, 0.7241])
```

Inicializando os parâmetros do modelo

Construímos as camadas de incorporação das palavras central e de contexto, respectivamente, e definimos a dimensão do vetor da palavra hiperparâmetro `embed_size` para 100.

```
embed_size = 100
net = nn.Sequential(nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size),
                    nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size))
```

Treinamento

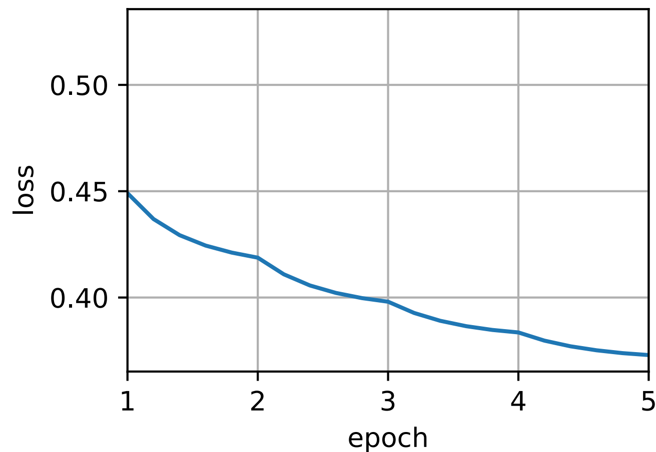
A função de treinamento é definida abaixo. Devido à existência de preenchimento, o cálculo da função de perda é ligeiramente diferente em comparação com as funções de treinamento anteriores.

```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    def init_weights(m):
        if type(m) == nn.Embedding:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs])
    metric = d2l.Accumulator(2)  # Sum of losses, no. of tokens
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
            optimizer.zero_grad()
            center, context_negative, mask, label = [
                data.to(device) for data in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])
            l = (loss(pred.reshape(label.shape).float(), label.float(), mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.sum().backward()
            optimizer.step()
            metric.add(l.sum(), l.numel())
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

Now, we can train a skip-gram model using negative sampling.

```
lr, num_epochs = 0.01, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.373, 368911.4 tokens/sec on cuda:0
```



14.4.3 Aplicando o modelo de incorporação de palavras

Depois de treinar o modelo de incorporação de palavras, podemos representar a similaridade no significado entre as palavras com base na similaridade do cosseno de dois vetores de palavras. Como podemos ver, ao usar o modelo de incorporação de palavras treinadas, as palavras com significado mais próximo da palavra "chip" estão principalmente relacionadas a chips.

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[vocab[query_token]]
    # Compute the cosine similarity. Add 1e-9 for numerical stability
    cos = torch.mv(W, x) / torch.sqrt(torch.sum(W * W, dim=1) *
                                      torch.sum(x * x) + 1e-9)
    topk = torch.topk(cos, k=k+1)[1].cpu().numpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.idx_to_token[i]}')

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.513: microprocessor
cosine sim=0.493: intel
cosine sim=0.451: drives
```

14.4.4 Sumário

- Podemos pré-treinar um modelo de grama de salto por meio de amostragem negativa.

14.4.5 Exercícios

1. Defina `sparse_grad = True` ao criar uma instância de `nn.Embedding`. Acelera o treinamento? Consulte a documentação do MXNet para aprender o significado desse argumento.
2. Tente encontrar sinônimos para outras palavras.
3. Ajuste os hiperparâmetros e observe e analise os resultados experimentais.
4. Quando o conjunto de dados é grande, costumamos amostrar as palavras de contexto e as palavras de ruído para a palavra de destino central no minibatch atual apenas ao atualizar os parâmetros do modelo. Em outras palavras, a mesma palavra de destino central pode ter palavras de contexto ou palavras de ruído diferentes em épocas diferentes. Quais são os benefícios desse tipo de treinamento? Tente implementar este método de treinamento.

Discussão[177]

14.5 Incorporação de palavras com vetores globais (GloVe)

Primeiro, devemos revisar o modelo skip-gram no word2vec. A probabilidade condicional $P(w_j \mid w_i)$ expressa no modelo skip-gram usando a operação softmax será registrada como q_{ij}, ou seja:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \tag{14.5.1}$$

onde $\mathbf{v}_i$ e $\mathbf{u}_i$ são as representações vetoriais da palavra w_i do índice i como a palavra central e a palavra de contexto, respectivamente, e $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ é o conjunto de índices de vocabulário.

Para a palavra w_i, ela pode aparecer no conjunto de dados várias vezes. Coletamos todas as palavras de contexto sempre que w_i é uma palavra central e mantemos duplicatas, denotadas como multiset $\mathcal{C}_i$. O número de um elemento em um multiconjunto é chamado de multiplicidade do elemento. Por exemplo, suponha que a palavra w_i apareça duas vezes no conjunto de dados: as janelas de contexto quando essas duas w_i se tornam palavras centrais na sequência de texto contêm índices de palavras de contexto $2, 1, 5, 2$ e $2, 3, 2, 1$. Então, multiset $\mathcal{C}_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$, onde a multiplicidade do elemento 1 é 2, a multiplicidade do elemento 2 é 4 e multiplicidades de os elementos 3 e 5 são 1. Denote a multiplicidade do elemento j no multiset $\mathcal{C}_i$ as x_{ij}: é o número da palavra w_j em todas as janelas de contexto para a palavra central w_i em todo o conjunto de dados. Como resultado, a função de perda do modelo skip-gram pode ser expressa de uma maneira diferente:

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log\, q_{ij}. \tag{14.5.2}$$

Adicionamos o número de todas as palavras de contexto para a palavra alvo central w_i para obter x_i, e registramos a probabilidade condicional x_{ij}/x_i para gerar a palavra de contexto w_j com base

[177] https://discuss.d2l.ai/t/1335

na palavra alvo central w_i como p_{ij}. Podemos reescrever a função de perda do modelo skip-gram como

$$-\sum_{i\in\mathcal{V}} x_i \sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}. \tag{14.5.3}$$

Na fórmula acima, $\sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}$ calcula a distribuição de probabilidade condicional p_{ij} para geração de palavras de contexto com base na central palavra-alvo w_i e a entropia cruzada da distribuição de probabilidade condicional q_{ij} prevista pelo modelo. A função de perda é ponderada usando a soma do número de palavras de contexto com a palavra alvo central w_i. Se minimizarmos a função de perda da fórmula acima, seremos capazes de permitir que a distribuição de probabilidade condicional prevista se aproxime o mais próximo possível da verdadeira distribuição de probabilidade condicional.

No entanto, embora o tipo mais comum de função de perda, a perda de entropia cruzada função às vezes não é uma boa escolha. Por um lado, como mencionamos em Section 14.2 o custo de deixar o a previsão do modelo q_{ij} torna-se a distribuição de probabilidade legal tem a soma de todos os itens em todo o dicionário em seu denominador. Isso pode facilmente levar a sobrecarga computacional excessiva. Por outro lado, costuma haver muitos palavras incomuns no dicionário e raramente aparecem no conjunto de dados. No função de perda de entropia cruzada, a previsão final da probabilidade condicional a distribuição em um grande número de palavras incomuns provavelmente será imprecisa.

14.5.1 O modelo GloVe

Para resolver isso, GloVe (Pennington et al., 2014), um modelo de incorporação de palavras que veio depois de word2vec, adota perda quadrada e faz três alterações no modelo de grama de salto com base nessa perda.

1. Aqui, usamos as variáveis de distribuição não probabilística $p'_{ij} = x_{ij}$ e $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ e pegue seus logs. Portanto, obtemos a perda quadrada $\left(\log\, p'_{ij} - \log\, q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log\, x_{ij}\right)^2$.
2. Adicionamos dois parâmetros do modelo escalar para cada palavra w_i: os termos de polarização b_i (para palavras-alvo centrais) e c_i (para palavras de contexto).
3. Substitua o peso de cada perda pela função $h(x_{ij})$. A função de peso $h(x)$ é uma função monótona crescente com o intervalo $[0, 1]$.

Portanto, o objetivo do GloVe é minimizar a função de perda.

$$\sum_{i\in\mathcal{V}} \sum_{j\in\mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log\, x_{ij}\right)^2. \tag{14.5.4}$$

Aqui, temos uma sugestão para a escolha da função de peso $h(x)$: quando $x < c$ (por exemplo, $c = 100$), faça $h(x) = (x/c)^\alpha$ (por exemplo, $\alpha = 0.75$), caso contrário, faça $h(x) = 1$. Como $h(0) = 0$, o termo de perda ao quadrado de $x_{ij} = 0$ pode ser simplesmente ignorado. Quando usamos o minibatch SGD para treinamento, conduzimos uma amostragem aleatória para obter um minibatch diferente de zero x_{ij} de cada intervalo de tempo e calculamos o gradiente para atualizar os parâmetros do modelo. Esses x_{ij} diferentes de zero são calculados antecipadamente com base em todo o conjunto de dados e contêm estatísticas globais para o conjunto de dados. Portanto, o nome GloVe é retirado de "Vetores globais".

Observe que se a palavra w_i aparecer na janela de contexto da palavra w_j, então a palavra w_j também aparecerá na janela de contexto da palavra w_i. Portanto, $x_{ij} = x_{ji}$. Ao contrário de word2vec, GloVe ajusta o simétrico log x_{ij} no lugar da probabilidade condicional assimétrica p_{ij}. Portanto, o vetor de palavra alvo central e o vetor de palavra de contexto de qualquer palavra são equivalentes no GloVe. No entanto, os dois conjuntos de vetores de palavras que são aprendidos pela mesma palavra podem ser diferentes no final devido a valores de inicialização diferentes. Depois de aprender todos os vetores de palavras, o GloVe usará a soma do vetor da palavra-alvo central e do vetor da palavra de contexto como o vetor da palavra final para a palavra.

14.5.2 Compreendendo o GloVe a partir das razões de probabilidade condicionais

Também podemos tentar entender a incorporação de palavras GloVe de outra perspectiva. Continuaremos a usar os símbolos anteriores nesta seção, $P(w_j \mid w_i)$ representa a probabilidade condicional de gerar a palavra de contexto w_j com a palavra alvo central w_i no conjunto de dados, e será registrado como p_{ij}. A partir de um exemplo real de um grande corpus, temos aqui os seguintes dois conjuntos de probabilidades condicionais com "gelo" e "vapor" como palavras-alvo centrais e a proporção entre elas:

w_k=	solid	gas	water	fashion
$p_1 = P(w_k \mid \text{ice})$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid \text{steam})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

Seremos capazes de observar fenômenos como:

- Para uma palavra w_k que está relacionada a "gelo", mas não a "vapor", como $w_k =$ solid, esperaríamos uma razão de probabilidade condicional maior, como o valor 8,9 na última linha da tabela acima.
- Para uma palavra w_k que está relacionada a "vapor", mas não a "gelo", como $w_k =$ gas, esperaríamos uma razão de probabilidade condicional menor, como o valor 0,085 na última linha da tabela acima.
- Para uma palavra w_k que está relacionada a "gelo" e "vapor", como $w_k =$ agua, esperaríamos uma razão de probabilidade condicional próxima de 1, como o valor 1,36 no último linha da tabela acima.
- Para uma palavra w_k que não está relacionada a "gelo" ou "vapor", como $w_k =$ fashion, esperaríamos uma razão de probabilidade condicional próxima de 1, como o valor 0,96 no último linha da tabela acima.

Podemos ver que a razão de probabilidade condicional pode representar a relação entre diferentes palavras de forma mais intuitiva. Podemos construir uma função de vetor de palavras para ajustar a razão de probabilidade condicional de forma mais eficaz. Como sabemos, para obter qualquer razão deste tipo são necessárias três palavras w_i, w_j, e w_k. A razão de probabilidade condicional com w_i como palavra alvo central é p_{ij}/p_{ik}. Podemos encontrar uma função que usa vetores de palavras para ajustar essa razão de probabilidade condicional.

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \tag{14.5.5}$$

O projeto possível da função f aqui não será exclusivo. Precisamos apenas considerar uma possibilidade mais razoável. Observe que a razão de probabilidade condicional é escalar, podemos

limitar f para ser uma função escalar: $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$. Depois de trocar o índice j por k, seremos capazes de ver que a função f satisfaz a condição $f(x)f(-x) = 1$, então uma possibilidade poderia ser $f(x) = \exp(x)$. Assim:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}. \tag{14.5.6}$$

Uma possibilidade que satisfaz o lado direito do sinal de aproximação é $\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$, onde α é uma constante. Considerando que $p_{ij} = x_{ij}/x_i$, após tomar o logaritmo, obtemos $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$. Usamos termos de polarização adicionais para ajustar $-\log \alpha + \log x_i$, como o termo de polarização de palavra-alvo central b_i e o termo de polarização de palavra de contexto c_j:

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij}). \tag{14.5.7}$$

Pegando o erro quadrado e ponderando os lados esquerdo e direito da fórmula acima, podemos obter a função de perda de GloVe.

14.5.3 Sumário

- Em alguns casos, a função de perda de entropia cruzada pode ter uma desvantagem. O GloVe usa a perda quadrática e o vetor de palavras para ajustar as estatísticas globais calculadas antecipadamente com base em todo o conjunto de dados.
- O vetor da palavra alvo central e o vetor da palavra de contexto de qualquer palavra são equivalentes no GloVe.

14.5.4 Exercícios

1. Se uma palavra aparecer na janela de contexto de outra palavra, como podemos usar o distância entre eles na sequência de texto para redesenhar o método para calculando a probabilidade condicional p_{ij}? Dica: Consulte a seção 4.2 do paper GloVe (Pennington et al., 2014).
2. Para qualquer palavra, o termo de polarização da palavra-alvo central e o termo de polarização da palavra de contexto serão equivalentes entre si no GloVe? Por quê?

Discussão[178]

14.6 Encontrando sinônimos e analogias

Em Section 14.4 treinamos um modelo de incorporação de palavras word2vec em um conjunto de dados de pequena escala e procurou por sinônimos usando a similaridade de cosseno de vetores de palavras. Na prática, vetores de palavras pré-treinados em um corpus de grande escala muitas vezes pode ser aplicado a tarefas de processamento de linguagem natural downstream. Esta seção irá demonstrar como usar esses vetores de palavras pré-treinados para encontrar sinônimos e analogias. Continuaremos a aplicar vetores de palavras pré-treinados em seções subsequentes.

[178] https://discuss.d2l.ai/t/385

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

14.6.1 Usando vetores de palavras pré-treinados

Abaixo lista os embeddings GloVe pré-treinados de dimensões 50, 100 e 300, que pode ser baixado do site do GloVe[179]. Os fastText pré-treinados embarcados estão disponíveis em vários idiomas. Aqui, consideramos uma versão em inglês ("wiki.en" 300-dimensional) que pode ser baixada do site fastText[180].

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

Definimos a seguinte classe TokenEmbedding para carregar os embeddings pré-treinados Glove e fastText acima.

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe website: https://nlp.stanford.edu/projects/glove/
        # fastText website: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
```

(continues on next page)

[179] https://nlp.stanford.edu/projects/glove/
[180] https://fasttext.cc/

(continued from previous page)

```
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, torch.tensor(idx_to_vec)

    def __getitem__(self, tokens):
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[torch.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

Em seguida, usamos embeddings GloVe de 50 dimensões pré-treinados em um subconjunto da Wikipedia. A incorporação de palavras correspondente é baixada automaticamente na primeira vez que criamos uma instância de incorporação de palavras pré-treinada.

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

```
Downloading ../data/glove.6B.50d.zip from http://d2l-data.s3-accelerate.amazonaws.com/glove.
↪6B.50d.zip...
```

Produza o tamanho do dicionário. O dicionário contém 400.000 palavras e um token especial desconhecido.

```
len(glove_6b50d)
```

```
400001
```

Podemos usar uma palavra para obter seu índice no dicionário ou podemos obter a palavra de seu índice.

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

14.6.2 Aplicação de vetores de palavras pré-treinados

Abaixo, demonstramos a aplicação de vetores de palavras pré-treinados, usando GloVe como exemplo.

Encontrando sinônimos

Aqui, reimplementamos o algoritmo usado para pesquisar sinônimos por cosseno similaridade introduzida em Section 14.1

A fim de reutilizar a lógica para buscar os k vizinhos mais próximos quando buscando analogias, encapsulamos esta parte da lógica separadamente no `knn` função (k-vizinhos mais próximos).

```
def knn(W, x, k):
    # The added 1e-9 is for numerical stability
    cos = torch.mv(W, x.reshape(-1,)) / (
        torch.sqrt(torch.sum(W * W, axis=1) + 1e-9) *
        torch.sqrt((x * x).sum()))
    _, topk = torch.topk(cos, k=k)
    return topk, [cos[int(i)] for i in topk]
```

Em seguida, buscamos sinônimos pré-treinando a instância do vetor de palavras `embed`.

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Remove input words
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```

O dicionário de instância de vetor de palavras pré-treinadas `glove_6b50d` já criado contém 400.000 palavras e um token especial desconhecido. Excluindo palavras de entrada e palavras desconhecidas, procuramos as três palavras que têm o significado mais semelhante a "chip".

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

A seguir, procuramos os sinônimos de "baby" e "beautiful".

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

Encontrando Analogias

Além de buscar sinônimos, também podemos usar o vetor de palavras pré-treinadas para buscar analogias entre palavras. Por exemplo, "man":"woman"::"son":"daughter" é um exemplo de analogia, "man" está para "woman" como "son" está para "daughter". O problema de buscar analogias pode ser definido da seguinte forma: para quatro palavras na relação analógica $a : b :: c : d$, dadas as três primeiras palavras, a, b e c, queremos encontre d. Suponha que a palavra vetor para a palavra w seja vec(w). Para resolver o problema de analogia, precisamos encontrar o vetor de palavras que é mais semelhante ao vetor de resultado de vec(c) + vec(b) − vec(a).

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

Verifique a analogia "male-female".

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

Analogia de "país-capital": "beijing" é para "china" como "tokyo" é para quê? A resposta deve ser "japão".

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

Analogia do "adjetivo-adjetivo superlativo": "ruim" está para o "pior", assim como "grande" está para o quê? A resposta deve ser "maior".

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

Analogia do "verbo presente-verbo no pretérito": "do" é "did" assim como "go" é para quê? A resposta deve ser "went".

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

14.6.3 Sumário

- Vetores de palavras pré-treinados em um corpus de grande escala podem frequentemente ser aplicados a tarefas de processamento de linguagem natural downstream.
- Podemos usar vetores de palavras pré-treinados para buscar sinônimos e analogias.

14.6.4 Exercícios

1. Teste os resultados do fastText usando `TokenEmbedding ('wiki.en')`.
2. Se o dicionário for extremamente grande, como podemos acelerar a localização de sinônimos e analogias?

Discussão[181]

14.7 Representações de codificador bidirecional de transformadores (BERT)

Introduzimos vários modelos de incorporação de palavras para a compreensão da linguagem natural. Após o pré-treinamento, a saída pode ser pensada como uma matriz onde cada linha é um vetor que representa uma palavra de um vocabulário predefinido. Na verdade, esses modelos de incorporação de palavras são todos *independentes do contexto*. Vamos começar ilustrando essa propriedade.

14.7.1 De Independente do Contexto para Sensível ao Contexto

Lembre-se dos experimentos em Section 14.4 e Section 14.6. Por exemplo, word2vec e GloVe atribuem o mesmo vetor pré-treinado à mesma palavra, independentemente do contexto da palavra (se houver). Formalmente, uma representação independente de contexto de qualquer token x é uma função $f(x)$ que leva apenas x como entrada. Dada a abundância de polissemia e semântica complexa em linguagens naturais, representações independentes de contexto têm limitações óbvias. Por exemplo, a palavra "guindaste" em contextos "um guindaste está voando" e "um motorista de guindaste veio" têm significados completamente diferentes; assim, a mesma palavra pode receber diferentes representações dependendo dos contextos.

Isso motiva o desenvolvimento de representações de palavras *sensíveis ao contexto*, onde as representações de palavras dependem de seus contextos. Portanto, uma representação sensível ao contexto do token x é uma função $f(x, c(x))$ dependendo de x e de seu contexto $c(x)$. Representações contextuais populares incluem TagLM (tagger de sequência aumentada de modelo de linguagem) (Peters et al., 2017b), CoVe (vetores de contexto) (McCann et al., 2017), e ELMo (Embeddings from Language Models) (Peters et al., 2018).

Por exemplo, tomando toda a sequência como entrada, ELMo é uma função que atribui uma representação a cada palavra da sequência de entrada. Especificamente, o ELMo combina todas as representações da camada intermediária do LSTM bidirecional pré-treinado como a representação de saída. Em seguida, a representação ELMo será adicionada ao modelo supervisionado existente de uma tarefa downstream como recursos adicionais, como concatenando a representação ELMo e a representação original (por exemplo, GloVe) de tokens no modelo existente. Por

[181] https://discuss.d2l.ai/t/1336

um lado, todos os pesos no modelo LSTM bidirecional pré-treinado são congelados após as representações ELMo serem adicionadas. Por outro lado, o modelo supervisionado existente é personalizado especificamente para uma determinada tarefa. Aproveitando os melhores modelos diferentes para diferentes tarefas naquele momento, adicionar ELMo melhorou o estado da arte em seis tarefas de processamento de linguagem natural: análise de sentimento, inferência de linguagem natural, rotulagem de função semântica, resolução de co-referência, reconhecimento de entidade nomeada e resposta a perguntas.

14.7.2 De Task-Specific para Task-Agnostic

Embora o ELMo tenha melhorado significativamente as soluções para um conjunto diversificado de tarefas de processamento de linguagem natural, cada solução ainda depende de uma arquitetura *específica para a tarefa*. No entanto, é praticamente não trivial criar uma arquitetura específica para cada tarefa de processamento de linguagem natural. O modelo GPT (Generative Pre-Training) representa um esforço na concepção um modelo *agnóstico de tarefa* geral para representações sensíveis ao contexto (Radford et al., 2018). Construído em um decodificador de transformador, O GPT pré-treina um modelo de linguagem que será usado para representar sequências de texto. Ao aplicar GPT a uma tarefa downstream, a saída do modelo de linguagem será alimentada em uma camada de saída linear adicionada para prever o rótulo da tarefa. Em nítido contraste com o ELMo que congela os parâmetros do modelo pré-treinado, GPT ajusta *todos* os parâmetros no decodificador de transformador pré-treinado durante a aprendizagem supervisionada da tarefa a jusante. GPT foi avaliada em doze tarefas de inferência de linguagem natural, resposta a perguntas, similaridade de sentenças e classificação, e melhorou o estado da arte em nove deles com mudanças mínimas para a arquitetura do modelo.

No entanto, devido à natureza autoregressiva dos modelos de linguagem, O GPT apenas olha para a frente (da esquerda para a direita). Em contextos "fui ao banco para depositar dinheiro" e "fui ao banco para me sentar", como "banco" é sensível ao contexto à sua esquerda, GPT retornará a mesma representação para "banco", embora tenha significados diferentes.

14.7.3 BERT: Combinando o melhor dos dois mundos

Como nós vimos, O ELMo codifica o contexto bidirecionalmente, mas usa arquiteturas específicas para tarefas; enquanto o GPT é agnóstico em relação à tarefa, mas codifica o contexto da esquerda para a direita. Combinando o melhor dos dois mundos, BERT (Representações de Codificador Bidirecional de Transformadores) codifica o contexto bidirecionalmente e requer mudanças mínimas de arquitetura para uma ampla gama de tarefas de processamento de linguagem natural (Devlin et al., 2018). Usando um codificador de transformador pré-treinado, O BERT é capaz de representar qualquer token com base em seu contexto bidirecional. Durante a aprendizagem supervisionada de tarefas posteriores, O BERT é semelhante ao GPT em dois aspectos. Primeiro, as representações de BERT serão alimentadas em uma camada de saída adicionada, com mudanças mínimas na arquitetura do modelo, dependendo da natureza das tarefas, como a previsão para cada token versus a previsão para a sequência inteira. Segundo, todos os parâmetros do codificador de transformador pré-treinado são ajustados, enquanto a camada de saída adicional será treinada do zero. Fig. 14.7.1 descreve as diferenças entre ELMo, GPT e BERT.

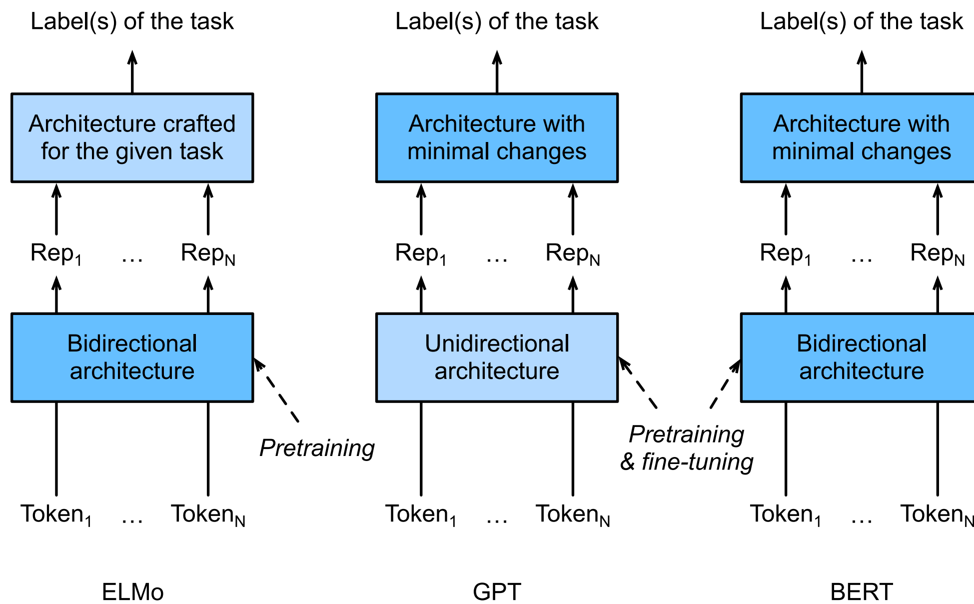


Fig. 14.7.1: A comparison of ELMo, GPT, and BERT.

O BERT melhorou ainda mais o estado da arte em onze tarefas de processamento de linguagem natural sob categorias amplas de i) classificação de texto único (por exemplo, análise de sentimento), ii) classificação de pares de texto (por exemplo, inferência de linguagem natural), iii) resposta a perguntas, iv) marcação de texto (por exemplo, reconhecimento de entidade nomeada). Tudo proposto em 2018, de ELMo sensível ao contexto a GPT e BERT agnósticos de tarefa, O pré-treinamento conceitualmente simples, mas empiricamente poderoso, de representações profundas para linguagens naturais revolucionou as soluções para várias tarefas de processamento de linguagem natural.

No resto deste capítulo, vamos mergulhar no pré-treinamento de BERT. Quando os aplicativos de processamento de linguagem natural são explicados em Chapter 15, ilustraremos o ajuste fino de BERT para aplicações downstream.

```
import torch
from torch import nn
from d2l import torch as d2l
```

14.7.4 Representação de entrada

No processamento de linguagem natural, algumas tarefas (por exemplo, análise de sentimento) usam um único texto como entrada, enquanto em algumas outras tarefas (por exemplo, inferência de linguagem natural), a entrada é um par de sequências de texto. A sequência de entrada de BERT representa sem ambiguidade texto único e pares de texto. Na antiga, a sequência de entrada de BERT é a concatenação de o token de classificação especial “<sep>”, tokens de uma sequência de texto, e o token de separação especial “<sep>”. No ultimo, a sequência de entrada de BERT é a concatenação de “<sep>”, tokens da primeira sequência de texto, “<sep>”, tokens da segunda sequência de texto e “<sep>”. Iremos distinguir de forma consistente a terminologia “sequência de entrada de BERT” de outros tipos de “sequências”. Por exemplo, uma *sequência de entrada de BERT* pode incluir uma *sequência de texto* ou duas *sequências de texto*.

Para distinguir pares de texto, o segmento aprendido embeddings $\mathbf{e}_A$ e $\mathbf{e}_B$ são adicionados aos embeddings de token da primeira e da segunda sequência, respectivamente. Para entradas de texto único, apenas $\mathbf{e}_A$ é usado.

O seguinte `get_tokens_and_segments` leva uma ou duas frases como entrada, em seguida, retorna tokens da sequência de entrada BERT e seus IDs de segmento correspondentes.

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0 and 1 are marking segment A and B, respectively
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

O BERT escolhe o codificador do transformador como sua arquitetura bidirecional. Comum no codificador do transformador, embeddings posicionais são adicionados em cada posição da sequência de entrada BERT. No entanto, diferente do codificador do transformador original, O BERT usa embeddings posicionais *aprendíveis*. Para resumir, Fig. 14.7.2 mostra que os embeddings da sequência de entrada de BERT são a soma dos embeddings de token, embeddings de segmento e embeddings posicionais.

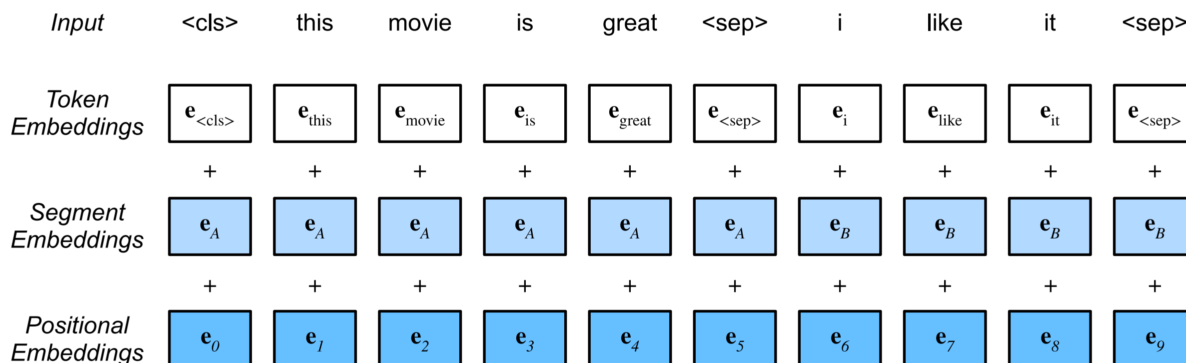


Fig. 14.7.2: Os embeddings da sequência de entrada de BERT são a soma dos embeddings de token, embeddings de segmento e embeddings posicionais.

A seguinte classe `BERTEncoder` é semelhante à classe `TransformerEncoder` conforme implementado em Section 10.7. Diferente de `TransformerEncoder`, `BERTEncoder` usa embeddings de segmento e embeddings posicionais aprendíveis.

```
#@save
class BERTEncoder(nn.Module):
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
```

(continues on next page)

(continued from previous page)

```
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module(f"{i}", d2l.EncoderBlock(
                key_size, query_size, value_size, num_hiddens, norm_shape,
                ffn_num_input, ffn_num_hiddens, num_heads, dropout, True))
        # In BERT, positional embeddings are learnable, thus we create a
        # parameter of positional embeddings that are long enough
        self.pos_embedding = nn.Parameter(torch.randn(1, max_len,
                                                      num_hiddens))

    def forward(self, tokens, segments, valid_lens):
        # Shape of `X` remains unchanged in the following code snippet:
        # (batch size, max sequence length, `num_hiddens`)
        X = self.token_embedding(tokens) + self.segment_embedding(segments)
        X = X + self.pos_embedding.data[:, :X.shape[1], :]
        for blk in self.blks:
            X = blk(X, valid_lens)
        return X
```

Suppose that the vocabulary size is 10,000. To demonstrate forward inference of `BERTEncoder`, let us create an instance of it and initialize its parameters.

Suponha que o tamanho do vocabulário seja 10.000. Para demonstrar a inferência direta de `BERTEncoder`, vamos criar uma instância dele e inicializar seus parâmetros.

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
norm_shape, ffn_num_input, num_layers, dropout = [768], 768, 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape, ffn_num_input,
                      ffn_num_hiddens, num_heads, num_layers, dropout)
```

Definimos `tokens` como sendo 2 sequências de entrada BERT de comprimento 8, onde cada token é um índice do vocabulário. A inferência direta de `BERTEncoder` com os `tokens` de entrada retorna o resultado codificado onde cada token é representado por um vetor cujo comprimento é predefinido pelo hiperparâmetro `num_hiddens`. Esse hiperparâmetro geralmente é conhecido como *tamanho oculto* (número de unidades ocultas) do codificador do transformador.

```
tokens = torch.randint(0, vocab_size, (2, 8))
segments = torch.tensor([[0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

```
torch.Size([2, 8, 768])
```

14.7.5 Tarefas de pré-treinamento

A inferência direta de `BERTEncoder` dá a representação de BERT de cada token do texto de entrada e o inserido tokens especiais “<cls>” e “<seq>”. A seguir, usaremos essas representações para calcular a função de perda para pré-treinamento de BERT. O pré-treinamento é composto pelas duas tarefas a seguir: modelagem de linguagem mascarada e previsão da próxima frase.

Modelagem de linguagem mascarada

Conforme ilustrado em Section 8.3, um modelo de linguagem prevê um token usando o contexto à sua esquerda. Para codificar o contexto bidirecionalmente para representar cada token, BERT mascara tokens aleatoriamente e usa tokens do contexto bidirecional para prever os tokens mascarados. Esta tarefa é conhecida como *modelo de linguagem mascarada*.

Nesta tarefa de pré-treinamento, 15% dos tokens serão selecionados aleatoriamente como os tokens mascarados para previsão. Para prever um token mascarado sem trapacear usando o rótulo, uma abordagem direta é sempre substituí-lo por um “<mask>” especial token na sequência de entrada BERT. No entanto, o token especial artificial “<mask>” nunca aparecerá no ajuste fino. Para evitar essa incompatibilidade entre o pré-treinamento e o ajuste fino, se um token for mascarado para previsão (por exemplo, “ótimo” foi selecionado para ser mascarado e previsto em “este filme é ótimo”), na entrada, ele será substituído por:

- uma “<mask>” especial token 80% do tempo (por exemplo, “este filme é ótimo” torna-se “este filme é <mask>”);
- um token aleatório 10% do tempo (por exemplo, “este filme é ótimo” torna-se “este filme é uma bebida”);
- o token de rótulo inalterado em 10% do tempo (por exemplo, “este filme é ótimo” torna-se “este filme é ótimo”).

Observe que por 10% de 15% do tempo, um token aleatório é inserido. Este ruído ocasional encoraja o BERT a ser menos inclinado para o token mascarado (especialmente quando o token de rótulo permanece inalterado) em sua codificação de contexto bidirecional.

Implementamos a seguinte classe `MaskLM` para prever tokens mascarados na tarefa de modelo de linguagem mascarada de pré-treinamento de BERT. A previsão usa um MLP de uma camada oculta (`self.mlp`). Na inferência direta, são necessárias duas entradas: o resultado codificado de `BERTEncoder` e as posições do token para predição. A saída são os resultados da previsão nessas posições.

```
#@save
class MaskLM(nn.Module):
    def __init__(self, vocab_size, num_hiddens, num_inputs=768, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential(nn.Linear(num_inputs, num_hiddens),
                                 nn.ReLU(),
                                 nn.LayerNorm(num_hiddens),
                                 nn.Linear(num_hiddens, vocab_size))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
        batch_size = X.shape[0]
        batch_idx = torch.arange(0, batch_size)
```

(continues on next page)

(continued from previous page)

```
        # Suppose that `batch_size` = 2, `num_pred_positions` = 3, then
        # `batch_idx` is `torch.tensor([0, 0, 0, 1, 1, 1])`
        batch_idx = torch.repeat_interleave(batch_idx, num_pred_positions)
        masked_X = X[batch_idx, pred_positions]
        masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
        mlm_Y_hat = self.mlp(masked_X)
        return mlm_Y_hat
```

Para demonstrar a inferência direta de `MaskLM`, nós criamos sua instância `mlm` e a inicializamos. Lembre-se de que `encoded_X` da inferência direta de `BERTEncoder` representa 2 sequências de entrada de BERT. Definimos `mlm_positions` como os 3 índices a serem previstos em qualquer sequência de entrada de BERT de`encoded_X`. A inferência direta de `mlm` retorna os resultados de predição `mlm_Y_hat` em todas as posições mascaradas `mlm_positions` de `encoded_X`. Para cada previsão, o tamanho do resultado é igual ao tamanho do vocabulário.

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm_positions = torch.tensor([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

```
torch.Size([2, 3, 10000])
```

Com os rótulos de verdade do solo `mlm_Y` dos tokens previstos `mlm_Y_hat` sob as máscaras, podemos calcular a perda de entropia cruzada da tarefa do modelo de linguagem mascarada no pré-treinamento de BERT.

```
mlm_Y = torch.tensor([[7, 8, 9], [10, 20, 30]])
loss = nn.CrossEntropyLoss(reduction='none')
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

```
torch.Size([6])
```

Previsão da próxima frase

Embora a modelagem de linguagem mascarada seja capaz de codificar o contexto bidirecional para representar palavras, não modela explicitamente a relação lógica entre pares de texto. Para ajudar a entender a relação entre duas sequências de texto, O BERT considera uma tarefa de classificação binária, *previsão da próxima frase*, em seu pré-treinamento. Ao gerar pares de frases para pré-treinamento, na metade do tempo, são de fato sentenças consecutivas com o rótulo "Verdadeiro"; enquanto, na outra metade do tempo, a segunda frase é amostrada aleatoriamente do corpus com o rótulo "Falso".

A seguinte classe `NextSentencePred` usa um MLP de uma camada oculta para prever se a segunda frase é a próxima frase da primeira na seqüência de entrada de BERT. Devido à autoatenção no codificador do transformador, a representação BERT do token especial "<cls>" codifica as duas sentenças da entrada. Portanto, a camada de saída (`self.output`) do classificador MLP leva `X` como entrada, onde `X` é a saída da camada oculta MLP cuja entrada é o código "<cls>" símbolo.

```
#@save
class NextSentencePred(nn.Module):
    def __init__(self, num_inputs, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Linear(num_inputs, 2)

    def forward(self, X):
        # `X` shape: (batch size, `num_hiddens`)
        return self.output(X)
```

Podemos ver que a inferência direta de uma instância `NextSentencePred` retorna previsões binárias para cada sequência de entrada de BERT.

```
# PyTorch by default won't flatten the tensor as seen in mxnet where, if
# flatten=True, all but the first axis of input data are collapsed together
encoded_X = torch.flatten(encoded_X, start_dim=1)
# input_shape for NSP: (batch size, `num_hiddens`)
nsp = NextSentencePred(encoded_X.shape[-1])
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

```
torch.Size([2, 2])
```

A perda de entropia cruzada das 2 classificações binárias também pode ser calculada.

```
nsp_y = torch.tensor([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

```
torch.Size([2])
```

É digno de nota que todos os rótulos em ambas as tarefas de pré-treinamento mencionadas pode ser obtido trivialmente a partir do corpus de pré-treinamento sem esforço de rotulagem manual. O BERT original foi pré-treinado na concatenação de BookCorpus (Zhu et al., 2015) e Wikipedia em inglês. Esses dois corpora de texto são enormes: eles têm 800 milhões de palavras e 2,5 bilhões de palavras, respectivamente.

14.7.6 Juntando todas as coisas

Ao pré-treinamento de BERT, a função de perda final é uma combinação linear de ambas as funções de perda para modelagem de linguagem mascarada e previsão da próxima frase. Agora podemos definir a classe `BERTModel` instanciando as três classes `BERTEncoder`,`MaskLM` e `NextSentencePred`. A inferência direta retorna as representações codificadas de BERT `encoded_X`, previsões de modelagem de linguagem mascarada `mlm_Y_hat`, e as previsões da próxima frase `nsp_Y_hat`.

```
#@save
class BERTModel(nn.Module):
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
```

(continues on next page)

(continued from previous page)

```
                 hid_in_features=768, mlm_in_features=768,
                 nsp_in_features=768):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape,
                    ffn_num_input, ffn_num_hiddens, num_heads, num_layers,
                    dropout, max_len=max_len, key_size=key_size,
                    query_size=query_size, value_size=value_size)
        self.hidden = nn.Sequential(nn.Linear(hid_in_features, num_hiddens),
                                    nn.Tanh())
        self.mlm = MaskLM(vocab_size, num_hiddens, mlm_in_features)
        self.nsp = NextSentencePred(nsp_in_features)

    def forward(self, tokens, segments, valid_lens=None, pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        # The hidden layer of the MLP classifier for next sentence prediction.
        # 0 is the index of the '<cls>' token
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

14.7.7 Sumário

- Os modelos de incorporação de palavras, como word2vec e GloVe, são independentes do contexto. Eles atribuem o mesmo vetor pré-treinado à mesma palavra, independentemente do contexto da palavra (se houver). É difícil para eles lidar bem com a polissemia ou semântica complexa em linguagens naturais.
- Para representações de palavras sensíveis ao contexto, como ELMo e GPT, as representações de palavras dependem de seus contextos.
- ELMo codifica o contexto bidirecionalmente, mas usa arquiteturas de tarefas específicas (no entanto, é praticamente não trivial criar uma arquitetura específica para cada tarefa de processamento de linguagem natural); enquanto o GPT é agnóstico em relação à tarefa, mas codifica o contexto da esquerda para a direita.
- O BERT combina o melhor dos dois mundos: ele codifica o contexto bidirecionalmente e requer mudanças mínimas de arquitetura para uma ampla gama de tarefas de processamento de linguagem natural.
- Os embeddings da sequência de entrada BERT são a soma dos embeddings de token, embeddings de segmento e embeddings posicionais.
- O pré-treinamento do BERT é composto de duas tarefas: modelagem de linguagem mascarada e previsão da próxima frase. O primeiro é capaz de codificar contexto bidirecional para representar palavras, enquanto o último modela explicitamente a relação lógica entre pares de texto.

14.7.8 Exercícios

1. Por que o BERT é bem-sucedido?
2. Todas as outras coisas sendo iguais, um modelo de linguagem mascarada exigirá mais ou menos etapas de pré-treinamento para convergir do que um modelo de linguagem da esquerda para a direita? Por quê?
3. Na implementação original do BERT, a rede feed-forward posicional em `BERTEncoder` (via `d2l.EncoderBlock`) e a camada totalmente conectada em `MaskLM` usam a unidade linear de erro Gaussiano (GELU) (Hendrycks & Gimpel, 2016) como a função de ativação. Pesquisa sobre a diferença entre GELU e ReLU.

Discussão[182]

14.8 O conjunto de dados para pré-treinamento de BERT

Para pré-treinar o modelo BERT conforme implementado em Section 14.7, precisamos gerar o conjunto de dados no formato ideal para facilitar as duas tarefas de pré-treinamento: modelagem de linguagem mascarada e previsão da próxima frase. Por um lado, o modelo BERT original é pré-treinado na concatenação de dois enormes corpora BookCorpus e Wikipedia em inglês (ver Section 14.7.5), tornando difícil para a maioria dos leitores deste livro. Por outro lado, o modelo pré-treinado de BERT pronto para uso pode não se adequar a aplicativos de domínios específicos, como medicina. Portanto, está ficando popular pré-treinar o BERT em um conjunto de dados customizado. Para facilitar a demonstração do pré-treinamento de BERT, usamos um corpus menor do WikiText-2 (Merity et al., 2016).

Comparando com o conjunto de dados PTB usado para pré-treinamento de word2vec em Section 14.3, WikiText-2 i) retém a pontuação original, tornando-a adequada para a previsão da próxima frase; ii) retém a caixa e os números originais; iii) é duas vezes maior.

```
import os
import random
import torch
from d2l import torch as d2l
```

No conjunto de dados WikiText-2, cada linha representa um parágrafo onde o espaço é inserido entre qualquer pontuação e seu token anterior. Os parágrafos com pelo menos duas frases são mantidos. Para dividir frases, usamos apenas o ponto como delimitador para simplificar. Deixamos discussões de técnicas de divisão de frases mais complexas nos exercícios no final desta seção.

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
```

(continues on next page)

[182] https://discuss.d2l.ai/t/1490

(continued from previous page)

```
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Uppercase letters are converted to lowercase ones
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

14.8.1 Definindo funções auxiliares para tarefas de pré-treinamento

Na sequência, começamos implementando funções auxiliares para as duas tarefas de pré-treinamento de BERT: previsão da próxima frase e modelagem de linguagem mascarada. Essas funções auxiliares serão chamadas mais tarde ao transformar o corpus de texto bruto no conjunto de dados do formato ideal para pré-treinar o BERT.

Gerando a Tarefa de Previsão da Próxima Sentença

De acordo com as descrições de Section 14.7.5, a função `_get_next_sentence` gera um exemplo de treinamento para a tarefa de classificação binária.

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` is a list of lists of lists
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

A função a seguir gera exemplos de treinamento para a previsão da próxima frase do `paragraph` de entrada invocando a função `_get_next_sentence`. Aqui, `paragraph` é uma lista de frases, onde cada frase é uma lista de tokens. O argumento `max_len` especifica o comprimento máximo de uma sequência de entrada de BERT durante o pré-treinamento.

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # Consider 1 '<cls>' token and 2 '<sep>' tokens
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```

Gerando a Tarefa de Modelagem de Linguagem Mascarada

A fim de gerar exemplos de treinamento para a tarefa de modelagem de linguagem mascarada de uma sequência de entrada de BERT, definimos a seguinte função `_replace_mlm_tokens`. Em suas entradas, `tokens` é uma lista de tokens que representam uma sequência de entrada de BERT, `candidate_pred_positions` é uma lista de índices de token da sequência de entrada BERT excluindo aqueles de tokens especiais (tokens especiais não são previstos na tarefa de modelagem de linguagem mascarada), e `num_mlm_preds` indica o número de previsões (recorde 15% de tokens aleatórios para prever). Seguindo a definição da tarefa de modelagem de linguagem mascarada em Section 14.7.5, em cada posição de previsão, a entrada pode ser substituída por uma "<mask>" especial token ou um token aleatório, ou permanecem inalterados. No final, a função retorna os tokens de entrada após possível substituição, os índices de token onde as previsões ocorrem e os rótulos para essas previsões.

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # Make a new copy of tokens for the input of a masked language model,
    # where the input may contain replaced '<mask>' or random tokens
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Shuffle for getting 15% random tokens for prediction in the masked
    # language modeling task
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80% of the time: replace the word with the '<mask>' token
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10% of the time: keep the word unchanged
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10% of the time: replace the word with a random word
            else:
                masked_token = random.randint(0, len(vocab) - 1)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

Ao invocar a função `_replace_mlm_tokens` mencionada, a função a seguir leva uma sequência de entrada de BERT (`tokens`) como uma entrada e retorna os índices dos tokens de entrada (após possível substituição de token conforme descrito em Section 14.7.5), os índices de token onde as previsões acontecem, e índices de rótulo para essas previsões.

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` is a list of strings
    for i, token in enumerate(tokens):
        # Special tokens are not predicted in the masked language modeling
```

(continues on next page)

(continued from previous page)

```
        # task
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 15% of random tokens are predicted in the masked language modeling task
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

14.8.2 Transformando texto em conjunto de dados de pré-treinamento

Agora estamos quase prontos para customizar uma classe `Dataset` para pré-treinamento de BERT. Antes disso, ainda precisamos definir uma função auxiliar `_pad_bert_inputs` para anexar a seção especial "<mask>" tokens para as entradas. Seu argumento `examples` contém as saídas das funções auxiliares `_get_nsp_data_from_paragraph` e `_get_mlm_data_from_tokens` para as duas tarefas de pré-treinamento.

```
#@save
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens,  = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
         is_next) in examples:
        all_token_ids.append(torch.tensor(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype=torch.long))
        all_segments.append(torch.tensor(segments + [0] * (
            max_len - len(segments)), dtype=torch.long))
        # `valid_lens` excludes count of '<pad>' tokens
        valid_lens.append(torch.tensor(len(token_ids), dtype=torch.float32))
        all_pred_positions.append(torch.tensor(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype=torch.long))
        # Predictions of padded tokens will be filtered out in the loss via
        # multiplication of 0 weights
        all_mlm_weights.append(
            torch.tensor([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)),
                dtype=torch.float32))
        all_mlm_labels.append(torch.tensor(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype=torch.long))
        nsp_labels.append(torch.tensor(is_next, dtype=torch.long))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

Colocando as funções auxiliares para gerar exemplos de treinamento das duas tarefas de pré-treinamento, e a função auxiliar para preencher as entradas juntas, nós personalizamos a seguinte classe `_WikiTextDataset` como o conjunto de dados WikiText-2 para pré-treinamento de BERT.

Implementando a função `__getitem__`, podemos acessar arbitrariamente os exemplos de pré-treinamento (modelagem de linguagem mascarada e previsão da próxima frase) gerado a partir de um par de frases do corpus WikiText-2.

O modelo BERT original usa embeddings WordPiece cujo tamanho de vocabulário é 30.000 (Wu et al., 2016). O método de tokenização do WordPiece é uma ligeira modificação de o algoritmo de codificação de par de bytes original em `subsec_Byte_Pair_Encoding`. Para simplificar, usamos a função `d2l.tokenize` para tokenização. Tokens raros que aparecem menos de cinco vezes são filtrados.

```
#@save
class _WikiTextDataset(torch.utils.data.Dataset):
    def __init__(self, paragraphs, max_len):
        # Input `paragraphs[i]` is a list of sentence strings representing a
        # paragraph; while output `paragraphs[i]` is a list of sentences
        # representing a paragraph, where each sentence is a list of tokens
        paragraphs = [d2l.tokenize(
            paragraph, token='word') for paragraph in paragraphs]
        sentences = [sentence for paragraph in paragraphs
                     for sentence in paragraph]
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[
            '<pad>', '<mask>', '<cls>', '<sep>'])
        # Get data for the next sentence prediction task
        examples = []
        for paragraph in paragraphs:
            examples.extend(_get_nsp_data_from_paragraph(
                paragraph, paragraphs, self.vocab, max_len))
        # Get data for the masked language model task
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)
                      + (segments, is_next))
                     for tokens, segments, is_next in examples]
        # Pad inputs
        (self.all_token_ids, self.all_segments, self.valid_lens,
         self.all_pred_positions, self.all_mlm_weights,
         self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(
            examples, max_len, self.vocab)

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx], self.all_pred_positions[idx],
                self.all_mlm_weights[idx], self.all_mlm_labels[idx],
                self.nsp_labels[idx])

    def __len__(self):
        return len(self.all_token_ids)
```

Usando a função `_read_wiki` e a classe `_WikiTextDataset`, definimos o seguinte `load_data_wiki` para download e conjunto de dados WikiText-2 e gerar exemplos de pré-treinamento a partir dele.

```
#@save
def load_data_wiki(batch_size, max_len):
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
```

(continues on next page)

(continued from previous page)

```
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                        shuffle=True, num_workers=num_workers)
    return train_iter, train_set.vocab
```

Setting the batch size to 512 and the maximum length of a BERT input sequence to be 64, we print out the shapes of a minibatch of BERT pretraining examples. Note that in each BERT input sequence, 10 (64×0.15) positions are predicted for the masked language modeling task.

Configurando o tamanho do lote para 512 e o comprimento máximo de uma sequência de entrada de BERT para 64, imprimimos as formas de um minibatch de exemplos de pré-treinamento de BERT. Observe que em cada sequência de entrada de BERT, 10 (64×0.15) posições estão previstas para a tarefa de modelagem de linguagem mascarada.

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break
```

```
Downloading ../data/wikitext-2-v1.zip from https://s3.amazonaws.com/research.metamind.io/
↪wikitext/wikitext-2-v1.zip...
torch.Size([512, 64]) torch.Size([512, 64]) torch.Size([512]) torch.Size([512, 10]) torch.
↪Size([512, 10]) torch.Size([512, 10]) torch.Size([512])
```

No final, vamos dar uma olhada no tamanho do vocabulário. Mesmo depois de filtrar tokens pouco frequentes, ainda é mais de duas vezes maior do que o conjunto de dados do PTB.

```
len(vocab)
```

```
20256
```

14.8.3 Sumário

- Comparando com o conjunto de dados PTB, o conjunto de datas WikiText-2 retém a pontuação, caixa e números originais e é duas vezes maior.
- Podemos acessar arbitrariamente os exemplos de pré-treinamento (modelagem de linguagem mascarada e previsão da próxima frase) gerados a partir de um par de frases do corpus WikiText-2.

14.8.4 Exercícios

1. Para simplificar, o período é usado como o único delimitador para dividir frases. Experimente outras técnicas de divisão de frases, como spaCy e NLTK. Tome o NLTK como exemplo. Você precisa instalar o NLTK primeiro: `pip install nltk`. No código, primeiro `import nltk`. Então, baixe o tokenizer de frase Punkt: `nltk.download('punkt')`. Para dividir frases como `sentences = 'This is great ! Why not ?'`, Invocar `nltk.tokenize.sent_tokenize(sentences)` retornará uma lista de duas strings de frase:`['This is great !', 'Why not ?']`.
2. Qual é o tamanho do vocabulário se não filtrarmos nenhum token infrequente?

Discussão[183]

14.9 Pré-treinando BERT

Com o modelo BERT implementado em: numref: `sec_bert` e os exemplos de pré-treinamento gerados a partir do conjunto de dados WikiText-2 em Section 14.8, iremos pré-treinar o BERT no conjunto de dados WikiText-2 nesta seção.

```
import torch
from torch import nn
from d2l import torch as d2l
```

Para começar, carregamos o conjunto de dados WikiText-2 como minibatches de exemplos de pré-treinamento para modelagem de linguagem mascarada e previsão da próxima frase. O tamanho do lote é 512 e o comprimento máximo de uma sequência de entrada de BERT é 64. Observe que no modelo BERT original, o comprimento máximo é 512.

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

14.9.1 Pré-treinamento de BERT

O BERT original tem duas versões de tamanhos de modelo diferentes (Devlin et al., 2018). O modelo base ($\text{BERT}_{\text{BASE}}$) usa 12 camadas (blocos codificadores do transformador) com 768 unidades ocultas (tamanho oculto) e 12 cabeças de autoatenção. O modelo grande ($\text{BERT}_{\text{LARGE}}$) usa 24 camadas com 1024 unidades ocultas e 16 cabeças de autoatenção. Notavelmente, o primeiro tem 110 milhões de parâmetros, enquanto o último tem 340 milhões de parâmetros. Para demonstração com facilidade, definimos um pequeno BERT, usando 2 camadas, 128 unidades ocultas e 2 cabeças de autoatenção.

```
net = d2l.BERTModel(len(vocab), num_hiddens=128, norm_shape=[128],
                    ffn_num_input=128, ffn_num_hiddens=256, num_heads=2,
                    num_layers=2, dropout=0.2, key_size=128, query_size=128,
                    value_size=128, hid_in_features=128, mlm_in_features=128,
                    nsp_in_features=128)
```

(continues on next page)

[183] https://discuss.d2l.ai/t/1496

(continued from previous page)

```
devices = d2l.try_all_gpus()
loss = nn.CrossEntropyLoss()
```

Antes de definir o ciclo de treinamento, definimos uma função auxiliar `_get_batch_loss_bert`. Dado o fragmento de exemplos de treinamento, esta função calcula a perda tanto para a modelagem de linguagem mascarada quanto para as tarefas de previsão da próxima frase. Observe que a perda final do pré-treinamento de BERT é apenas a soma da perda de modelagem de linguagem mascarada e a próxima perda de previsão de frase.

```
#@save
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X,
                         segments_X, valid_lens_x,
                         pred_positions_X, mlm_weights_X,
                         mlm_Y, nsp_y):
    # Forward pass
    _, mlm_Y_hat, nsp_Y_hat = net(tokens_X, segments_X,
                                  valid_lens_x.reshape(-1),
                                  pred_positions_X)
    # Compute masked language model loss
    mlm_l = loss(mlm_Y_hat.reshape(-1, vocab_size), mlm_Y.reshape(-1)) *\
    mlm_weights_X.reshape(-1, 1)
    mlm_l = mlm_l.sum() / (mlm_weights_X.sum() + 1e-8)
    # Compute next sentence prediction loss
    nsp_l = loss(nsp_Y_hat, nsp_y)
    l = mlm_l + nsp_l
    return mlm_l, nsp_l, l
```

Invocando as duas funções auxiliares mencionadas, a seguinte função `train_bert` define o procedimento para pré-treinar BERT (`net`) no conjunto de dados WikiText-2 (`train_iter`). Treinar o BERT pode demorar muito. Em vez de especificar o número de épocas para treinamento como na função `train_ch13` (veja Section 13.1), a entrada `num_steps` da seguinte função especifica o número de etapas de iteração para treinamento.

```
def train_bert(train_iter, net, loss, vocab_size, devices, num_steps):
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.Adam(net.parameters(), lr=1e-3)
    step, timer = 0, d2l.Timer()
    animator = d2l.Animator(xlabel='step', ylabel='loss',
                            xlim=[1, num_steps], legend=['mlm', 'nsp'])
    # Sum of masked language modeling losses, sum of next sentence prediction
    # losses, no. of sentence pairs, count
    metric = d2l.Accumulator(4)
    num_steps_reached = False
    while step < num_steps and not num_steps_reached:
        for tokens_X, segments_X, valid_lens_x, pred_positions_X,\
            mlm_weights_X, mlm_Y, nsp_y in train_iter:
            tokens_X = tokens_X.to(devices[0])
            segments_X = segments_X.to(devices[0])
            valid_lens_x = valid_lens_x.to(devices[0])
            pred_positions_X = pred_positions_X.to(devices[0])
            mlm_weights_X = mlm_weights_X.to(devices[0])
            mlm_Y, nsp_y = mlm_Y.to(devices[0]), nsp_y.to(devices[0])
            trainer.zero_grad()
```

(continues on next page)

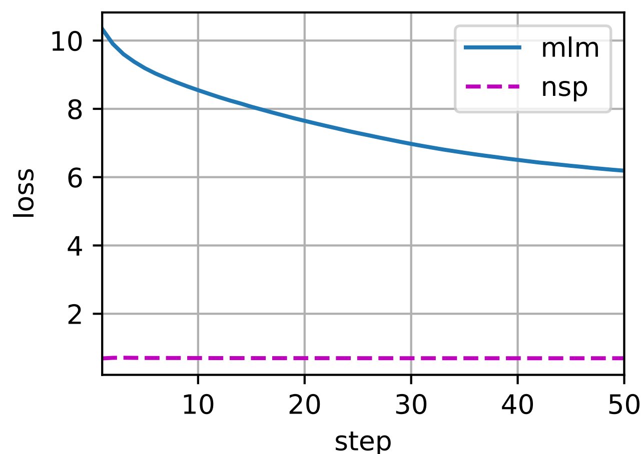
(continued from previous page)

```
            timer.start()
            mlm_l, nsp_l, l = _get_batch_loss_bert(
                net, loss, vocab_size, tokens_X, segments_X, valid_lens_x,
                pred_positions_X, mlm_weights_X, mlm_Y, nsp_y)
            l.backward()
            trainer.step()
            metric.add(mlm_l, nsp_l, tokens_X.shape[0], 1)
            timer.stop()
            animator.add(step + 1,
                         (metric[0] / metric[3], metric[1] / metric[3]))
            step += 1
            if step == num_steps:
                num_steps_reached = True
                break

    print(f'MLM loss {metric[0] / metric[3]:.3f}, '
          f'NSP loss {metric[1] / metric[3]:.3f}')
    print(f'{metric[2] / timer.sum():.1f} sentence pairs/sec on '
          f'{str(devices)}')
```

Podemos representar graficamente a perda de modelagem de linguagem mascarada e a perda de previsão da próxima frase durante o pré-treinamento de BERT.

```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

```
MLM loss 6.190, NSP loss 0.697
2947.7 sentence pairs/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



14.9.2 Representando Texto com BERT

Após o pré-treinamento de BERT, podemos usá-lo para representar um único texto, pares de texto ou qualquer token neles. A função a seguir retorna as representações BERT (`net`) para todos os tokens em `tokens_a` e`tokens_b`.

```
def get_bert_encoding(net, tokens_a, tokens_b=None):
    tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
    token_ids = torch.tensor(vocab[tokens], device=devices[0]).unsqueeze(0)
    segments = torch.tensor(segments, device=devices[0]).unsqueeze(0)
    valid_len = torch.tensor(len(tokens), device=devices[0]).unsqueeze(0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X
```

Considere a frase "um guindaste está voando". Lembre-se da representação de entrada de BERT conforme discutido em Section 14.7.4. Após inserir tokens especiais "<cls>" (usado para classificação) e "<cls>" (usado para separação), a sequência de entrada de BERT tem um comprimento de seis. Uma vez que zero é o índice de "<cls>" símbolo, `encoded_text[:, 0, :]` é a representação BERT de toda a sentença de entrada. Para avaliar o token de polissemia "guindaste", também imprimimos os três primeiros elementos da representação BERT do token.

```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# Tokens: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```

```
(torch.Size([1, 6, 128]),
 torch.Size([1, 128]),
 tensor([-1.4874, -0.0197, -1.0948], device='cuda:0', grad_fn=<SliceBackward>))
```

Agora considere um par de frases "veio um motorista de guindaste" e "acabou de sair". Da mesma forma, `encoded_pair[:, 0, :]` é o resultado codificado de todo o par de frases do BERT pré-treinado. Observe que os três primeiros elementos do token de polissemia "guindaste" são diferentes daqueles quando o contexto é diferente. Isso sustenta que as representações de BERT são sensíveis ao contexto.

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# Tokens: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

```
(torch.Size([1, 10, 128]),
 torch.Size([1, 128]),
 tensor([-1.3744, -0.7841, -1.4191], device='cuda:0', grad_fn=<SliceBackward>))
```

Em Chapter 15, vamos ajustar um modelo BERT pré-treinado para aplicativos de processamento de linguagem natural downstream.

14.9.3 Sumário

- O BERT original tem duas versões, onde o modelo básico tem 110 milhões de parâmetros e o modelo grande tem 340 milhões de parâmetros.
- Após o pré-treinamento de BERT, podemos usá-lo para representar texto único, pares de texto ou qualquer token neles.
- No experimento, o mesmo token tem diferentes representações de BERT quando seus contextos são diferentes. Isso sustenta que as representações de BERT são sensíveis ao contexto.

14.9.4 Exercícios

1. No experimento, podemos ver que a perda de modelagem da linguagem mascarada é significativamente maior do que a perda de previsão da próxima frase. Por quê?
2. Defina o comprimento máximo de uma sequência de entrada de BERT como 512 (igual ao modelo de BERT original). Use as configurações do modelo BERT original como $\text{BERT}_{\text{LARGE}}$. Você encontra algum erro ao executar esta seção? Por quê?

Discussão[184]

[184] https://discuss.d2l.ai/t/1497

15 Processamento de Linguagem Natural: Aplicações

Vimos como representar tokens de texto e treinar suas representações em Chapter 14. Essas representações de texto pré-treinadas podem ser fornecidas a vários modelos para diferentes tarefas de processamento de linguagem natural *downstream*.

Este livro não pretende cobrir as aplicações de processamento de linguagem natural de uma maneira abrangente. Nosso foco é *como aplicar a aprendizagem de representação (profunda) de idiomas para resolver problemas de processamento de linguagem natural*. No entanto, já discutimos várias aplicações de processamento de linguagem natural sem pré-treinamento nos capítulos anteriores, apenas para explicar arquiteturas de aprendizado profundo. Por exemplo, em Chapter 8, contamos com RNNs para projetar modelos de linguagem para gerar textos semelhantes a novelas. Em Chapter 9 e Chapter 10, também projetamos modelos baseados em RNNs e mecanismos de atenção para tradução automática. Dadas as representações de texto pré-treinadas, neste capítulo, consideraremos mais duas tarefas de processamento de linguagem natural *downstream*: análise de sentimento e inferência de linguagem natural. Estes são aplicativos de processamento de linguagem natural populares e representativos: o primeiro analisa um único texto e o último analisa as relações de pares de texto.

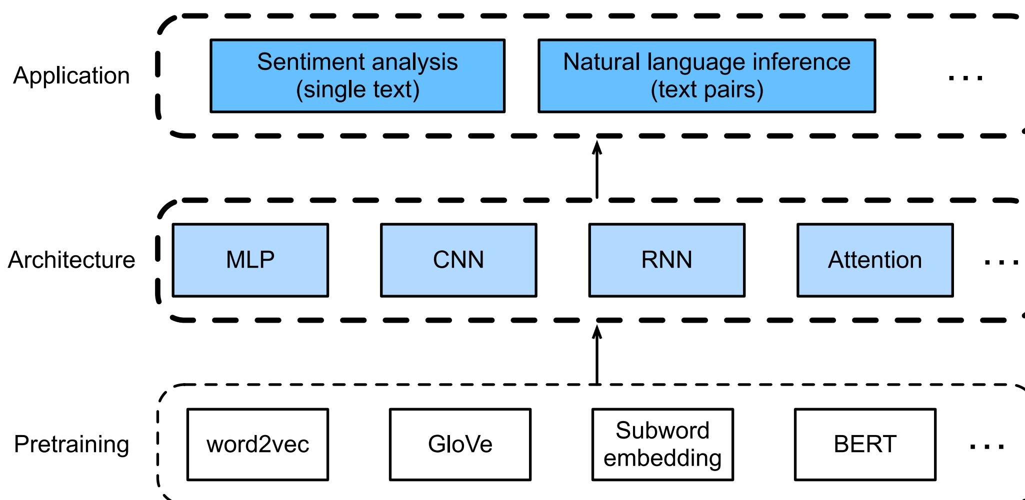


Fig. 15.1: As representações de texto pré-treinadas podem ser alimentadas para várias arquiteturas de *deep learning* para diferentes aplicações de processamento de linguagem natural *downstream*. Este capítulo enfoca como projetar modelos para diferentes aplicações de processamento de linguagem natural *downstream*.

Conforme descrito em Fig. 15.1, este capítulo se concentra na descrição das ideias básicas de

projeto de modelos de processamento de linguagem natural usando diferentes tipos de arquiteturas de aprendizado profundo, como MLPs, CNNs, RNNs e atenção. Embora seja possível combinar qualquer representação de texto pré-treinada com qualquer arquitetura para qualquer tarefa de processamento de linguagem natural *downstream* em Fig. 15.1, selecionamos algumas combinações representativas. Especificamente, exploraremos arquiteturas populares baseadas em RNNs e CNNs para análise de sentimento. Para inferência de linguagem natural, escolhemos atenção e MLPs para demonstrar como analisar pares de texto. No final, apresentamos como ajustar um modelo BERT pré-treinado para uma ampla gama de aplicações de processamento de linguagem natural, como em um nível de sequência (classificação de texto único e classificação de par de texto) e um nível de *token* (marcação de texto e resposta a perguntas). Como um caso empírico concreto, faremos o ajuste fino do BERT para processamento de linguagem natural.

Como apresentamos em Section 14.7, BERT requer mudanças mínimas de arquitetura para uma ampla gama de aplicativos de processamento de linguagem natural. No entanto, esse benefício vem com o custo de um ajuste fino um grande número de parâmetros BERT para as aplicações *downstream*. Quando o espaço ou o tempo são limitados, aqueles modelos elaborados com base em MLPs, CNNs, RNNs e atenção são mais viáveis. A seguir, começamos pelo aplicativo de análise de sentimento e ilustrar o design do modelo baseado em RNNs e CNNs, respectivamente.

15.1 Análise de Sentimentos e o *Dataset*

A classificação de texto é uma tarefa comum no processamento de linguagem natural, que transforma uma sequência de texto de comprimento indefinido em uma categoria de texto. É semelhante à classificação de imagem, o aplicativo usado com mais frequência neste livro, por exemplo, Section 18.9. A única diferença é que, em vez de uma imagem, o exemplo da classificação de texto é uma frase de texto.

Esta seção se concentrará no carregamento de dados para uma das subquestões neste campo: usar a classificação de sentimento do texto para analisar as emoções do autor do texto. Esse problema também é chamado de análise de sentimento e tem uma ampla gama de aplicações. Por exemplo, podemos analisar resenhas de usuários de produtos para obter estatísticas de satisfação do usuário ou analisar os sentimentos do usuário sobre as condições de mercado e usá-las para prever tendências futuras.

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

15.1.1 O Dataset de Análise de Sentimento

Usamos o *dataset* Large Movie Review Dataset[185] de Stanford para análise de sentimento. Este conjunto de dados é dividido em dois conjuntos de dados para fins de treinamento e teste, cada um contendo 25.000 resenhas de filmes baixadas da IMDb. Em cada conjunto de dados, o número de comentários rotulados como “positivos” e “negativos” é igual.

[185] https://ai.stanford.edu/~amaas/data/sentiment/

Lendo o *Dataset*

Primeiro, baixamos esse *dataset* para o caminho "../data" e o extraímos para "../data/aclImdb".

```
#@save
d2l.DATA_HUB['aclImdb'] = (
    'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz',
    '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```

Em seguida, leia os conjuntos de dados de treinamento e teste. Cada exemplo é uma revisão e seu rótulo correspondente: 1 indica "positivo" e 0 indica "negativo".

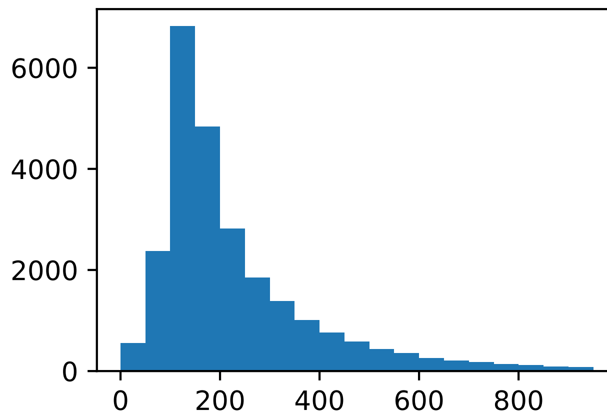
```
#@save
def read_imdb(data_dir, is_train):
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

```
# trainings: 25000
label: 1 review: Normally the best way to annoy me in a film is to include so
label: 1 review: The Bible teaches us that the love of money is the root of a
label: 1 review: Being someone who lists Night of the Living Dead at number t
```

Tokenização e Vocabulário

Usamos uma palavra como token e, em seguida, criamos um dicionário com base no conjunto de dados de treinamento.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])

d2l.set_figsize()
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```



Preenchimento com o mesmo comprimento

Como as resenhas têm durações diferentes, não podem ser combinadas diretamente em minibatches. Aqui, fixamos o comprimento de cada comentário em 500, truncando ou adicionando índices “<unk>”.

```
num_steps = 500  # sequence length
train_features = torch.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
torch.Size([25000, 500])
```

Criando o Iterador de Dados

Agora, criaremos um iterador de dados. Cada iteração retornará um minibatch de dados.

```
train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])), 64)

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('# batches:', len(train_iter))
```

```
X: torch.Size([64, 500]) , y: torch.Size([64])
# batches: 391
```

15.1.2 Juntando Tudo

Por último, salvaremos uma função `load_data_imdb` em`d2l`, que retorna o vocabulário e iteradores de dados.

```
#@save
def load_data_imdb(batch_size, num_steps=500):
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])),
                                batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(test_data[1])),
                               batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

15.1.3 Resumo

- A classificação de texto pode classificar uma sequência de texto em uma categoria.
- Para classificar um sentimento de texto, carregamos um conjunto de dados IMDb e tokenizar suas palavras. Em seguida, preenchemos a sequência de texto para revisões curtas e criamos um iterador de dados.

15.1.4 Exercícios

1. Descubra um conjunto de dados de linguagem natural diferente (como revisões da Amazon[186]) e crie uma função data_loader semelhante como `load_data_imdb`.

Discussões[187]

15.2 Análise de Sentimento: Usando Redes Neurais Recorrentes

Semelhante a sinônimos e analogias de pesquisa, a classificação de texto também é uma aplicação posterior de incorporação de palavras. Nesta seção, vamos aplicar vetores de palavras pré-treinados (GloVe) e redes neurais recorrentes bidirecionais com múltiplas camadas ocultas (Maas et al., 2011), como mostrado em Fig. 15.2.1. Usaremos o modelo para determinar se uma sequência de texto de comprimento indefinido contém emoção positiva ou negativa.

[186] https://snap.stanford.edu/data/web-Amazon.html
[187] https://discuss.d2l.ai/t/1387

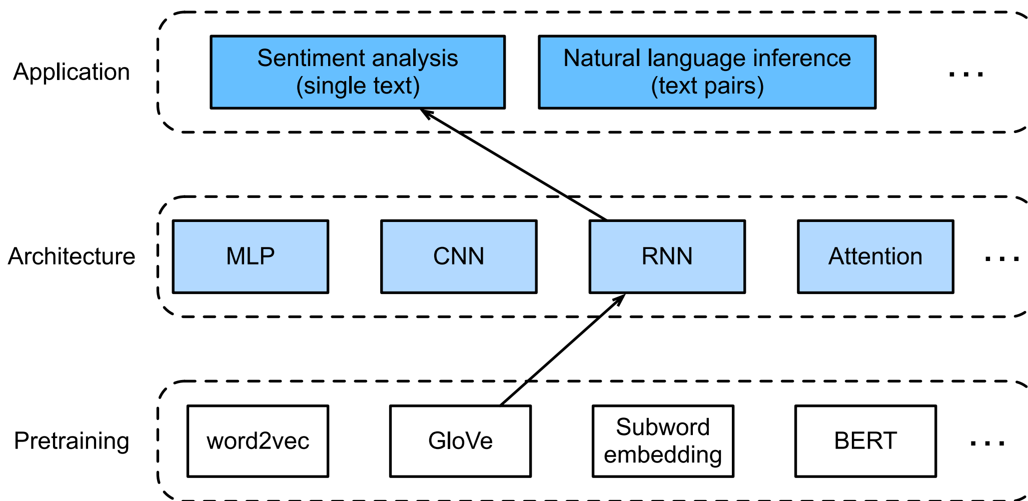


Fig. 15.2.1: Esta seção alimenta o GloVe pré-treinado para uma arquitetura baseada em RNN para análise de sentimento.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://ai.stanford.edu/~amaas/data/sentiment/
↪aclImdb_v1.tar.gz...
```

15.2.1 Usando um Modelo de Rede Neural Recorrente

Neste modelo, cada palavra obtém primeiro um vetor de recurso da camada de incorporação. Em seguida, codificamos ainda mais a sequência de recursos usando uma rede neural recorrente bidirecional para obter as informações da sequência. Por fim, transformamos as informações da sequência codificada para a saída por meio da camada totalmente conectada. Especificamente, podemos concatenar estados ocultos de memória de longo-curto prazo bidirecionais na etapa de tempo inicial e etapa de tempo final e passá-la para a classificação da camada de saída como informação de sequência de recurso codificada. Na classe `BiRNN` implementada abaixo, a instânciaEmbedding é a camada de incorporação, a instância `LSTM` é a camada oculta para codificação de sequência e a instânciaDense é a camada de saída para os resultados de classificação gerados.

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set `bidirectional` to True to get a bidirectional recurrent neural
        # network
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
                               bidirectional=True)
        self.decoder = nn.Linear(4*num_hiddens, 2)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of words). Because LSTM
        # needs to use sequence as the first dimension, the input is
        # transformed and the word feature is then extracted. The output shape
        # is (no. of words, batch size, word vector dimension).
        embeddings = self.embedding(inputs.T)
        # Since the input (embeddings) is the only argument passed into
        # nn.LSTM, both h_0 and c_0 default to zero.
        # we only use the hidden states of the last hidden layer
        # at different time step (outputs). The shape of `outputs` is
        # (no. of words, batch size, 2 * no. of hidden units).
        self.encoder.flatten_parameters()
        outputs, _ = self.encoder(embeddings)
        # Concatenate the hidden states of the initial time step and final
        # time step to use as the input of the fully connected layer. Its
        # shape is (batch size, 4 * no. of hidden units)
        encoding = torch.cat((outputs[0], outputs[-1]), dim=1)
        outs = self.decoder(encoding)
        return outs
```

Criando uma rede neural recorrente bidirecional com duas camadas ocultas.

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
    if type(m) == nn.LSTM:
        for param in m._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(m._parameters[param])
net.apply(init_weights);
```

Carregando Vetores de Palavras Pré-treinados

Como o conjunto de dados de treinamento para classificação de sentimento não é muito grande, para lidar com o *overfitting*, usaremos diretamente vetores de palavras pré-treinados em um corpus maior como vetores de características de todas as palavras. Aqui, carregamos um vetor de palavras GloVe de 100 dimensões para cada palavra do `vocabulário` do dicionário.

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

```
Downloading ../data/glove.6B.100d.zip from http://d2l-data.s3-accelerate.amazonaws.com/glove.
↪6B.100d.zip...
```

Consultando os vetores de palavras que estão em nosso vocabulário.

```
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```

```
torch.Size([49346, 100])
```

Em seguida, usaremos esses vetores de palavras como vetores de características para cada palavra nas revisões. Observe que as dimensões dos vetores de palavras pré-treinados precisam ser consistentes com o tamanho de saída da camada de incorporação `embed_size` no modelo criado. Além disso, não atualizamos mais esses vetores de palavras durante o treinamento.

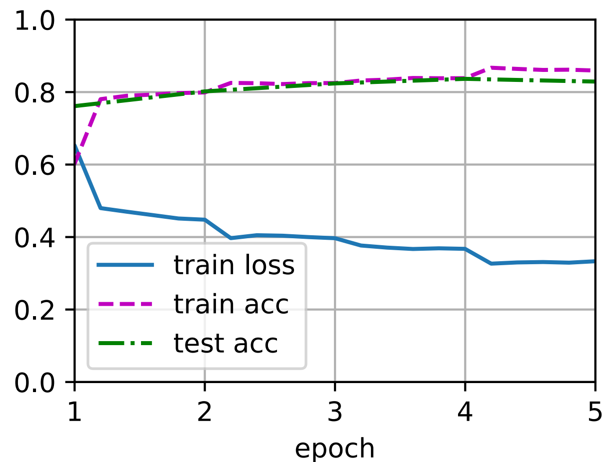
```
net.embedding.weight.data.copy_(embeds)
net.embedding.weight.requires_grad = False
```

Treinamento e Avaliação do Modelo

Agora podemos começar a treinar.

```
lr, num_epochs = 0.01, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.333, train acc 0.860, test acc 0.829
749.0 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Finalmente, definir a função de previsão.

```
#@save
def predict_sentiment(net, vocab, sentence):
    sentence = torch.tensor(vocab[sentence.split()], device=d2l.try_gpu())
    label = torch.argmax(net(sentence.reshape(1, -1)), dim=1)
    return 'positive' if label == 1 else 'negative'
```

Em seguida, usamos o modelo treinado para classificar os sentimentos de duas frases simples.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

15.2.2 Resumo

- A classificação de texto transforma uma sequência de texto de comprimento indefinido em uma categoria de texto. Esta é uma aplicação *downstream* de incorporação de palavras.
- Podemos aplicar vetores de palavras pré-treinados e redes neurais recorrentes para classificar as emoções em um texto.

15.2.3 Exercícios

1. Aumente o número de épocas. Que taxa de precisão você pode alcançar nos conjuntos de dados de treinamento e teste? Que tal tentar reajustar outros hiperparâmetros?
2. O uso de vetores de palavras pré-treinados maiores, como vetores de palavras GloVe 300-dimensionais, melhorará a acurácia da classificação?
3. Podemos melhorar a acurácia da classificação usando a ferramenta de tokenização de palavras spaCy? Você precisa instalar spaCy: `pip install spacy` e instalar o pacote em inglês: `python -m spacy download en`. No código, primeiro importe spacy: `import spacy`. Em seguida, carregue o pacote spacy em inglês: `spacy_en = spacy.load ('en')`. Finalmente, defina a função `def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` e substitua a função original de `tokenizer`. Deve-se notar que o vetor de palavras do GloVe usa "-" para conectar cada palavra ao armazenar frases nominais. Por exemplo, a frase "new york" é representada como "new-york" no GloVe. Depois de usar a tokenização spaCy, "new york" pode ser armazenado como "new york".

Discussões[188]

15.3 Análise de Sentimento: Usando Redes Neurais Convolucionais

Em Chapter 6, exploramos como processar dados de imagens bidimensionais com redes neurais convolucionais bidimensionais. Nos modelos de linguagem anteriores e nas tarefas de classificação de texto, tratamos os dados de texto como uma série temporal com apenas uma dimensão e, naturalmente, usamos redes neurais recorrentes para processar esses dados. Na verdade, também podemos tratar texto como uma imagem unidimensional, para que possamos usar redes neurais convolucionais unidimensionais para capturar associações entre palavras adjacentes. Conforme descrito em Fig. 15.3.1 Esta seção descreve uma abordagem inovadora para aplicar Redes neurais convolucionais para análise de sentimento: textCNN (Kim, 2014).

[188] https://discuss.d2l.ai/t/1424

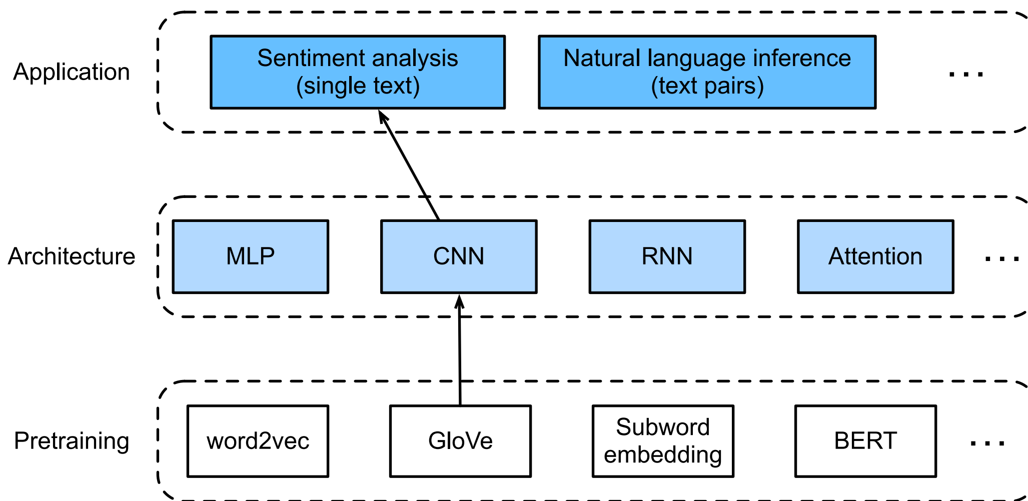


Fig. 15.3.1: Esta seção alimenta o GloVe pré-treinado para uma arquitetura baseada em CNN para análise de sentimento.

Primeiro, importe os pacotes e módulos necessários para o experimento.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

15.3.1 Camada Convolucional Unidimensional

Antes de apresentar o modelo, vamos explicar como funciona uma camada convolucional unidimensional. Como uma camada convolucional bidimensional, uma camada convolucional unidimensional usa uma operação de correlação cruzada unidimensional. Na operação de correlação cruzada unidimensional, a janela de convolução começa do lado esquerdo da matriz de entrada e desliza na matriz de entrada da esquerda para a direita sucessivamente. Quando a janela de convolução desliza para uma determinada posição, o subarray de entrada na janela e o array kernel são multiplicados e somados por elemento para obter o elemento no local correspondente no array de saída. Conforme mostrado em Fig. 15.3.2, a entrada é uma matriz unidimensional com largura 7 e a largura da matriz do kernel é 2. Como podemos ver, a largura de saída é $7-2+1=6$ e o primeiro elemento é obtido executando a multiplicação por elemento no submatriz de entrada mais à esquerda com uma largura de 2 e o array *kernel* e, em seguida, somando os resultados.

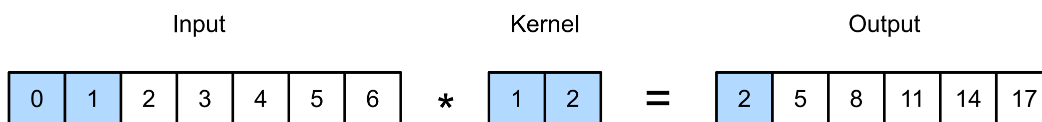


Fig. 15.3.2: Operação de correlação cruzada unidimensional. As partes sombreadas são o primeiro elemento de saída, bem como os elementos de matriz de entrada e *kernel* usados em seu cálculo: $0 \times 1 + 1 \times 2 = 2$.

Em seguida, implementamos a correlação cruzada unidimensional na função `corr1d`. Ele aceita a matriz de entrada `X` e a matriz de *kernel* `K` e produz a matriz `Y`.

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

Agora, iremos reproduzir os resultados da operação de correlação cruzada unidimensional em Fig. 15.3.2.

```
X, K = torch.tensor([0, 1, 2, 3, 4, 5, 6]), torch.tensor([1, 2])
corr1d(X, K)
```

```
tensor([ 2.,  5.,  8., 11., 14., 17.])
```

A operação de correlação cruzada unidimensional para vários canais de entrada também é semelhante à operação de correlação cruzada bidimensional para vários canais de entrada. Em cada canal, ela executa a operação de correlação cruzada unidimensional no *kernel* e sua entrada correspondente e adiciona os resultados dos canais para obter a saída. Fig. 15.3.3 mostra uma operação de correlação cruzada unidimensional com três canais de entrada.

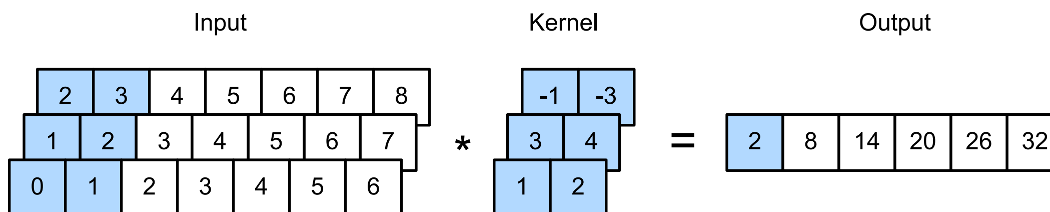


Fig. 15.3.3: Operação de correlação cruzada unidimensional com três canais de entrada. As partes sombreadas são o primeiro elemento de saída, bem como os elementos da matriz de entrada e *kernel* usados em seu cálculo: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

Agora, reproduzimos os resultados da operação de correlação cruzada unidimensional com o canal de entrada múltipla em Fig. 15.3.3.

```
def corr1d_multi_in(X, K):
    # First, we traverse along the 0th dimension (channel dimension) of `X`
    # and `K`. Then, we add them together by using * to turn the result list
    # into a positional argument of the `add_n` function
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = torch.tensor([[0, 1, 2, 3, 4, 5, 6],
              [1, 2, 3, 4, 5, 6, 7],
              [2, 3, 4, 5, 6, 7, 8]])
K = torch.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
tensor([ 2.,  8., 14., 20., 26., 32.])
```

A definição de uma operação de correlação cruzada bidimensional nos diz que uma operação de correlação cruzada unidimensional com múltiplos canais de entrada pode ser considerada como uma operação de correlação cruzada bidimensional com um único canal de entrada. Conforme mostrado em Fig. 15.3.4, também podemos apresentar a operação de correlação cruzada unidimensional com múltiplos canais de entrada em Fig. 15.3.3 como a operação de correlação cruzada bidimensional equivalente com um único canal de entrada. Aqui, a altura do *kernel* é igual à altura da entrada.

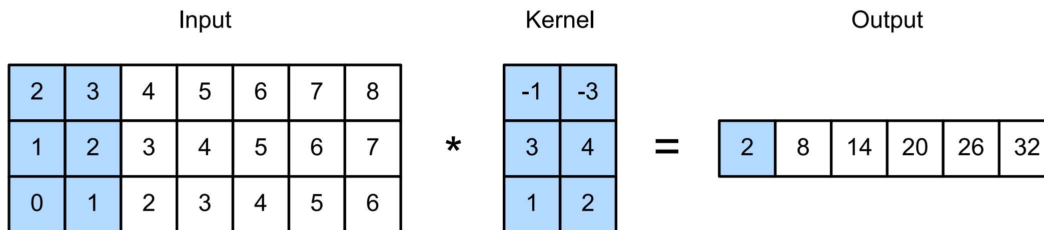


Fig. 15.3.4: Operação de correlação cruzada bidimensional com um único canal de entrada. As partes destacadas são o primeiro elemento de saída e os elementos de matriz de entrada e *kernel* usados em seu cálculo: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Ambas as saídas em Fig. 15.3.2 e Fig. 15.3.3 têm apenas um canal. Nós discutimos como especificar múltiplos canais de saída em uma camada convolucional bidimensional em Section 6.4. Da mesma forma, também podemos especificar vários canais de saída na camada convolucional unidimensional para estender os parâmetros do modelo na camada convolucional.

15.3.2 Camada de Pooling Máximo ao Longo do Tempo

Da mesma forma, temos uma camada de *pooling* unidimensional. A camada de *pooling* máximo ao longo do tempo usada no TextCNN na verdade corresponde a uma camada de *pooling* global unidimensional máxima. Assumindo que a entrada contém vários canais, e cada canal consiste em valores em intervalos de tempo diferentes, a saída de cada canal será o maior valor de todos os intervalos de tempo no canal. Portanto, a entrada da camada de *pooling* max-over-time pode ter diferentes intervalos de tempo em cada canal.

Para melhorar o desempenho da computação, geralmente combinamos exemplos de tempo de diferentes comprimentos em um minibatch e tornamos os comprimentos de cada exemplo de tempo no lote consistentes, acrescentando caracteres especiais (como 0) ao final dos exemplos mais curtos. Naturalmente, os caracteres especiais adicionados não têm significado intrínseco. Como o objetivo principal da camada de *pooling max-over-time* é capturar os recursos mais importantes de temporização, ela geralmente permite que o modelo não seja afetado pelos caracteres adicionados manualmente.

15.3.3 O Modelo TextCNN

O TextCNN usa principalmente uma camada convolucional unidimensional e uma camada de *pooling* máximo ao longo do tempo. Suponha que a sequência de texto de entrada consista em n palavras e cada palavra seja representada por um vetor de palavra de dimensão d. Então, o exemplo de entrada tem uma largura de n, uma altura de 1 e d canais de entrada. O cálculo de textCNN pode ser dividido principalmente nas seguintes etapas:

1. Definindo vários *kernels* de convolução unidimensionais e usando-os para realizar cálculos de convolução nas entradas. Os núcleos de convolução com larguras diferentes podem capturar a correlação de diferentes números de palavras adjacentes.
2. Executando o *pooling* máximo ao longo do tempo em todos os canais de saída e, a seguir, concatene os valores de saída do *pooling* desses canais em um vetor.
3. O vetor concatenado é transformado na saída de cada categoria por meio da camada totalmente conectada. Uma camada de eliminação pode ser usada nesta etapa para lidar com o *overfitting*.

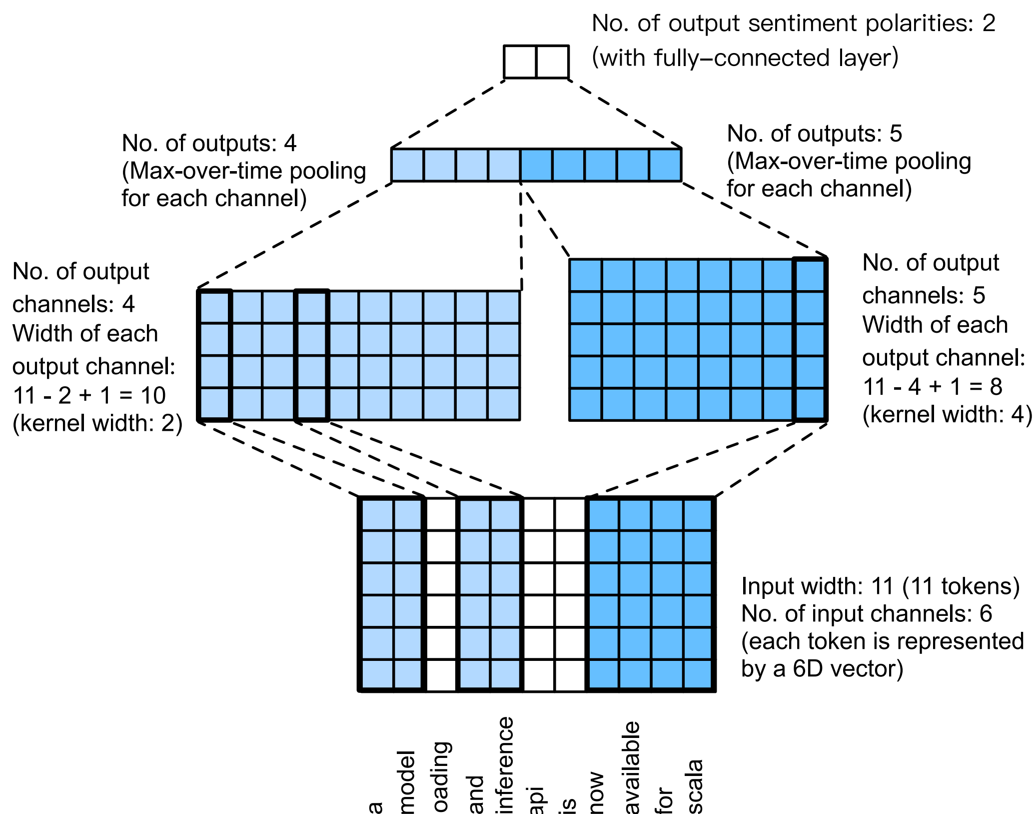


Fig. 15.3.5: Design de TextCNN.

Fig. 15.3.5 dá um exemplo para ilustrar o textCNN. A entrada aqui é uma frase com 11 palavras, com cada palavra representada por um vetor de palavra de 6 dimensões. Portanto, a sequência de entrada tem uma largura de 11 e 6 canais de entrada. Assumimos que há dois núcleos de convolução unidimensionais com larguras de 2 e 4, e 4 e 5 canais de saída, respectivamente. Portanto, após o cálculo da convolução unidimensional, a largura dos quatro canais de saída é $11-2+1 = 10$, enquanto a largura dos outros cinco canais é $11 - 4 + 1 = 8$. Mesmo que a largura de cada canal seja diferente, ainda podemos executar o *pooling* máximo ao longo do tempo para cada canal e

concatenar as saídas do pooling dos 9 canais em um vetor de 9 dimensões. Finalmente, usamos uma camada totalmente conectada para transformar o vetor 9-dimensional em uma saída bidimensional: previsões de sentimento positivo e sentimento negativo.

A seguir, implementaremos um modelo textCNN. Em comparação com a seção anterior, além de substituir a rede neural recorrente por uma camada convolucional unidimensional, aqui usamos duas camadas de incorporação, uma com peso fixo e outra que participa do treinamento.

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # The embedding layer does not participate in training
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # The max-over-time pooling layer has no weight, so it can share an
        # instance
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k))

    def forward(self, inputs):
        # Concatenate the output of two embedding layers with shape of
        # (batch size, no. of words, word vector dimension) by word vector
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(inputs)), dim=2)
        # According to the input format required by Conv1d, the word vector
        # dimension, that is, the channel dimension of the one-dimensional
        # convolutional layer, is transformed into the previous dimension
        embeddings = embeddings.permute(0, 2, 1)
        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, a tensor with the shape of (batch size, channel size, 1)
        # can be obtained. Use the flatten function to remove the last
        # dimension and then concatenate on the channel dimension
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim=-1)
            for conv in self.convs], dim=1)
        # After applying the dropout method, use a fully connected layer to
        # obtain the output
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Criando uma instância TextCNN. Possui 3 camadas convolucionais com larguras de *kernel* de 3, 4 e 5, todas com 100 canais de saída.

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)

def init_weights(m):
    if type(m) in (nn.Linear, nn.Conv1d):
```

(continues on next page)

(continued from previous page)

```
        nn.init.xavier_uniform_(m.weight)

net.apply(init_weights);
```

Carregando Vetores de Palavras Pré-treinados

Como na seção anterior, carregamos os vetores de palavras GloVe 100-dimensionais pré-treinados e inicializamos as camadas de incorporação `embedding` e `constant_embedding`. Aqui, o primeiro participa do treinamento, enquanto o segundo tem peso fixo.

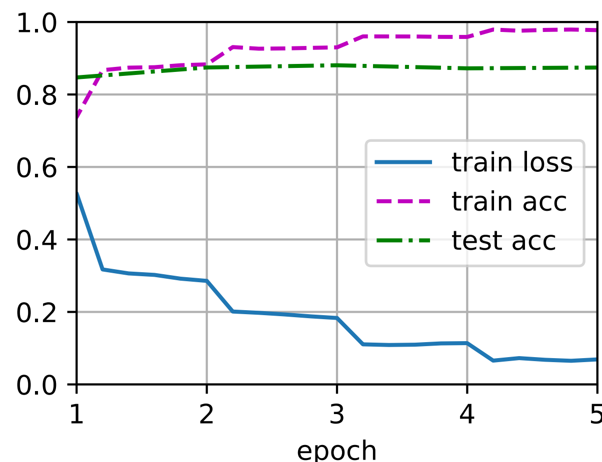
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

Treinamento e Avaliação do Modelo

Agora podemos treinar o modelo.

```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.069, train acc 0.978, test acc 0.874
2703.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



A seguir, usamos o modelo treinado para classificar os sentimentos de duas frases simples.

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

15.3.4 Resumo

- Podemos usar convolução unidimensional para processar e analisar dados de tempo.
- Uma operação de correlação cruzada unidimensional com múltiplos canais de entrada pode ser considerada como uma operação de correlação cruzada bidimensional com um único canal de entrada.
- A entrada da camada de *pooling* máximo ao longo do tempo pode ter diferentes números de etapas de tempo em cada canal.
- TextCNN usa principalmente uma camada convolucional unidimensional e uma camada de *pooling* máximo ao longo do tempo.

15.3.5 Exercícios

1. Ajuste os hiperparâmetros e compare os dois métodos de análise de sentimento, usando redes neurais recorrentes e usando redes neurais convolucionais, no que diz respeito à precisão e eficiência operacional.
2. Você pode melhorar ainda mais a precisão do modelo no conjunto de teste usando os três métodos apresentados na seção anterior: ajuste de hiperparâmetros, uso de vetores de palavras pré-treinados maiores e uso da ferramenta de tokenização de palavras spaCy?
3. Para quais outras tarefas de processamento de linguagem natural você pode usar o textCNN?
4. Adicione codificação posicional nas representações de entrada. Isso melhora o desempenho?

Discussões[189]

15.4 Inferência de Linguagem Natural e o *Dataset*

Em Section 15.1, discutimos o problema da análise de sentimento. Esta tarefa visa classificar uma única sequência de texto em categorias predefinidas, como um conjunto de polaridades de sentimento. No entanto, quando há a necessidade de decidir se uma frase pode ser inferida de outra ou eliminar a redundância identificando frases semanticamente equivalentes, saber classificar uma sequência de texto é insuficiente. Em vez disso, precisamos ser capazes de raciocinar sobre pares de sequências de texto.

[189] https://discuss.d2l.ai/t/1425

15.4.1 Inferência de Linguagem Natural

Inferência de linguagem natural estuda se uma *hipótese* pode ser inferida de uma *premissa*, onde ambas são uma sequência de texto. Em outras palavras, a inferência de linguagem natural determina a relação lógica entre um par de sequências de texto. Esses relacionamentos geralmente se enquadram em três tipos:

- *Implicação*: a hipótese pode ser inferida a partir da premissa.
- *Contradição*: a negação da hipótese pode ser inferida a partir da premissa.
- *Neutro*: todos os outros casos.

A inferência de linguagem natural também é conhecida como a tarefa de reconhecimento de vinculação textual. Por exemplo, o par a seguir será rotulado como *implicação* porque "mostrar afeto" na hipótese pode ser inferido de "abraçar um ao outro" na premissa.

> Premissa: Duas mulheres estão se abraçando.
>
> Hipótese: Duas mulheres estão demonstrando afeto.

A seguir está um exemplo de *contradição*, pois "executando o exemplo de codificação" indica "não dormindo" em vez de "dormindo".

> Premissa: Um homem está executando o exemplo de codificação do *Dive into Deep Learning*.
>
> Hipótese: O homem está dormindo.

O terceiro exemplo mostra uma relação de *neutralidade* porque nem "famoso" nem "não famoso" podem ser inferidos do fato de que "estão se apresentando para nós".

> Premissa: Os músicos estão se apresentando para nós.
>
> Hipótese: Os músicos são famosos.

A inferência da linguagem natural tem sido um tópico central para a compreensão da linguagem natural. Desfruta de uma ampla gama de aplicações, desde recuperação de informações para resposta a perguntas de domínio aberto. Para estudar esse problema, começaremos investigando um popular conjunto de dados de referência de inferência em linguagem natural.

15.4.2 Conjunto de dados Stanford Natural Language Inference (SNLI)

Stanford Natural Language Inference (SNLI) Corpus é uma coleção de mais de 500.000 pares de frases em inglês rotulados (Bowman et al., 2015). Baixamos e armazenamos o conjunto de dados SNLI extraído no caminho `../data/snli_1.0`.

```
import os
import re
import torch
from torch import nn
from d2l import torch as d2l

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')
```

(continues on next page)

(continued from previous page)

```
data_dir = d2l.download_extract('SNLI')
```

Lendo o *Dataset*

O conjunto de dados SNLI original contém informações muito mais ricas do que realmente precisamos em nossos experimentos. Assim, definimos uma função `read_snli` para extrair apenas parte do conjunto de dados e, em seguida, retornar listas de premissas, hipóteses e seus rótulos.

```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and labels."""
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with space
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```

Agora, vamos imprimir os primeiros 3 pares de premissa e hipótese, bem como seus rótulos ("0", "1" e "2" correspondem a "implicação", "contradição" e "neutro", respectivamente).

```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```

```
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is training his horse for a competition .
label: 2
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is at a diner , ordering an omelette .
label: 1
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is outdoors , on a horse .
label: 0
```

O conjunto de treinamento tem cerca de 550.000 pares, e o conjunto de teste tem cerca de 10.000 pares. O seguinte mostra que os três rótulos "implicação", "contradição" e "neutro" são equilibrados em o conjunto de treinamento e o conjunto de teste.

```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```

```
[183416, 183187, 182764]
[3368, 3237, 3219]
```

Definindo uma Classe para Carregar o *Dataset*

Abaixo, definimos uma classe para carregar o dataset SNLI herdando da classe `Dataset` no Gluon. O argumento `num_steps` no construtor de classe especifica o comprimento de uma sequência de texto para que cada minibatch de sequências tenha a mesma forma. Em outras palavras, tokens após os primeiros `num_steps` em uma sequência mais longa são cortados, enquanto tokens especiais “& lt; pad & gt;” serão anexados a sequências mais curtas até que seu comprimento se torne `num_steps`. Implementando a função `__getitem__`, podemos acessar arbitrariamente a premissa, hipótese e rótulo com o índice `idx`.

```
#@save
class SNLIDataset(torch.utils.data.Dataset):
    """A customized dataset to load the SNLI dataset."""
    def __init__(self, dataset, num_steps, vocab=None):
        self.num_steps = num_steps
        all_premise_tokens = d2l.tokenize(dataset[0])
        all_hypothesis_tokens = d2l.tokenize(dataset[1])
        if vocab is None:
            self.vocab = d2l.Vocab(all_premise_tokens + all_hypothesis_tokens,
                                   min_freq=5, reserved_tokens=['<pad>'])
        else:
            self.vocab = vocab
        self.premises = self._pad(all_premise_tokens)
        self.hypotheses = self._pad(all_hypothesis_tokens)
        self.labels = torch.tensor(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples')

    def _pad(self, lines):
        return torch.tensor([d2l.truncate_pad(
            self.vocab[line], self.num_steps, self.vocab['<pad>'])
                         for line in lines])

    def __getitem__(self, idx):
        return (self.premises[idx], self.hypotheses[idx]), self.labels[idx]

    def __len__(self):
        return len(self.premises)
```

Juntando Tudo

Agora podemos invocar a função `read_snli` e a classe `SNLIDataset` para baixar o conjunto de dados SNLI e retornar instâncias de `DataLoader` para ambos os conjuntos de treinamento e teste, junto com o vocabulário do conjunto de treinamento. Vale ressaltar que devemos utilizar o vocabulário construído a partir do conjunto de treinamento como aquele do conjunto de teste. Como resultado, qualquer novo *token* do conjunto de teste será desconhecido para o modelo treinado no conjunto de treinamento.

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """Download the SNLI dataset and return data iterators and vocabulary."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vocab)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True,
                                             num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                            shuffle=False,
                                            num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab
```

Aqui, definimos o tamanho do lote em 128 e o comprimento da sequência em 50, e invoque a função `load_data_snli` para obter os iteradores de dados e vocabulário. Em seguida, imprimimos o tamanho do vocabulário.

```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```

```
read 549367 examples
read 9824 examples
```

```
18678
```

Agora imprimimos a forma do primeiro minibatch. Ao contrário da análise de sentimento, temos 2 entradas `X[0]` e `X[1]` representando pares de premissas e hipóteses.

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

```
torch.Size([128, 50])
torch.Size([128, 50])
torch.Size([128])
```

15.4.3 Resumo

- A inferência em linguagem natural estuda se uma hipótese pode ser inferida de uma premissa, onde ambas são uma sequência de texto.
- Na inferência em linguagem natural, as relações entre premissas e hipóteses incluem implicação, contradição e neutro.
- *Stanford Natural Language Inference* (SNLI) Corpus é um popular *dataset* de referência de inferência em linguagem natural.

15.4.4 Exercícios

1. A tradução automática há muito é avaliada com base na correspondência superficial de n-grama entre uma tradução de saída e uma tradução de verdade. Você pode criar uma medida para avaliar os resultados da tradução automática usando a inferência de linguagem natural?
2. Como podemos alterar os hiperparâmetros para reduzir o tamanho do vocabulário?

Discussões[190]

15.5 Inferência de Linguagem Natural: Usando a Atenção

Introduzimos a tarefa de inferência em linguagem natural e o conjunto de dados SNLI em Section 15.4. Em vista de muitos modelos baseados em arquiteturas complexas e profundas, Parikh et al. proposto para abordar a inferência de linguagem natural com mecanismos de atenção e chamou-o de "modelo de atenção decomposto" (Parikh et al., 2016). Isso resulta em um modelo sem camadas recorrentes ou convolucionais, alcançando o melhor resultado no momento no conjunto de dados SNLI com muito menos parâmetros. Nesta seção, iremos descrever e implementar este método baseado em atenção (com MLPs) para inferência de linguagem natural, conforme descrito em `fig_nlp-map-nli -ention`.

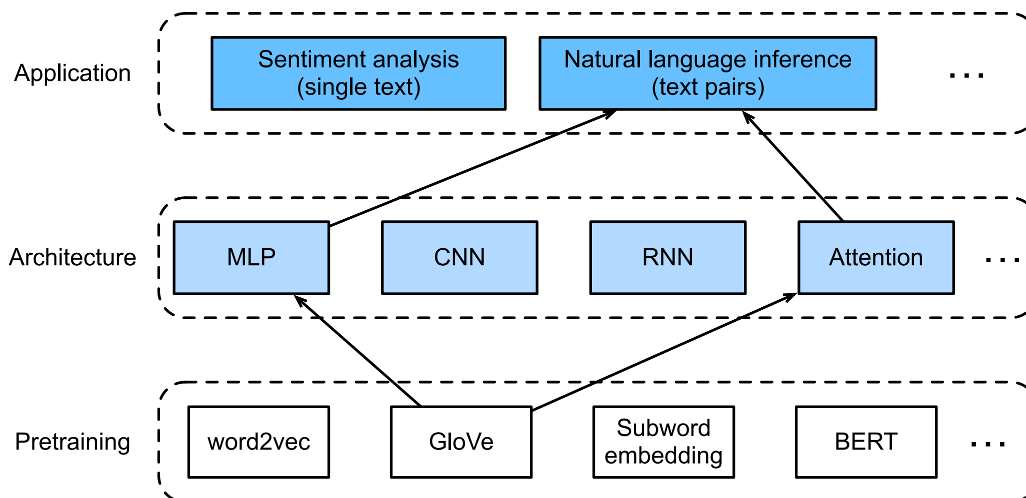


Fig. 15.5.1: Esta seção alimenta o GloVe pré-treinado para uma arquitetura baseada em atenção e MLPs para inferência de linguagem natural.

[190] https://discuss.d2l.ai/t/1388

15.5.1 O Modelo

Mais simples do que preservar a ordem das palavras em premissas e hipóteses, podemos apenas alinhar as palavras em uma sequência de texto com todas as palavras na outra e vice-versa, em seguida, compare e agregue essas informações para prever as relações lógicas entre premissas e hipóteses. Semelhante ao alinhamento de palavras entre as frases fonte e alvo na tradução automática, o alinhamento de palavras entre premissas e hipóteses pode ser perfeitamente realizado por mecanismos de atenção.

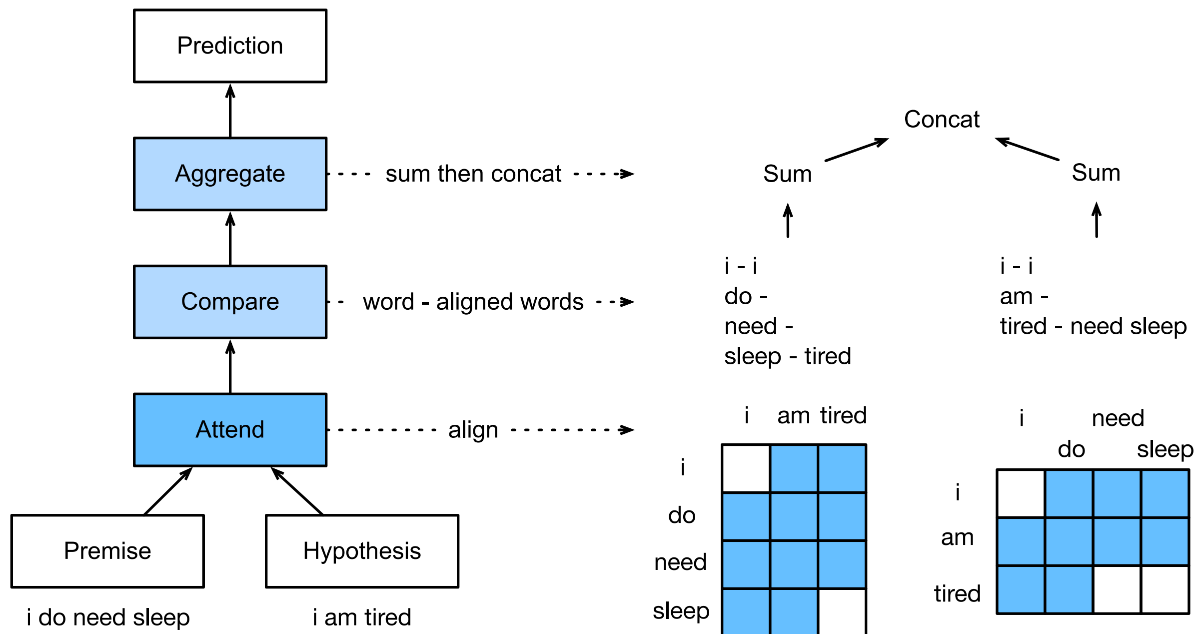


Fig. 15.5.2: Inferência de linguagem natural usando mecanismos de atenção.

Fig. 15.5.2 descreve o método de inferência de linguagem natural usando mecanismos de atenção. Em um nível superior, consiste em três etapas treinadas em conjunto: alinhar, comparar e agregar. Iremos ilustrá-los passo a passo a seguir.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

Alinhar

A primeira etapa é alinhar as palavras em uma sequência de texto a cada palavra na outra sequência. Suponha que a premissa seja "preciso dormir" e a hipótese "estou cansado". Devido à semelhança semântica, podemos desejar alinhar "i" na hipótese com "i" na premissa, e alinhe "cansado" na hipótese com "sono" na premissa. Da mesma forma, podemos desejar alinhar "i" na premissa com "i" na hipótese, e alinhar "necessidade" e "sono" na premissa com "cansado" na hipótese. Observe que esse alinhamento é *suave* usando a média ponderada, onde, idealmente, grandes pesos estão associados às palavras a serem alinhadas. Para facilitar a demonstração, Fig. 15.5.2 mostra tal alinhamento de uma maneira *dura*.

Agora descrevemos o alinhamento suave usando mecanismos de atenção com mais detalhes. Denotamos por $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_m)$ e $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ a premissa e hipótese, cujo número de palavras são m e n, respectivamente, onde $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ $(i = 1, \ldots, m, j = 1, \ldots, n)$ é um vetor de incorporação de palavras d-dimensional. Para o alinhamento suave, calculamos os pesos de atenção $e_{ij} \in \mathbb{R}$ como

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j), \tag{15.5.1}$$

onde a função f é um MLP definido na seguinte função `mlp`. A dimensão de saída de f é especificada pelo argumento `num_hiddens` de`mlp`.

```
def mlp(num_inputs, num_hiddens, flatten):
    net = []
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_inputs, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_hiddens, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    return nn.Sequential(*net)
```

Deve-se destacar que, em (15.5.1) f pega as entradas $\mathbf{a}_i$ and $\mathbf{b}_j$ separadamente em vez de pegar um par delas juntas como entrada. Este truque de *decomposição* leva a apenas aplicações $m + n$ (complexidade linear) de f em vez de mn aplicativos (complexidade quadrática).

Normalizando os pesos de atenção em (15.5.1), calculamos a média ponderada de todas as palavras incluídas na hipótese para obter a representação da hipótese que está suavemente alinhada com a palavra indexada por i na premissa:

$$\boldsymbol{\beta}_i = \sum_{j=1}^{n}\frac{\exp(e_{ij})}{ \sum_{k=1}^{n} \exp(e_{ik})} \mathbf{b}_j. \tag{15.5.2}$$

Da mesma forma, calculamos o alinhamento suave de palavras da premissa para cada palavra indexada por j na hipótese:

$$\boldsymbol{\alpha}_j = \sum_{i=1}^{m}\frac{\exp(e_{ij})}{ \sum_{k=1}^{m} \exp(e_{kj})} \mathbf{a}_i. \tag{15.5.3}$$

Abaixo, definimos a classe `Attend` para calcular o alinhamento suave das hipóteses (`beta`) com as premissas de entrada `A` e o alinhamento suave das premissas (`alfa`) com as hipóteses de entrada `B`.

```
class Attend(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B):
        # Shape of `A`/`B`: (`batch_size`, no. of words in sequence A/B,
```

(continues on next page)

(continued from previous page)

```
        # `embed_size`)
        # Shape of `f_A`/`f_B`: (`batch_size`, no. of words in sequence A/B,
        # `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # Shape of `e`: (`batch_size`, no. of words in sequence A,
        # no. of words in sequence B)
        e = torch.bmm(f_A, f_B.permute(0, 2, 1))
        # Shape of `beta`: (`batch_size`, no. of words in sequence A,
        # `embed_size`), where sequence B is softly aligned with each word
        # (axis 1 of `beta`) in sequence A
        beta = torch.bmm(F.softmax(e, dim=-1), B)
        # Shape of `alpha`: (`batch_size`, no. of words in sequence B,
        # `embed_size`), where sequence A is softly aligned with each word
        # (axis 1 of `alpha`) in sequence B
        alpha = torch.bmm(F.softmax(e.permute(0, 2, 1), dim=-1), A)
        return beta, alpha
```

Comparando

Na próxima etapa, comparamos uma palavra em uma sequência com a outra sequência que está suavemente alinhada com essa palavra. Observe que no alinhamento suave, todas as palavras de uma sequência, embora provavelmente com pesos de atenção diferentes, serão comparadas com uma palavra na outra sequência. Para facilitar a demonstração, Fig. 15.5.2 emparelha palavras com palavras alinhadas de uma forma *dura*. Por exemplo, suponha que a etapa de atendimento determina que "necessidade" e "sono" na premissa estão ambos alinhados com "cansado" na hipótese, o par "cansado - preciso dormir" será comparado.

Na etapa de comparação, alimentamos a concatenação (operador $[\cdot, \cdot]$) de palavras de uma sequência e palavras alinhadas de outra sequência em uma função g (um MLP):

$$\begin{split}\mathbf{v}_{A,i} = g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \ldots, m\\ \mathbf{v}_{B,j} = g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \ldots, n.\end{split} \tag{15.5.4}$$

Em:eqref:`eq_nli_v_ab`, $\mathbf{v}_{A,i}$ é a comparação entre a palavra i na premissa e todas as palavras da hipótese que estão suavemente alinhadas com a palavra i; enquanto $\mathbf{v}_{B,j}$ é a comparação entre a palavra j na hipótese e todas as palavras da premissa que estão suavemente alinhadas com a palavra j. A seguinte classe `Compare` define como a etapa de comparação.

```
class Compare(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(torch.cat([A, beta], dim=2))
        V_B = self.g(torch.cat([B, alpha], dim=2))
        return V_A, V_B
```

Agregando

Com dois conjuntos de vetores de comparação $\mathbf{v}_{A,i}$ ($i = 1, \ldots, m$) e $\mathbf{v}_{B,j}$ ($j = 1, \ldots, n$) disponível, na última etapa, agregaremos essas informações para inferir a relação lógica. Começamos resumindo os dois conjuntos:

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^{n} \mathbf{v}_{B,j}. \tag{15.5.5}$$

Em seguida, alimentamos a concatenação de ambos os resultados do resumo na função h (um MLP) para obter o resultado da classificação do relacionamento lógico:

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]). \tag{15.5.6}$$

A etapa de agregação é definida na seguinte classe `Aggregate`.

```
class Aggregate(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_inputs, num_hiddens, flatten=True)
        self.linear = nn.Linear(num_hiddens, num_outputs)

    def forward(self, V_A, V_B):
        # Sum up both sets of comparison vectors
        V_A = V_A.sum(dim=1)
        V_B = V_B.sum(dim=1)
        # Feed the concatenation of both summarization results into an MLP
        Y_hat = self.linear(self.h(torch.cat([V_A, V_B], dim=1)))
        return Y_hat
```

Juntando Tudo

Ao reunir as etapas de atendimento, comparação e agregação, definimos o modelo de atenção decomposto para treinar conjuntamente essas três etapas.

```
class DecomposableAttention(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_inputs_attend=100,
                 num_inputs_compare=200, num_inputs_agg=400, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_inputs_attend, num_hiddens)
        self.compare = Compare(num_inputs_compare, num_hiddens)
        # There are 3 possible outputs: entailment, contradiction, and neutral
        self.aggregate = Aggregate(num_inputs_agg, num_hiddens, num_outputs=3)

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

15.5.2 Treinamento e Avaliação do Modelo

Agora vamos treinar e avaliar o modelo de atenção decomposto definido no conjunto de dados SNLI. Começamos lendo o *dataset*.

Lendo o *Dataset*

Baixamos e lemos o conjunto de dados SNLI usando a função definida em Section 15.4. O tamanho do lote e o comprimento da sequência são definidos em 256 e 50, respectivamente.

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```

```
read 549367 examples
read 9824 examples
```

Criando o Modelo

Usamos a incorporação GloVe pré-treinada 100-dimensional para representar os *tokens* de entrada. Assim, predefinimos a dimensão dos vetores $\mathbf{a}_i$ e $\mathbf{b}_j$ em (15.5.1) como 100. A dimensão de saída das funções f in (15.5.1) e g em (15.5.4) é definida como 200. Em seguida, criamos uma instância de modelo, inicializamos seus parâmetros, e carregamos o *GloVe* embarcado para inicializar vetores de *tokens* de entrada.

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds);
```

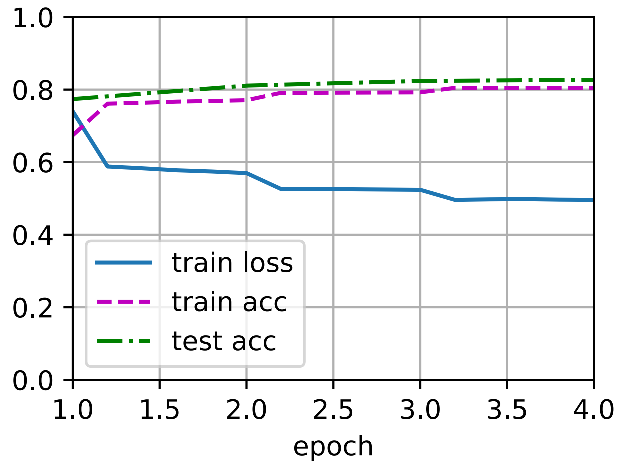
Treinamento e Avaliação do Modelo

Em contraste com a função `split_batch` em Section 12.5 que recebe entradas únicas, como sequências de texto (ou imagens), definimos uma função `split_batch_multi_inputs` para obter várias entradas, como premissas e hipóteses em minibatches.

Agora podemos treinar e avaliar o modelo no *dataset* SNLI.

```
lr, num_epochs = 0.001, 4
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.496, train acc 0.804, test acc 0.827
18580.5 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Usando o Modelo

Finalmente, defina a função de previsão para produzir a relação lógica entre um par de premissas e hipóteses.

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    net.eval()
    premise = torch.tensor(vocab[premise], device=d2l.try_gpu())
    hypothesis = torch.tensor(vocab[hypothesis], device=d2l.try_gpu())
    label = torch.argmax(net([premise.reshape((1, -1)),
                              hypothesis.reshape((1, -1))]), dim=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
            else 'neutral'
```

Podemos usar o modelo treinado para obter o resultado da inferência em linguagem natural para um par de frases de amostra.

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```

```
'contradiction'
```

15.5.3 Resumo

- O modelo de atenção decomposto consiste em três etapas para prever as relações lógicas entre premissas e hipóteses: atendimento, comparação e agregação.
- Com mecanismos de atenção, podemos alinhar palavras em uma sequência de texto com todas as palavras na outra e vice-versa. Esse alinhamento é suave usando a média ponderada, em que, idealmente, grandes pesos são associados às palavras a serem alinhadas.
- O truque da decomposição leva a uma complexidade linear mais desejável do que a complexidade quadrática ao calcular os pesos de atenção.
- Podemos usar a incorporação de palavras pré-treinadas como a representação de entrada

para tarefas de processamento de linguagem natural *downstream*, como inferência de linguagem natural.

15.5.4 Exercícios

1. Treine o modelo com outras combinações de hiperparâmetros. Você pode obter melhor precisão no conjunto de teste?
2. Quais são as principais desvantagens do modelo de atenção decomponível para inferência de linguagem natural?
3. Suponha que desejamos obter o nível de similaridade semântica (por exemplo, um valor contínuo entre 0 e 1) para qualquer par de sentenças. Como devemos coletar e rotular o conjunto de dados? Você pode projetar um modelo com mecanismos de atenção?

Discussõess[191]

15.6 Ajuste Fino de BERT para Aplicações de Nível de Sequência e de Token

Nas seções anteriores deste capítulo, projetamos diferentes modelos para aplicações de processamento de linguagem natural, como os baseados em RNNs, CNNs, atenção e MLPs. Esses modelos são úteis quando há restrição de espaço ou tempo, no entanto, elaborar um modelo específico para cada tarefa de processamento de linguagem natural é praticamente inviável. Em Section 14.7, introduzimos um modelo de pré-treinamento, BERT, que requer mudanças mínimas de arquitetura para uma ampla gama de tarefas de processamento de linguagem natural. Por um lado, na altura da sua proposta, o BERT melhorou o estado da arte em várias tarefas de processamento de linguagem natural. Por outro lado, conforme observado em Section 14.9, as duas versões do modelo BERT original vêm com 110 milhões e 340 milhões de parâmetros. Assim, quando há recursos computacionais suficientes, podemos considerar o ajuste fino do BERT para aplicativos de processamento de linguagem natural *downstream*.

A seguir, generalizamos um subconjunto de aplicações de processamento de linguagem natural como nível de sequência e nível de *token*. No nível da sequência, apresentamos como transformar a representação BERT da entrada de texto no rótulo de saída em classificação de texto único e classificação ou regressão de par de texto. No nível do *token*, apresentaremos brevemente novos aplicativos, como marcação de texto e resposta a perguntas, e esclareceremos como o BERT pode representar suas entradas e ser transformado em rótulos de saída. Durante o ajuste fino, as "mudanças mínimas de arquitetura" exigidas pelo BERT em diferentes aplicativos são as camadas extras totalmente conectadas. Durante o aprendizado supervisionado de uma aplicação *downstream*, os parâmetros das camadas extras são aprendidos do zero, enquanto todos os parâmetros no modelo BERT pré-treinado são ajustados.

[191] https://discuss.d2l.ai/t/1530

15.6.1 Classificação de Texto Único

- Classificação de texto único * pega uma única sequência de texto como entrada e produz seu resultado de classificação. Além da análise de sentimento que estudamos neste capítulo, o Corpus de Aceitabilidade Linguística (CoLA) também é um conjunto de dados para classificação de texto único, julgando se uma determinada frase é gramaticalmente aceitável ou não (Warstadt et al., 2019). Por exemplo, "Eu deveria estudar." é aceitável, mas "Eu deveria estudando." não é.

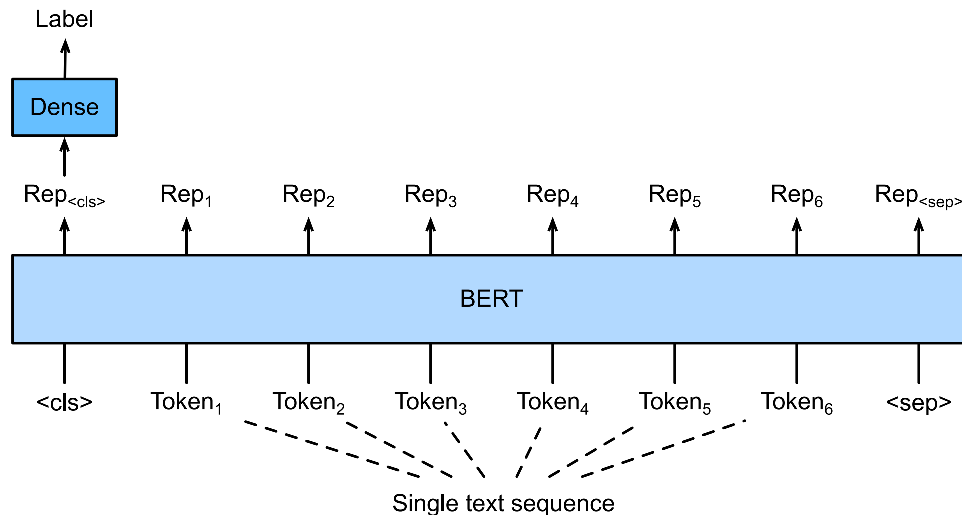


Fig. 15.6.1: Ajuste fino do BERT para aplicações de classificação de texto único, como análise de sentimento e teste de aceitabilidade linguística. Suponha que o texto único de entrada tenha seis *tokens*.

Section 14.7 descreve a representação de entrada de BERT. A sequência de entrada de BERT representa inequivocamente texto único e pares de texto, onde o *token* de classificação especial "<cls>" é usado para classificação de sequência e o *token* de classificação especial "<sep>" marca o fim de um único texto ou separa um par de texto. Conforme mostrado em Fig. 15.6.1, em aplicativos de classificação de texto único, a representação BERT do *token* de classificação especial "<cls>" codifica as informações de toda a sequência de texto de entrada. Como a representação do texto único de entrada, ele será alimentado em um pequeno MLP que consiste em camadas totalmente conectadas (densas) para gerar a distribuição de todos os valores de rótulo discretos.

15.6.2 Classificação ou Regressão de Pares de Texto

Também examinamos a inferência da linguagem natural neste capítulo. Pertence à *classificação de pares de texto*, um tipo de aplicativo que classifica um par de texto.

Tomando um par de texto como entrada, mas gerando um valor contínuo, *similaridade textual semântica* é uma tarefa popular de *regressão de par de texto*. Esta tarefa mede a similaridade semântica das sentenças. Por exemplo, no conjunto de dados Semantic Textual Similarity Benchmark, a pontuação de similaridade de um par de sentenças é uma escala ordinal que varia de 0 (sem sobreposição de significado) a 5 (significando equivalência) (Cer et al., 2017). O objetivo é prever essas pontuações. Exemplos do conjunto de dados de referência de similaridade textual semântica incluem (sentença 1, sentença 2, pontuação de similaridade):

- "Um avião está decolando.", "Um avião está decolando.", 5.000;

- "Mulher está comendo alguma coisa.", "Mulher está comendo carne.", 3.000;
- "Uma mulher está dançando.", "Um homem está falando.", 0,000.

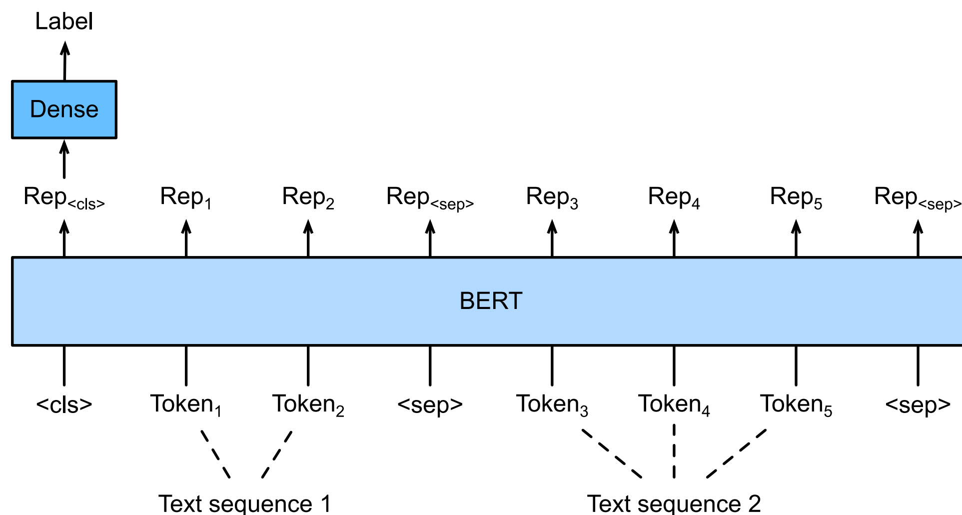


Fig. 15.6.2: Ajuste fino do BERT para aplicações de classificação ou regressão de pares de texto, como inferência de linguagem natural e similaridade textual semântica. Suponha que o par de texto de entrada tenha dois e três *tokens*.

Comparando com a classificação de texto único em Fig. 15.6.1, ajuste fino de BERT para classificação de par de texto em Fig. 15.6.2 é diferente na representação de entrada. Para tarefas de regressão de pares de texto, como semelhança textual semântica, mudanças triviais podem ser aplicadas, como a saída de um valor de rótulo contínuo e usando a perda quadrática média: eles são comuns para regressão.

15.6.3 Marcação de Texto

Agora, vamos considerar as tarefas de nível de *token*, como *marcação de texto*, onde cada *token* é atribuído a um rótulo. Entre as tarefas de marcação de texto, *marcação de classe gramatical* atribui a cada palavra uma marcação de classe gramatical (por exemplo, adjetivo e determinante) de acordo com a função da palavra na frase. Por exemplo, de acordo com o conjunto de *tags* Penn Treebank II, a frase "*John Smith 's car is new*" ("O carro de John Smith é novo") deve ser marcada como "NNP (substantivo, singular próprio) NNP POS (desinência possessiva) NN (substantivo, singular ou massa) VB (verbo, forma básica) JJ (adjetivo)".

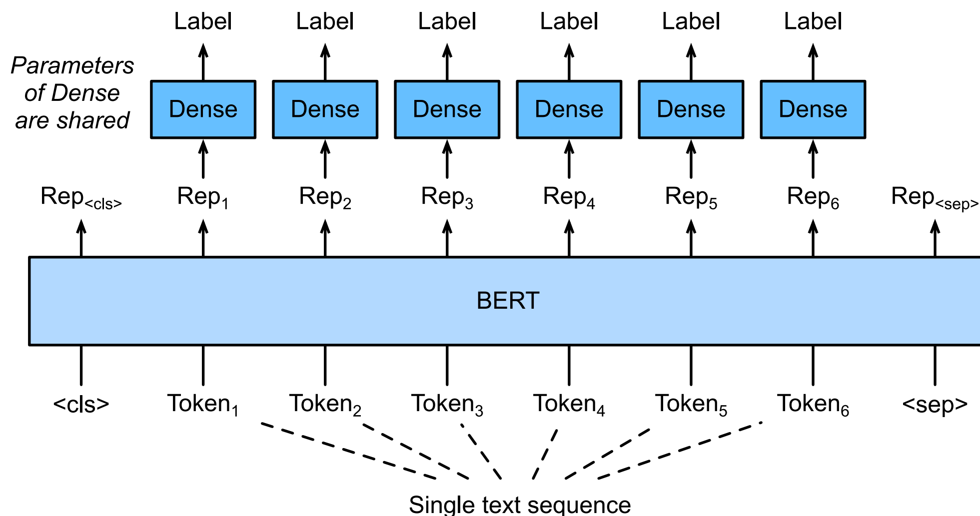


Fig. 15.6.3: Ajuste fino do BERT para aplicativos de marcação de texto, como marcação de classes gramaticais. Suponha que o texto único de entrada tenha seis *tokens*.

O ajuste fino do BERT para aplicações de marcação de texto é ilustrado em Fig. 15.6.3. Comparando com Fig. 15.6.1, a única distinção reside na marcação de texto, a representação BERT de *cada token* do texto de entrada é alimentado nas mesmas camadas extras totalmente conectadas para dar saída ao rótulo de o *token*, como uma *tag* de classe gramatical.

15.6.4 Resposta a Perguntas

Como outro aplicativo de nível de *token*, *responder a perguntas* reflete as capacidades de compreensão de leitura. Por exemplo, o conjunto de dados de resposta a perguntas de Stanford (SQuAD v1.1) consiste na leitura de passagens e perguntas, em que a resposta a cada pergunta é apenas um segmento de texto (extensão de texto) da passagem sobre a qual a pergunta se refere cite:`Rajpurkar.Zhang.Lopyrev.ea.2016`. Para explicar, considere uma passagem "Alguns especialistas relatam que a eficácia de uma máscara é inconclusiva. No entanto, os fabricantes de máscaras insistem que seus produtos, como as máscaras respiratórias N95, podem proteger contra o vírus." e uma pergunta "Quem disse que as máscaras respiratórias N95 podem proteger contra o vírus?". A resposta deve ser o intervalo de texto "fabricantes de máscara" na passagem. Assim, o objetivo no SQuAD v1.1 é prever o início e o fim da extensão do texto na passagem dada um par de pergunta e passagem.

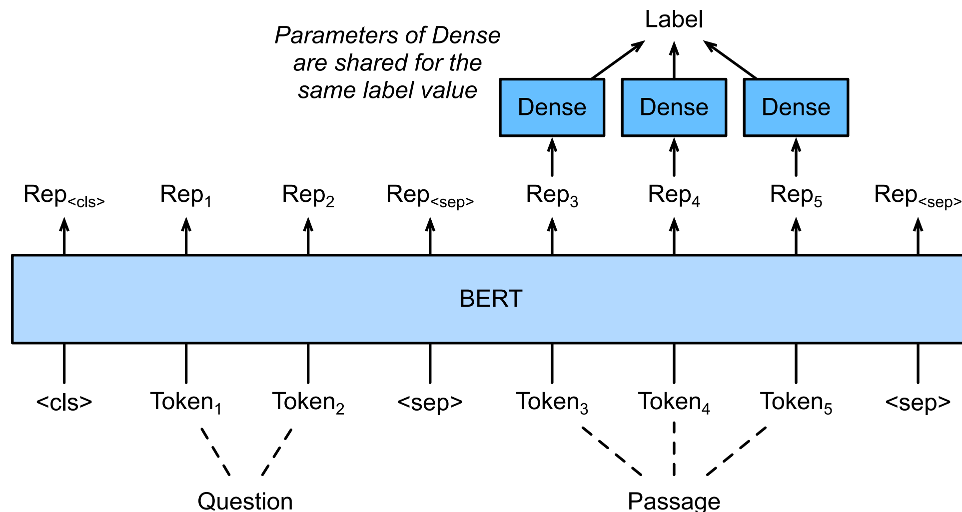


Fig. 15.6.4: Ajuste fino do BERT para resposta a perguntas. Suponha que o par de texto de entrada tenha dois e três *tokens*.

Para ajustar o BERT para responder às perguntas, a pergunta e a passagem são compactadas como a primeira e a segunda sequência de texto, respectivamente, na entrada do BERT. Para prever a posição do início do intervalo de texto, a mesma camada adicional totalmente conectada transformará a representação BERT de qualquer *token* da passagem da posição i em uma pontuação escalar s_i. Essas pontuações de todas as fichas de passagem são posteriormente transformadas pela operação *softmax* em uma distribuição de probabilidade, de modo que cada posição de ficha i na passagem recebe uma probabilidade p_i de ser o início do período de texto. A previsão do final da extensão do texto é igual à anterior, exceto que os parâmetros em sua camada adicional totalmente conectada são independentes daqueles para a previsão do início. Ao prever o final, qualquer *token* de passagem da posição i é transformado pela mesma camada totalmente conectada em uma pontuação escalar e_i. Fig. 15.6.4 descreve o ajuste fino do BERT para responder a perguntas.

Para responder a perguntas, o objetivo de treinamento da aprendizagem supervisionada é tão simples quanto maximizar as probabilidades-log das posições inicial e final de verdade. Ao prever a amplitude, podemos calcular a pontuação $s_i + e_j$ para um intervalo válido da posição i para a posição j ($i \leq j$), e produzir o intervalo com a pontuação mais alta.

15.6.5 Resumo

- BERT requer mudanças mínimas de arquitetura (camadas extras totalmente conectadas) para aplicações de processamento de linguagem natural em nível de sequência e *token*, como classificação de texto único (por exemplo, análise de sentimento e teste de aceitabilidade linguística), classificação de par de texto ou regressão (por exemplo, inferência de linguagem natural e semelhança textual semântica), marcação de texto (por exemplo, marcação de classe gramatical) e resposta a perguntas.
- Durante o aprendizado supervisionado de uma aplicação *downstream*, os parâmetros das camadas extras são aprendidos do zero, enquanto todos os parâmetros no modelo BERT pré-treinado são ajustados.

15.6.6 Exercícios

1. Vamos projetar um algoritmo de mecanismo de pesquisa para artigos de notícias. Quando o sistema recebe uma consulta (por exemplo, "indústria de petróleo durante o surto de coronavírus"), ele deve retornar uma lista classificada de artigos de notícias que são mais relevantes para a consulta. Suponha que temos um grande conjunto de artigos de notícias e um grande número de consultas. Para simplificar o problema, suponha que o artigo mais relevante tenha sido rotulado para cada consulta. Como podemos aplicar a amostragem negativa (ver Section 14.2.1) e BERT no projeto do algoritmo?
2. Como podemos alavancar o BERT nos modelos de treinamento de idiomas?
3. Podemos alavancar o BERT na tradução automática?

Discussões[192]

15.7 Inferência de Linguagem Natural: Ajuste Fino do BERT

Nas seções anteriores deste capítulo, nós projetamos uma arquitetura baseada na atenção (em Section 15.5) para a tarefa de inferência de linguagem natural no conjunto de dados SNLI (conforme descrito em Section 15.4). Agora, revisitamos essa tarefa fazendo o ajuste fino do BERT. Conforme discutido em Section 15.6, a inferência de linguagem natural é um problema de classificação de pares de texto em nível de sequência, e o ajuste fino de BERT requer apenas uma arquitetura adicional baseada em MLP, conforme ilustrado em Fig. 15.7.1.

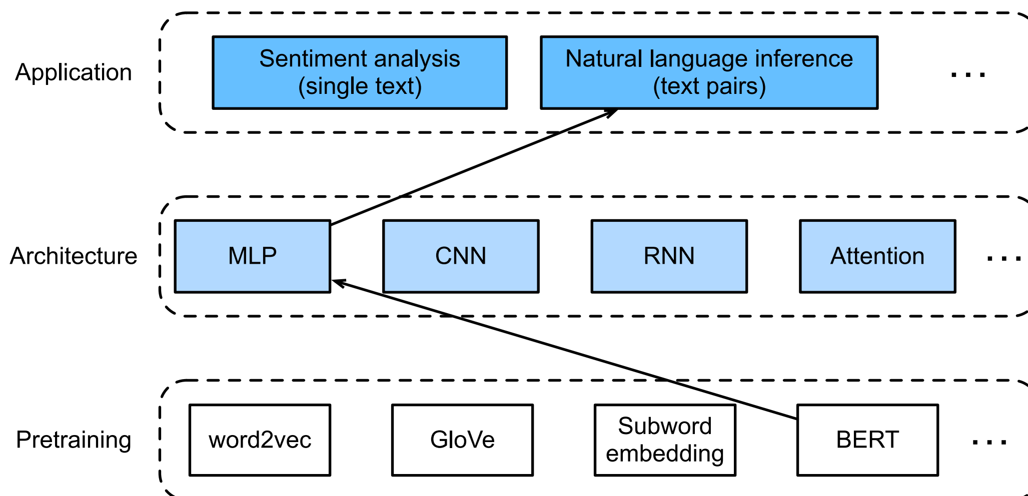


Fig. 15.7.1: Esta seção alimenta BERT pré-treinado para uma arquitetura baseada em MLP para inferência de linguagem natural.

Nesta secção, vamos baixar uma versão pequena pré-treinada de BERT, então ajuste-o para inferência de linguagem natural no conjunto de dados SNLI.

```
import json
import multiprocessing
import os
```

(continues on next page)

[192] https://discuss.d2l.ai/t/396

(continued from previous page)

```
import torch
from torch import nn
from d2l import torch as d2l
```

15.7.1 Carregando o BERT Pré-treinado

Explicamos como pré-treinar BERT no conjunto de dados WikiText-2 em Section 14.8 e Section 14.9 (observe que o modelo BERT original é pré-treinado em corpora muito maiores). Conforme discutido em Section 14.9, o modelo BERT original tem centenas de milhões de parâmetros. Na sequência, nós fornecemos duas versões de BERT pré-treinados: “bert.base” é quase tão grande quanto o modelo de base BERT original, que requer muitos recursos computacionais para o ajuste fino, enquanto “bert.small” é uma versão pequena para facilitar a demonstração.

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.torch.zip',
                             '225d66f04cae318b841a13d32af3acc165f253ac')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.torch.zip',
                              'c72329e68a732bef0452e4b96a1c341c8910f81f')
```

Qualquer um dos modelos BERT pré-treinados contém um arquivo “vocab.json” que define o conjunto de vocabulário e um arquivo “pretrained.params” dos parâmetros pré-treinados. Implementamos a seguinte função `load_pretrained_model` para carregar os parâmetros BERT pré-treinados.

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_layers, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # Define an empty vocabulary to load the predefined vocabulary
    vocab = d2l.Vocab()
    vocab.idx_to_token = json.load(open(os.path.join(data_dir, 'vocab.json')))
    vocab.token_to_idx = {token: idx for idx, token in enumerate(
        vocab.idx_to_token)}
    bert = d2l.BERTModel(len(vocab), num_hiddens, norm_shape=[256],
                         ffn_num_input=256, ffn_num_hiddens=ffn_num_hiddens,
                         num_heads=4, num_layers=2, dropout=0.2,
                         max_len=max_len, key_size=256, query_size=256,
                         value_size=256, hid_in_features=256,
                         mlm_in_features=256, nsp_in_features=256)
    # Load pretrained BERT parameters
    bert.load_state_dict(torch.load(os.path.join(data_dir,
                                                 'pretrained.params')))
    return bert, vocab
```

Para facilitar a demonstração na maioria das máquinas, vamos carregar e ajustar a versão pequena (“bert.small”) do BERT pré-treinado nesta seção. No exercício, mostraremos como ajustar o “bert.base” muito maior para melhorar significativamente a precisão do teste.

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_layers=2, dropout=0.1, max_len=512, devices=devices)
```

```
Downloading ../data/bert.small.torch.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪bert.small.torch.zip...
```

15.7.2 O Conjunto de Dados para Ajuste Fino do BERT

Para a inferência de linguagem natural da tarefa *downstream* no conjunto de dados SNLI, definimos uma classe de conjunto de dados customizada `SNLIBERTDataset`. Em cada exemplo, a premissa e a hipótese formam um par de sequência de texto e são compactados em uma sequência de entrada de BERT conforme descrito em Fig. 15.6.2. Lembre-se Section 14.7.4 que IDs de segmento são usados para distinguir a premissa e a hipótese em uma sequência de entrada do BERT. Com o comprimento máximo predefinido de uma sequência de entrada de BERT (`max_len`), o último *token* do mais longo do par de texto de entrada continua sendo removido até `max_len` é atendido. Para acelerar a geração do conjunto de dados SNLI para o ajuste fino de BERT, usamos 4 processos de trabalho para gerar exemplos de treinamento ou teste em paralelo.

```python
class SNLIBERTDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [[
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
            *[d2l.tokenize([s.lower() for s in sentences])
              for sentences in dataset[:2]])]

        self.labels = torch.tensor(dataset[2])
        self.vocab = vocab
        self.max_len = max_len
        (self.all_token_ids, self.all_segments,
         self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
        print('read ' + str(len(self.all_token_ids)) + ' examples')

    def _preprocess(self, all_premise_hypothesis_tokens):
        pool = multiprocessing.Pool(4)  # Use 4 worker processes
        out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
        all_token_ids = [
            token_ids for token_ids, segments, valid_len in out]
        all_segments = [segments for token_ids, segments, valid_len in out]
        valid_lens = [valid_len for token_ids, segments, valid_len in out]
        return (torch.tensor(all_token_ids, dtype=torch.long),
                torch.tensor(all_segments, dtype=torch.long),
                torch.tensor(valid_lens))

    def _mp_worker(self, premise_hypothesis_tokens):
        p_tokens, h_tokens = premise_hypothesis_tokens
        self._truncate_pair_of_tokens(p_tokens, h_tokens)
        tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
        token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
                             * (self.max_len - len(tokens))
        segments = segments + [0] * (self.max_len - len(segments))
        valid_len = len(tokens)
        return token_ids, segments, valid_len

    def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
        # Reserve slots for '<CLS>', '<SEP>', and '<SEP>' tokens for the BERT
        # input
```

(continues on next page)

(continued from previous page)

```
        while len(p_tokens) + len(h_tokens) > self.max_len - 3:
            if len(p_tokens) > len(h_tokens):
                p_tokens.pop()
            else:
                h_tokens.pop()

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx]), self.labels[idx]

    def __len__(self):
        return len(self.all_token_ids)
```

Depois de baixar o conjunto de dados SNLI, geramos exemplos de treinamento e teste instanciando a classe `SNLIBERTDataset`. Esses exemplos serão lidos em minibatches durante o treinamento e teste de inferência de linguagem natural.

```
# Reduce `batch_size` if there is an out of memory error. In the original BERT
# model, `max_len` = 512
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True,
                                    num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                   num_workers=num_workers)
```

```
read 549367 examples
read 9824 examples
```

15.7.3 Ajuste Fino do BERT

Como Fig. 15.6.2 indica, ajuste fino do BERT para inferência de linguagem natural requer apenas um MLP extra que consiste em duas camadas totalmente conectadas (veja `self.hidden` e `self.output` na seguinte classe `BERTClassifier`). Este MLP transforma o Representação de BERT do *token* especial "<cls>", que codifica as informações tanto da premissa quanto da hipótese, em três resultados de inferência de linguagem natural: implicação, contradição e neutro.

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Linear(256, 3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

Na sequência, o modelo BERT pré-treinado `bert` é alimentado na instância`BERTClassifier` `net` para a aplicação *downstream*. Em implementações comuns de ajuste fino de BERT, apenas os parâmetros da camada de saída do MLP adicional (`net.output`) serão aprendidos do zero. Todos os parâmetros do codificador BERT pré-treinado (`net.encoder`) e a camada oculta do MLP adicional (`net.hidden`) serão ajustados.

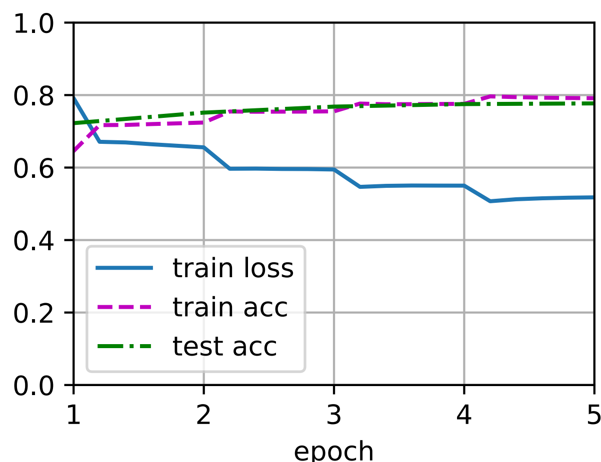
```
net = BERTClassifier(bert)
```

Lembre-se disso in Section 14.7 ambas as classes `MaskLM` e`NextSentencePred` têm parâmetros em suas MLPs empregadas. Esses parâmetros são parte daqueles no modelo BERT pré-treinado `bert`, e, portanto, parte dos parâmetros em `net`. No entanto, esses parâmetros são apenas para computação a perda de modelagem de linguagem mascarada e a perda de previsão da próxima frase durante o pré-treinamento. Essas duas funções de perda são irrelevantes para o ajuste fino de aplicativos *downstream*, assim, os parâmetros das MLPs empregadas em `MaskLM` e `NextSentencePred` não são atualizados (obsoletos) quando o BERT é ajustado.

Para permitir parâmetros com gradientes obsoletos, o sinalizador `ignore_stale_grad = True` é definido na função `step` de `d2l.train_batch_ch13`. Usamos esta função para treinar e avaliar o modelo `net` usando o conjunto de treinamento (`train_iter`) e o conjunto de teste (`test_iter`) de SNLI. Devido aos recursos computacionais limitados, a precisão de treinamento e teste pode ser melhorado ainda mais: deixamos suas discussões nos exercícios.

```
lr, num_epochs = 1e-4, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction='none')
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.518, train acc 0.791, test acc 0.777
8488.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



15.7.4 Resumo

- Podemos ajustar o modelo BERT pré-treinado para aplicativos *downstream*, como inferência de linguagem natural no conjunto de dados SNLI.
- Durante o ajuste fino, o modelo BERT torna-se parte do modelo para a aplicação *downstream*. Os parâmetros relacionados apenas à perda de pré-treinamento não serão atualizados durante o ajuste fino.

15.7.5 Exercícios

1. Faça o ajuste fino de um modelo de BERT pré-treinado muito maior que é quase tão grande quanto o modelo de base de BERT original se seu recurso computacional permitir. Defina os argumentos na função `load_pretrained_model` como: substituindo 'bert.small' por 'bert.base', aumentando os valores de `num_hiddens = 256`, `ffn_num_hiddens = 512`, `num_heads = 4`, `num_layers = 2` para `768`, `3072`, `12`, `12`, respectivamente. Aumentando os períodos de ajuste fino (e possivelmente ajustando outros hiperparâmetros), você pode obter uma precisão de teste superior a 0,86?
2. Como truncar um par de sequências de acordo com sua proporção de comprimento? Compare este método de truncamento de par e aquele usado na classe `SNLIBERTDataset`. Quais são seus prós e contras?

Discussões[193]

[193] https://discuss.d2l.ai/t/1526

16 Sistemas de Recomendação

Shuai Zhang (*Amazon*), **Aston Zhang** (*Amazon*), ande **Yi Tay** (*Google*)

Os sistemas de recomendação são amplamente empregados na indústria e onipresentes em nossa vida diária. Esses sistemas são utilizados em várias áreas, como sites de compras online (por exemplo, amazon.com), site de serviços de música / filmes (por exemplo, Netflix e Spotify), lojas de aplicativos móveis (por exemplo, loja de aplicativos IOS e google play), online publicidade, apenas para citar alguns.

O principal objetivo dos sistemas de recomendação é ajudar os usuários a descobrir itens relevantes, como filmes para assistir, textos para ler ou produtos para comprar, de modo a criar uma experiência agradável para o usuário. Além disso, os sistemas de recomendação estão entre os sistemas de aprendizado de máquina mais poderosos que os varejistas online implementam para gerar receita incremental. Os sistemas de recomendação são substitutos dos motores de busca, reduzindo os esforços em buscas proativas e surpreendendo os usuários com ofertas que eles nunca procuraram. Muitas empresas conseguiram se posicionar à frente de seus concorrentes com a ajuda de sistemas de recomendação mais eficazes. Como tal, os sistemas de recomendação são centrais não apenas para nossas vidas diárias, mas também altamente indispensáveis em alguns setores.

Neste capítulo, cobriremos os fundamentos e avanços dos sistemas de recomendação, junto com a exploração de algumas técnicas fundamentais comuns para a construção de sistemas de recomendação com diferentes fontes de dados disponíveis e suas implementações. Especificamente, você aprenderá como prever a classificação que um usuário pode dar a um item em potencial, como gerar uma lista de recomendação de itens e como prever a taxa de cliques de recursos abundantes. Essas tarefas são comuns em aplicativos do mundo real. Ao estudar este capítulo, você obterá experiência prática relacionada à solução de problemas de recomendação do mundo real não apenas com métodos clássicos, mas também com modelos baseados em aprendizado profundo mais avançados.

16.1 Visão geral dos sistemas de recomendação

Na última década, a Internet evoluiu para uma plataforma para serviços online de grande escala, o que mudou profundamente a maneira como nos comunicamos, lemos notícias, compramos produtos e assistimos filmes. Nesse ínterim, o número sem precedentes de itens (usamos o termo *item* para nos referir a filmes, notícias, livros e produtos) oferecidos online requer um sistema que pode nos ajudar a descobrir os itens de nossa preferência. Os sistemas de recomendação são, portanto, ferramentas poderosas de filtragem de informações que podem facilitar serviços personalizados e fornecer experiência sob medida para usuários individuais. Em suma, os sistemas de recomendação desempenham um papel fundamental na utilização da riqueza de dados disponíveis para fazer escolhas gerenciáveis. Hoje em dia, os sistemas de recomendação estão

no centro de vários provedores de serviços online, como Amazon, Netflix e YouTube. Lembre-se do exemplo de livros de aprendizagem profunda recomendados pela Amazon em Fig. 1.3.3. Os benefícios de empregar sistemas de recomendação são duplos: por um lado, pode reduzir muito o esforço dos usuários em encontrar itens e aliviar a questão da sobrecarga de informações. Por outro lado, pode agregar valor ao negócio online prestadores de serviços e é uma importante fonte de receita. Este capítulo irá apresentar os conceitos fundamentais, modelos clássicos e avanços recentes com aprendizado profundo no campo de sistemas de recomendação, juntamente com exemplos implementados.

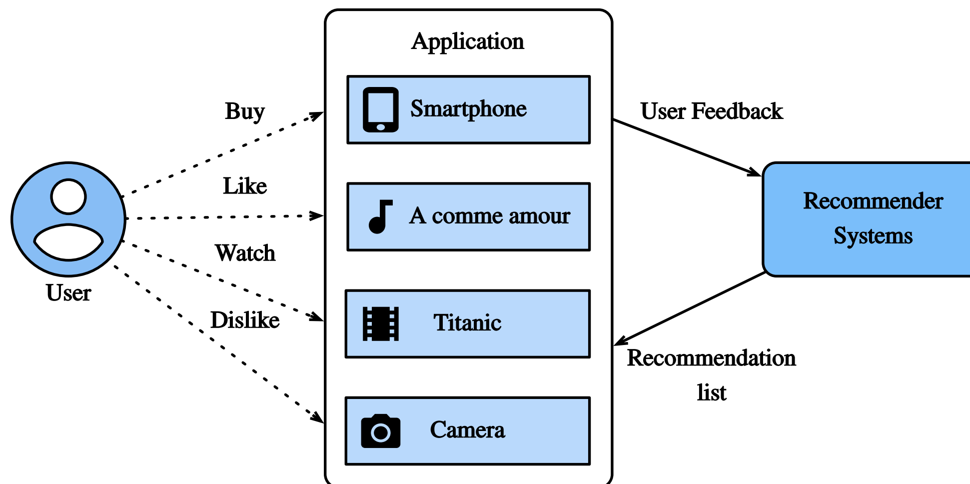


Fig. 16.1.1: Illustration of the Recommendation Process

16.1.1 Filtragem colaborativa

Começamos a jornada com o conceito importante em sistemas de recomendação —filtragem colaborativa (CF), que foi cunhado pela primeira vez pelo sistema Tapestry (Goldberg et al., 1992), referindo-se a "as pessoas colaboram para se ajudarem a realizar o processo de filtragem a fim de lidar com as grandes quantidades de e-mail e mensagens postadas em grupos de notícias". Este termo foi enriquecido com mais sentidos. Em um sentido amplo, é o processo de filtragem de informações ou padrões usando técnicas que envolvem colaboração entre vários usuários, agentes e fontes de dados. FC tem muitas formas e vários métodos de FC propostos desde seu advento.

No geral, as técnicas de CF podem ser categorizadas em: CF baseado em memória, CF baseado em modelo e seus híbridos (Su & Khoshgoftaar, 2009). As técnicas representativas de CF baseado em memória são o CF baseado no vizinho mais próximo, como CF baseado em usuário e CF baseado em item (Sarwar et al., 2001). Modelos de fator latente, como fatoração de matriz, são exemplos de CF baseado em modelo. O CF baseado em memória tem limitações para lidar com dados esparsos e em grande escala, uma vez que calcula os valores de similaridade com base em itens comuns. Métodos baseados em modelos tornam-se mais populares com seus melhor capacidade para lidar com esparsidade e escalabilidade. Muitas abordagens de CF baseadas em modelo podem ser estendidas com redes neurais, levando a modelos mais flexíveis e escaláveis com a aceleração de computação no aprendizado profundo (Zhang et al., 2019). Em geral, o CF usa apenas os dados de interação do item do usuário para fazer previsões e recomendações. Além do CF, os sistemas de recomendação baseados em conteúdo e contexto também são úteis para incorporar as descrições de conteúdo de itens/usuários e sinais contextuais, como carimbos de data/hora e locais. Obviamente, podemos precisar ajustar os tipos/estruturas do modelo quando diferentes dados de entrada estiverem disponíveis.

16.1.2 Feedback explícito e feedback implícito

Para saber a preferência dos usuários, o sistema deve coletar feedback deles. O feedback pode ser explícito ou implícito (Hu et al., 2008). Por exemplo, IMDB[194] coleta classificações por estrelas que variam de uma a dez estrelas para filmes. O YouTube fornece os botões de polegar para cima e para baixo para que os usuários mostrem suas preferências. É evidente que a coleta de feedback explícito exige que os usuários indiquem seus interesses de forma proativa. No entanto, o feedback explícito nem sempre está disponível, pois muitos usuários podem relutar em avaliar os produtos. Relativamente falando, o feedback implícito está frequentemente disponível, uma vez que se preocupa principalmente com a modelagem do comportamento implícito, como cliques do usuário. Como tal, muitos sistemas de recomendação são centrados no feedback implícito que reflete indiretamente a opinião do usuário por meio da observação do comportamento do usuário. Existem diversas formas de feedback implícito, incluindo histórico de compras, histórico de navegação, relógios e até movimentos do mouse. Por exemplo, um usuário que comprou muitos livros do mesmo autor provavelmente gosta desse autor. Observe que o feedback implícito é inerentemente barulhento. Nós podemos apenas *adivinhar* suas preferências e verdadeiros motivos. Um usuário assistiu a um filme não indica necessariamente uma visão positiva desse filme.

16.1.3 Tarefas de recomendação

Várias tarefas de recomendação foram investigadas nas últimas décadas. Com base no domínio de aplicativos, há recomendação de filmes, recomendações de notícias, recomendação de ponto de interesse (Ye et al., 2011) e assim por diante. Também é possível diferenciar as tarefas com base nos tipos de feedback e dados de entrada, por exemplo, a tarefa de previsão de classificação visa prever as classificações explícitas. A recomendação Top-n (classificação de itens) classifica todos os itens de cada usuário pessoalmente com base no feedback implícito. Se as informações de carimbo de data / hora também forem incluídas, podemos construir uma recomendação ciente de sequência (Quadrana et al., 2018). Outra tarefa popular é chamada de previsão de taxa de cliques, que também é baseada em feedback implícito, mas vários recursos categóricos podem ser utilizados. Recomendar para novos usuários e recomendar novos itens para usuários existentes são chamados de recomendação de inicialização a frio (Schein et al., 2002).

[194] https://www.imdb.com/

16.1.4 Sumário

- Os sistemas de recomendação são importantes para usuários individuais e indústrias. A filtragem colaborativa é um conceito-chave na recomendação.
- Existem dois tipos de feedbacks: feedback implícito e feedback explícito. Várias tarefas de recomendação foram exploradas durante a última década.

16.1.5 Exercícios

1. Você pode explicar como os sistemas de recomendação influenciam sua vida diária?
2. Que tarefas de recomendação interessantes você acha que podem ser investigadas?

Discussão[195]

[195] https://discuss.d2l.ai/t/398

17 | Redes Adversariais Generativas

17.1 Redes Adversariais Generativas

Ao longo da maior parte deste livro, falamos sobre como fazer previsões. De uma forma ou de outra, usamos redes neurais profundas, mapeamentos aprendidos de exemplos de dados para rótulos. Esse tipo de aprendizado é chamado de aprendizado discriminativo, pois gostaríamos de ser capazes de discriminar entre fotos de gatos e fotos de cachorros. Classificadores e regressores são exemplos de aprendizagem discriminativa. E as redes neurais treinadas por retropropagação mudaram tudo que pensávamos saber sobre aprendizagem discriminativa em grandes *datasets* complicados. A precisão da classificação em imagens de alta resolução passou de inútil para o nível humano (com algumas ressalvas) em apenas 5-6 anos. Vamos poupar você de outro discurso sobre todas as outras tarefas discriminativas em que as redes neurais profundas se saem surpreendentemente bem.

Mas o aprendizado de máquina é mais do que apenas resolver tarefas discriminativas. Por exemplo, dado um grande conjunto de dados, sem rótulos, podemos querer aprender um modelo que capture de forma concisa as características desses dados. Dado esse modelo, poderíamos amostrar exemplos de dados sintéticos que se assemelham à distribuição dos dados de treinamento. Por exemplo, dado um grande corpus de fotografias de rostos, podemos querer ser capazes de gerar uma nova imagem fotorrealística que pareça plausivelmente proveniente do mesmo *dataset*. Esse tipo de aprendizado é chamado de modelagem generativa.

Até recentemente, não tínhamos nenhum método que pudesse sintetizar novas imagens fotorrealistas. Mas o sucesso das redes neurais profundas para aprendizagem discriminativa abriu novas possibilidades. Uma grande tendência nos últimos três anos tem sido a aplicação de redes profundas discriminativas para superar desafios em problemas que geralmente não consideramos problemas de aprendizagem supervisionada. Os modelos de linguagem de rede neural recorrente são um exemplo do uso de uma rede discriminativa (treinada para prever o próximo caractere) que, uma vez treinada, pode atuar como um modelo gerador.

Em 2014, um artigo inovador apresentou Redes adversariais gerativas (GANs) (Goodfellow et al., 2014), uma nova maneira inteligente de alavancar o poder dos modelos discriminativos para obter bons modelos generativos. Em sua essência, as GANs confiam na ideia de que um gerador de dados é bom se não podemos distinguir os dados falsos dos reais. Em estatística, isso é chamado de teste de duas amostras - um teste para responder à pergunta se os conjuntos de dados $X = \{x_1, \ldots, x_n\}$ and $X' = \{x'_1, \ldots, x'_n\}$ foram retirados da mesma distribuição. A principal diferença entre a maioria dos artigos de estatística e os GANs é que os últimos usam essa ideia de forma construtiva. Em outras palavras, em vez de apenas treinar um modelo para dizer "ei, esses dois conjuntos de dados não parecem vir da mesma distribuição", eles usam o teste de duas amostras[196] para fornecer sinais de treinamento para um modelo generativo. Isso nos permite melhorar o

[196] https://en.wikipedia.org/wiki/Two-sample_hypothesis_testing

gerador de dados até que ele gere algo que se pareça com os dados reais. No mínimo, ele precisa enganar o classificador. Mesmo que nosso classificador seja uma rede neural profunda de última geração.

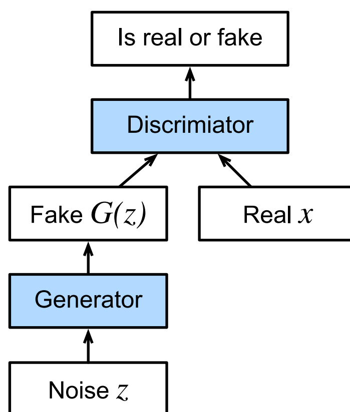


Fig. 17.1.1: Redes Adversariais Generativas

A arquitetura GAN é ilustrada em Fig. 17.1.1. Como você pode ver, há duas peças na arquitetura GAN - primeiro, precisamos de um dispositivo (digamos, uma rede profunda, mas realmente pode ser qualquer coisa, como um mecanismo de renderização de jogo) que pode potencialmente ser capaz de gerar dados que parecem assim como a coisa real. Se estamos lidando com imagens, isso precisa gerar imagens. Se estamos lidando com a fala, ela precisa gerar sequências de áudio e assim por diante. Chamamos isso de rede do gerador. O segundo componente é a rede discriminadora. Ele tenta distinguir dados falsos e reais uns dos outros. Ambas as redes competem entre si. A rede do gerador tenta enganar a rede do discriminador. Nesse ponto, a rede discriminadora se adapta aos novos dados falsos. Essas informações, por sua vez, são utilizadas para melhorar a rede do gerador, e assim por diante.

O discriminador é um classificador binário para distinguir se a entrada x é real (dos dados reais) ou falsa (do gerador). Normalmente, o discriminador gera uma previsão escalar $o \in \mathbb{R}$ para a entrada $\mathbf{x}$, como usar uma camada densa com tamanho oculto 1 e, em seguida, aplicar a função sigmóide para obter a probabilidade prevista $D(\mathbf{x}) = 1/(1+e^{-o})$. Suponha que o rótulo y para os dados verdadeiros seja 1 e 0 para os dados falsos. Treinamos o discriminador para minimizar a perda de entropia cruzada, *ou seja*,

$$\min_D \{ - y \log D(\mathbf{x}) - (1-y)\log(1-D(\mathbf{x})) \}, \tag{17.1.1}$$

Para o gerador, ele primeiro desenha algum parâmetro $\mathbf{z}\in\mathbb{R}^d$ de uma fonte de aleatoriedade, *por exemplo*, uma distribuição normal $\mathbf{z} \sim \mathcal{N}(0, 1)$. Frequentemente chamamos $\mathbf{z}$ como a variável latente. Em seguida, aplica uma função para gerar $\mathbf{x}'=G(\mathbf{z})$. O objetivo do gerador é enganar o discriminador para classificar $\mathbf{x}'=G(\mathbf{z})$ como dados verdadeiros, *ou seja*, queremos $D(G(\mathbf{z})) \approx 1$. Em outras palavras, para um determinado discriminador D, atualizamos os parâmetros do gerador G para maximizar a perda de entropia cruzada quando $y=0$, *ou seja*,

$$\max_G \{ - (1-y) \log(1-D(G(\mathbf{z}))) \} = \max_G \{ - \log(1-D(G(\mathbf{z}))) \}. \tag{17.1.2}$$

Se o gerador fizer um trabalho perfeito, então $D(\mathbf{x}')\approx 1$ então a perda acima próxima a 0, o que resulta em gradientes muito pequenos para fazer um bom progresso para o discriminador. Então, comumente, minimizamos a seguinte perda:

$$\min_G \{ - y \log(D(G(\mathbf{z}))) \} = \min_G \{ - \log(D(G(\mathbf{z}))) \}, \tag{17.1.3}$$

que é apenas alimentar $\mathbf{x}' = G(\mathbf{z})$ no discriminador, mas dando o rótulo $y = 1$.

Resumindo, D e G estão jogando um jogo "minimax" com a função objetivo abrangente:

$$min_D max_G \{-E_{x\sim\text{Data}} log D(\mathbf{x}) - E_{z\sim\text{Noise}} log(1 - D(G(\mathbf{z})))\}. \tag{17.1.4}$$

Muitos dos aplicativos GANs estão no contexto de imagens. Como demonstração, vamos nos contentar em ajustar primeiro uma distribuição muito mais simples. Ilustraremos o que acontece se usarmos GANs para construir o estimador de parâmetros mais ineficiente do mundo para um gaussiano. Vamos começar.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

17.1.1 Gerando Alguns Dados "Reais"

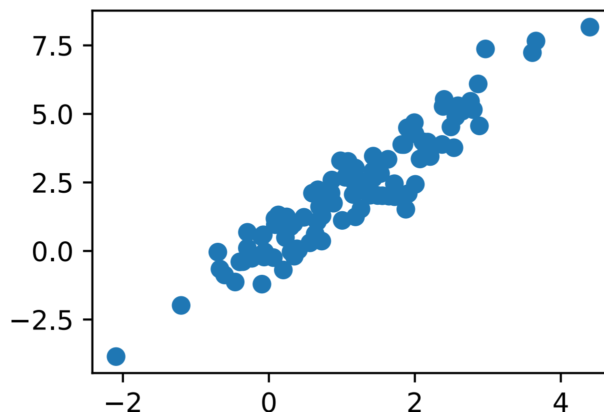
Como esse será o exemplo mais bobo do mundo, simplesmente geramos dados extraídos de uma gaussiana.

```
X = torch.normal(0.0, 1, (1000, 2))
A = torch.tensor([[1, 2], [-0.1, 0.5]])
b = torch.tensor([1, 2])
data = torch.matmul(X, A) + b
```

Vamos ver o que temos. Esta deve ser uma Gaussiana deslocada de alguma forma bastante arbitrária com média b e matriz de covariância $A^T A$.

```
d2l.set_figsize()
d2l.plt.scatter(data[:100, (0)].detach().numpy(), data[:100, (1)].detach().numpy());
print(f'The covariance matrix is\n{torch.matmul(A.T, A)}')
```

```
The covariance matrix is
tensor([[1.0100, 1.9500],
        [1.9500, 4.2500]])
```



```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

17.1.2 Gerador

Nossa rede de geradores será a rede mais simples possível - um modelo linear de camada única. Isso porque estaremos conduzindo essa rede linear com um gerador de dados Gaussiano. Portanto, ele literalmente só precisa aprender os parâmetros para falsificar as coisas perfeitamente.

```
net_G = nn.Sequential(nn.Linear(2, 2))
```

17.1.3 Discriminador

Para o discriminador, seremos um pouco mais discriminativos: usaremos um MLP com 3 camadas para tornar as coisas um pouco mais interessantes.

```
net_D = nn.Sequential(
    nn.Linear(2, 5), nn.Tanh(),
    nn.Linear(5, 3), nn.Tanh(),
    nn.Linear(3, 1))
```

17.1.4 Treinamento

Primeiro, definimos uma função para atualizar o discriminador.

```
#@save
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Update discriminator."""
    batch_size = X.shape[0]
    ones = torch.ones((batch_size,), device=X.device)
    zeros = torch.zeros((batch_size,), device=X.device)
    trainer_D.zero_grad()
    real_Y = net_D(X)
    fake_X = net_G(Z)
    # Do not need to compute gradient for `net_G`, detach it from
    # computing gradients.
    fake_Y = net_D(fake_X.detach())
    loss_D = (loss(real_Y, ones.reshape(real_Y.shape)) +
              loss(fake_Y, zeros.reshape(fake_Y.shape))) / 2
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

O gerador é atualizado de forma semelhante. Aqui, reutilizamos a perda de entropia cruzada, mas mudamos o rótulo dos dados falsos de 0 para 1.

```
#@save
def update_G(Z, net_D, net_G, loss, trainer_G):
```

(continues on next page)

(continued from previous page)

```
    """Update generator."""
    batch_size = Z.shape[0]
    ones = torch.ones((batch_size,), device=Z.device)
    trainer_G.zero_grad()
    # We could reuse `fake_X` from `update_D` to save computation
    fake_X = net_G(Z)
    # Recomputing `fake_Y` is needed since `net_D` is changed
    fake_Y = net_D(fake_X)
    loss_G = loss(fake_Y, ones.reshape(fake_Y.shape))
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

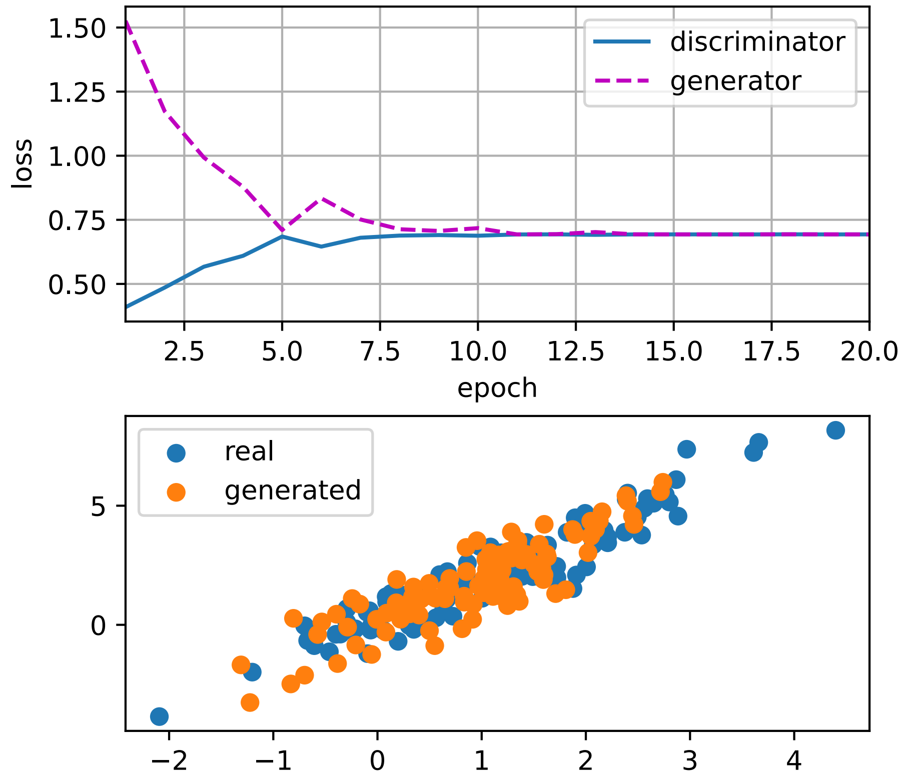
Tanto o discriminador quanto o gerador realizam uma regressão logística binária com a perda de entropia cruzada. Usamos Adam para facilitar o processo de treinamento. Em cada iteração, primeiro atualizamos o discriminador e depois o gerador. Visualizamos perdas e exemplos gerados.

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D)
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Visualize generated examples
        Z = torch.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).detach().numpy()
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(['real', 'generated'])
        # Show the losses
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch + 1, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec')
```

Agora especificamos os hiperparâmetros para se ajustar à distribuição gaussiana.

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, data[:100].detach().numpy())
```

```
loss_D 0.693, loss_G 0.693, 1539.5 examples/sec
```



17.1.5 Resumo

- Redes adversárias gerativas (GANs) são compostas por duas redes profundas, a geradora e a discriminadora.
- O gerador gera a imagem o mais próximo possível da imagem verdadeira para enganar o discriminador, maximizando a perda de entropia cruzada, *ou seja* $\max \log(D(\mathbf{x}'))$. O discriminador tenta distinguir as imagens geradas das imagens verdadeiras, minimizando a perda de entropia cruzada, *ou seja*, $\min -y \log D(\mathbf{x}) - (1-y)\log(1-D(\mathbf{x}))$.

17.1.6 Exercícios

- Existe um equilíbrio onde o gerador vence, *ou seja* o discriminador acaba incapaz de distinguir as duas distribuições em amostras finitas?

Discussões[197]

[197] https://discuss.d2l.ai/t/1082

17.2 Redes Adversariais Gerativas Convolucionais Profundas

Em Section 17.1, apresentamos as idéias básicas por trás de como funcionam os GANs. Mostramos que eles podem extrair amostras de alguma distribuição simples e fácil de amostrar, como uma distribuição uniforme ou normal, e transformá-los em amostras que parecem corresponder à distribuição de algum conjunto de dados. E embora nosso exemplo de correspondência de uma distribuição gaussiana 2D tenha chegado ao ponto, não é especialmente empolgante.

Nesta seção, demonstraremos como você pode usar GANs para gerar imagens fotorrealísticas. Estaremos baseando nossos modelos nos GANs convolucionais profundos (DCGAN) introduzidos em (Radford et al., 2015). Vamos tomar emprestada a arquitetura convolucional que se mostrou tão bem-sucedida para problemas discriminativos de visão de computador e mostrar como, por meio de GANs, eles podem ser aproveitados para gerar imagens fotorrealistas.

```
import warnings
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

17.2.1 O *Dataset* de Pokémon

O *dataset* que usaremos é uma coleção de sprites Pokémon obtidos em pokemondb[198]. Primeiro baixe, extraia e carregue este conjunto de dados.

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
pokemon = torchvision.datasets.ImageFolder(data_dir)
```

```
Downloading ../data/pokemon.zip from http://d2l-data.s3-accelerate.amazonaws.com/pokemon.zip.
↪..
```

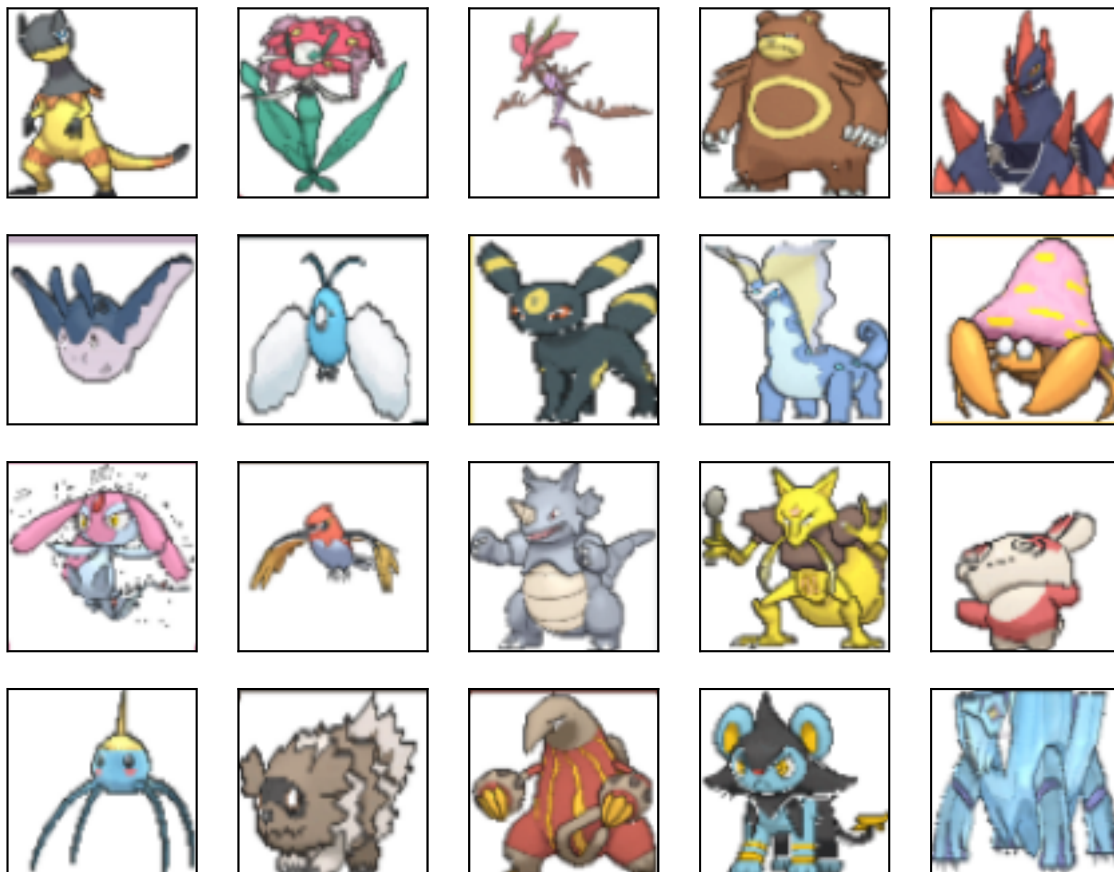
Redimensionamos cada imagem em 64×64. A transformação `ToTensor` projetará o valor do pixel em $[0, 1]$, enquanto nosso gerador usará a função tanh para obter saídas em $[-1, 1]$. Portanto, normalizamos os dados com $0,5$ de média e $0,5$ de desvio padrão para corresponder ao intervalo de valores.

```
batch_size = 256
transformer = torchvision.transforms.Compose([
    torchvision.transforms.Resize((64, 64)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(0.5, 0.5)
])
pokemon.transform = transformer
data_iter = torch.utils.data.DataLoader(
    pokemon, batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

[198] https://pokemondb.net/sprites

Vamos visualizar as primeiras 20 imagens.

```
warnings.filterwarnings('ignore')
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[0:20,:,:,:].permute(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```



17.2.2 O Gerador

O gerador precisa mapear a variável de ruído$\mathbf{z} \in \mathbb{R}^d$, um vetor comprimento d, para uma imagem RGB com largura e altura 64×64. Em Section 13.11 introduzimos a rede totalmente convolucional que usa a camada de convolução transposta (consulte Section 13.10) para aumentar o tamanho da entrada. O bloco básico do gerador contém uma camada de convolução transposta seguida pela normalização do lote e ativação ReLU.

```
class G_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.ConvTranspose2d(in_channels, out_channels,
                                kernel_size, strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
```

(continues on next page)

(continued from previous page)

```
        self.activation = nn.ReLU()

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

Por padrão, a camada de convolução transposta usa um kernel $k_h = k_w = 4$, passos $s_h = s_w = 2$ e um preenchimento $p_h = p_w = 1$. Com uma forma de entrada de $n_h^{'} \times n_w^{'} = 16 \times 16$, o bloco gerador dobrará a largura e a altura da entrada.

$$
\begin{aligned}
n_h^{'} \times n_w^{'} &= [(n_h k_h - (n_h-1)(k_h-s_h) - 2p_h] \times [(n_w k_w - (n_w-1)(k_w-s_w) - 2p_w]\\
&= [(k_h + s_h (n_h-1) - 2p_h] \times [(k_w + s_w (n_w-1) - 2p_w]\\
&= [(4 + 2 \times (16-1) - 2 \times 1] \times [(4 + 2 \times (16-1) - 2 \times 1]\\
&= 32 \times 32 .\\
\end{aligned}
\tag{17.2.1}
$$

```
x = torch.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk(x).shape
```

```
torch.Size([2, 20, 32, 32])
```

Se alterar a camada de convolução transposta para um kernel 4×4 kernel, passos 1×1 e preenchimento zero. Com um tamanho de entrada de 1×1, a saída terá sua largura e altura aumentadas em 3, respectivamente.

```
x = torch.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk(x).shape
```

```
torch.Size([2, 20, 4, 4])
```

O gerador consiste em quatro blocos básicos que aumentam a largura e a altura da entrada de 1 para 32. Ao mesmo tempo, ele primeiro projeta a variável latente em canais 64×8 e, em seguida, divide os canais a cada vez. Por fim, uma camada de convolução transposta é usada para gerar a saída. Ele ainda dobra a largura e a altura para corresponder à forma desejada de 64×64 e reduz o tamanho do canal para 3. A função de ativação tanh é aplicada aos valores de saída do projeto na faixa $(-1, 1)$.

```
n_G = 64
net_G = nn.Sequential(
    G_block(in_channels=100, out_channels=n_G*8,
            strides=1, padding=0),                  # Output: (64 * 8, 4, 4)
    G_block(in_channels=n_G*8, out_channels=n_G*4), # Output: (64 * 4, 8, 8)
    G_block(in_channels=n_G*4, out_channels=n_G*2), # Output: (64 * 2, 16, 16)
    G_block(in_channels=n_G*2, out_channels=n_G),   # Output: (64, 32, 32)
    nn.ConvTranspose2d(in_channels=n_G, out_channels=3,
                       kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh())  # Output: (3, 64, 64)
```

Gere uma variável latente de 100 dimensões para verificar a forma de saída do gerador.

```
x = torch.zeros((1, 100, 1, 1))
net_G(x).shape
```

```
torch.Size([1, 3, 64, 64])
```

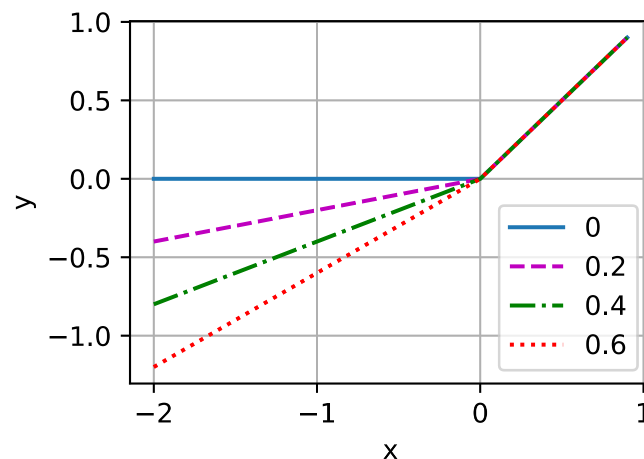
17.2.3 Discriminador

O discriminador é uma rede de rede convolucional normal, exceto que usa um ReLU com vazamento como sua função de ativação. Dado $\alpha \in [0, 1]$, sua definição é

$$\textrm{leaky ReLU}(x) = \begin{cases}x & \text{if}\ x > 0\\ \alpha x &\text{otherwise}\end{cases}. \tag{17.2.2}$$

Como pode ser visto, é um ReLU normal se $\alpha = 0$, e uma função de identidade se $\alpha = 1$. Para $\alpha \in (0, 1)$, o ReLU com vazamento é uma função não linear que fornece uma saída diferente de zero para uma entrada negativa. Seu objetivo é corrigir o problema de "ReLU agonizante" de que um neurônio pode sempre produzir um valor negativo e, portanto, não pode fazer nenhum progresso, pois o gradiente de ReLU é 0.

```
alphas = [0, .2, .4, .6, .8, 1]
x = torch.arange(-2, 1, 0.1)
Y = [nn.LeakyReLU(alpha)(x).detach().numpy() for alpha in alphas]
d2l.plot(x.detach().numpy(), Y, 'x', 'y', alphas)
```



O bloco básico do discriminador é uma camada de convolução seguida por uma camada de normalização em lote e uma ativação ReLU com vazamento. Os hiperparâmetros da camada de convolução são semelhantes à camada de convolução transposta no bloco gerador.

```
class D_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
```

(continues on next page)

(continued from previous page)

```
                                 strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.LeakyReLU(alpha, inplace=True)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

Um bloco básico com configurações padrão reduzirá pela metade a largura e a altura das entradas, como demonstramos em Section 6.3. Por exemplo, dada uma forma de entrada $n_h = n_w = 16$, com uma forma de kernel $k_h = k_w = 4$, uma forma de passo $s_h = s_w = 2$ e uma forma de preenchimento $p_h = p_w = 1$, a forma de saída será:

$$
\begin{aligned}
n_h^{'} \times n_w^{'} &= \lfloor(n_h - k_h + 2p_h + s_h)/s_h\rfloor \times \lfloor(n_w - k_w + 2p_w + s_w)/s_w\rfloor \\
&= \lfloor(16 - 4 + 2 \times 1 + 2)/2\rfloor \times \lfloor(16 - 4 + 2 \times 1 + 2)/2\rfloor \\
&= 8 \times 8.
\end{aligned}
\tag{17.2.3}
$$

```
x = torch.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk(x).shape
```

```
torch.Size([2, 20, 8, 8])
```

O discriminador é um espelho do gerador.

```
n_D = 64
net_D = nn.Sequential(
    D_block(n_D),  # Output: (64, 32, 32)
    D_block(in_channels=n_D, out_channels=n_D*2),  # Output: (64 * 2, 16, 16)
    D_block(in_channels=n_D*2, out_channels=n_D*4),  # Output: (64 * 4, 8, 8)
    D_block(in_channels=n_D*4, out_channels=n_D*8),  # Output: (64 * 8, 4, 4)
    nn.Conv2d(in_channels=n_D*8, out_channels=1,
              kernel_size=4, bias=False))  # Output: (1, 1, 1)
```

Ele usa uma camada de convolução com canal de saída 1 como a última camada para obter um único valor de predição.

```
x = torch.zeros((1, 3, 64, 64))
net_D(x).shape
```

```
torch.Size([1, 1, 1, 1])
```

17.2.4 Treinamento

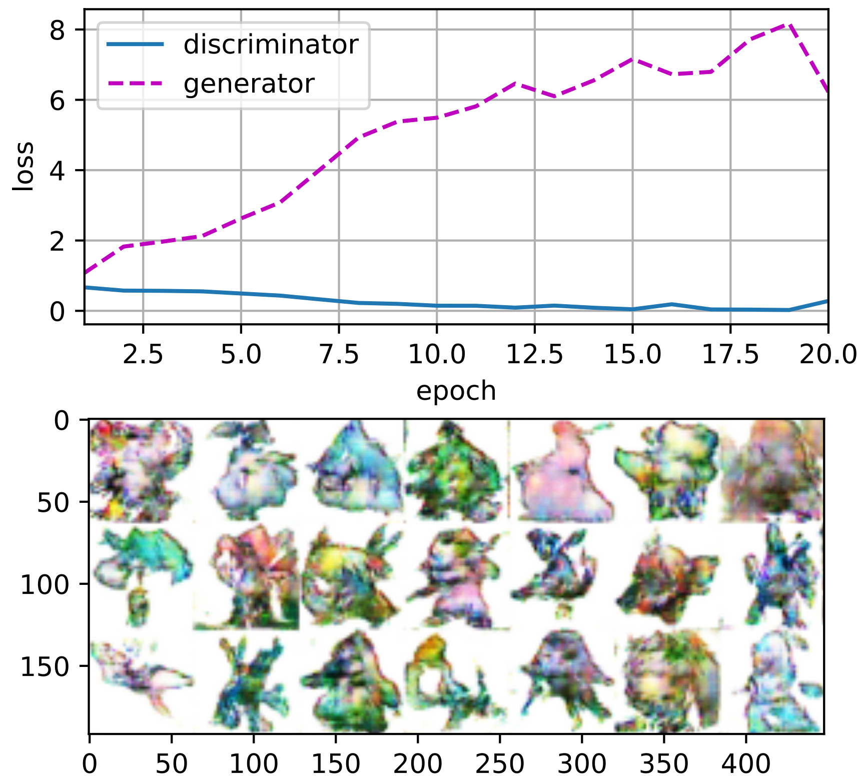
Comparado ao GAN básico em Section 17.1, usamos a mesma taxa de aprendizado para o gerador e o discriminador, uma vez que são semelhantes entre si. Além disso, alteramos β_1 em Adam (Section 11.10) de $0,9$ para $0,5$. Ele diminui a suavidade do momentum, a média móvel exponencialmente ponderada dos gradientes anteriores, para cuidar dos gradientes que mudam rapidamente porque o gerador e o discriminador lutam um com o outro. Além disso, o ruído gerado aleatoriamente Z, é um tensor 4-D e estamos usando GPU para acelerar o cálculo.

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    net_D, net_G = net_D.to(device), net_G.to(device)
    trainer_hp = {'lr': lr, 'betas': [0.5,0.999]}
    trainer_D = torch.optim.Adam(net_D.parameters(), **trainer_hp)
    trainer_G = torch.optim.Adam(net_G.parameters(), **trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs + 1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
            X, Z = X.to(device), Z.to(device)
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, trainer_D),
                       d2l.update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Show generated examples
        Z = torch.normal(0, 1, size=(21, latent_dim, 1, 1), device=device)
        # Normalize the synthetic data to N(0, 1)
        fake_x = net_G(Z).permute(0, 2, 3, 1) / 2 + 0.5
        imgs = torch.cat(
            [torch.cat([
                fake_x[i * 7 + j].cpu().detach() for j in range(7)], dim=1)
             for i in range(len(fake_x)//7)], dim=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs)
        # Show the losses
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec on {str(device)}')
```

Treinamos o modelo com um pequeno número de épocas apenas para demonstração. Para melhor desempenho, a variável `num_epochs` pode ser definida para um número maior.

```
latent_dim, lr, num_epochs = 100, 0.005, 20
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

```
loss_D 0.281, loss_G 6.224, 1057.2 examples/sec on cuda:0
```



17.2.5 Resumo

- A arquitetura DCGAN tem quatro camadas convolucionais para o Discriminador e quatro camadas convolucionais "com passos fracionários" para o Gerador.
- O Discriminador é um conjunto de convoluções de 4 camadas com normalização em lote (exceto sua camada de entrada) e ativações ReLU com vazamento.
- ReLU com vazamento é uma função não linear que fornece uma saída diferente de zero para uma entrada negativa. Ele tem como objetivo corrigir o problema do "ReLU agonizante" e ajudar os gradientes a fluírem mais facilmente pela arquitetura.

17.2.6 Exercícios

1. O que acontecerá se usarmos a ativação ReLU padrão em vez de ReLU com vazamento?
2. Aplique DCGAN no Fashion-MNIST e veja qual categoria funciona bem e qual não funciona.

Discussões[199]

[199] https://discuss.d2l.ai/t/1083

18 | Apêndice: Matemática para *Deep Learning*

Brent Werness (*Amazon*), **Rachel Hu** (*Amazon*), e autores deste livro

Uma das partes maravilhosas do *deep learning* moderno é o fato de que muito dele pode ser compreendido e usado sem uma compreensão completa da matemática abaixo dele. Isso é um sinal de que o campo está amadurecendo. Assim como a maioria dos desenvolvedores de *software* não precisa mais se preocupar com a teoria das funções computáveis, os profissionais de aprendizado profundo também não devem se preocupar com os fundamentos teóricos do aprendizado de máxima *likelihood*.

Mas ainda não chegamos lá.

Na prática, às vezes você precisará entender como as escolhas arquitetônicas influenciam o fluxo de gradiente ou as suposições implícitas que você faz ao treinar com uma determinada função de perda. Você pode precisar saber o que mede a entropia mundial e como isso pode ajudá-lo a entender exatamente o que cada bit por caractere significam em seu modelo. Tudo isso requer um conhecimento matemático mais profundo.

Este apêndice tem como objetivo fornecer a base matemática que você precisa para entender a teoria central do *deep learning*, moderno, mas não é exaustivo. Começaremos examinando a álgebra linear em maior profundidade. Desenvolvemos uma compreensão geométrica de todos os objetos algébricos lineares comuns e operações que nos permitirão visualizar os efeitos de várias transformações em nossos dados. Um elemento chave é o desenvolvimento das noções básicas de autodescomposições.

Em seguida, desenvolvemos a teoria do cálculo diferencial até o ponto em que podemos entender completamente por que o gradiente é a direção de descida mais acentuada e por que a retropropagação assume a forma que assume. O cálculo integral é então discutido no grau necessário para apoiar nosso próximo tópico, a teoria da probabilidade.

Os problemas encontrados na prática frequentemente não são certos e, portanto, precisamos de uma linguagem para falar sobre coisas incertas. Revisamos a teoria das variáveis aleatórias e as distribuições mais comumente encontradas para que possamos discutir os modelos de forma probabilística. Isso fornece a base para o classificador *naive* Bayes, uma técnica de classificação probabilística.

Intimamente relacionado à teoria da probabilidade está o estudo da estatística. Embora as estatísticas sejam um campo muito grande para fazer justiça em uma seção curta, apresentaremos conceitos fundamentais que todos os praticantes de aprendizado de máquina devem estar cientes, em particular: avaliação e comparação de estimadores, realização de testes de hipótese e construção de intervalos de confiança.

Por último, voltamos ao tópico da teoria da informação, que é o estudo matemático do armazenamento e transmissão da informação. Isso fornece a linguagem central pela qual podemos discutir quantitativamente quanta informação um modelo contém em um domínio do discurso.

Juntos, eles formam o núcleo dos conceitos matemáticos necessários para iniciar o caminho em direção a uma compreensão profunda do *deep learning*.

18.1 Operações de Geometria e Álgebra Linear

Em Section 2.3, encontramos os fundamentos da álgebra linear e vimos como ela poderia ser usada para expressar operações comuns para transformar nossos dados. A álgebra linear é um dos principais pilares matemáticos subjacentes a grande parte do trabalho que fazemos no *deep learning* e no *machine learning* de forma mais ampla. Embora Section 2.3 contenha maquinário suficiente para comunicar a mecânica dos modelos modernos de aprendizado profundo, há muito mais sobre o assunto. Nesta seção, iremos mais fundo, destacando algumas interpretações geométricas de operações de álgebra linear e introduzindo alguns conceitos fundamentais, incluindo de autovalores e autovetores.

18.1.1 Geometria Vetorial

Primeiro, precisamos discutir as duas interpretações geométricas comuns de vetores, como pontos ou direções no espaço. Fundamentalmente, um vetor é uma lista de números, como a lista Python abaixo.

```
v = [1, 7, 0, 1]
```

Os matemáticos geralmente escrevem isso como um vetor *coluna* ou *linha*, ou seja, como

$$\mathbf{x} = \begin{bmatrix}1\\7\\0\\1\end{bmatrix}, \tag{18.1.1}$$

ou

$$\mathbf{x}^\top = \begin{bmatrix}1 & 7 & 0 & 1\end{bmatrix}. \tag{18.1.2}$$

Muitas vezes têm interpretações diferentes, onde os exemplos de dados são vetores de coluna e os pesos usados para formar somas ponderadas são vetores de linha. No entanto, pode ser benéfico ser flexível. Como descrevemos em Section 2.3, embora a orientação padrão de um único vetor seja um vetor coluna, para qualquer matriz que representa um conjunto de dados tabular, tratando cada exemplo de dados como um vetor de linha na matriz é mais convencional.

Dado um vetor, a primeira interpretação que devemos dar é como um ponto no espaço. Em duas ou três dimensões, podemos visualizar esses pontos usando os componentes dos vetores para definir a localização dos pontos no espaço comparada a uma referência fixa chamada *origem*. Isso pode ser visto em Fig. 18.1.1.

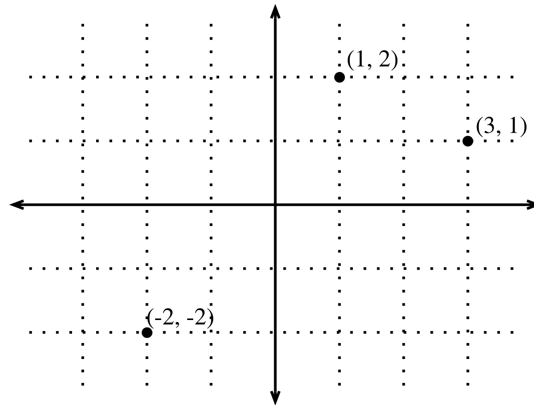


Fig. 18.1.1: Uma ilustração da visualização de vetores como pontos no plano. O primeiro componente do vetor fornece a coordenada x, o segundo componente fornece a coordenada y. As dimensões superiores são análogas, embora muito mais difíceis de visualizar.

Esse ponto de vista geométrico nos permite considerar o problema em um nível mais abstrato. Não mais confrontado com algum problema aparentemente intransponível como classificar fotos como gatos ou cachorros, podemos começar a considerar as tarefas abstratamente como coleções de pontos no espaço e retratando a tarefa como descobrir como separar dois grupos distintos de pontos.

Paralelamente, existe um segundo ponto de vista que as pessoas costumam tomar de vetores: como direções no espaço. Não podemos apenas pensar no vetor $\mathbf{v} = [3, 2]^\top$ como a localização 3 unidades à direita e 2 unidades acima da origem, também podemos pensar nisso como a própria direção para mover 3 passos para a direita e 2 para cima. Desta forma, consideramos todos os vetores da figura Fig. 18.1.2 iguais.

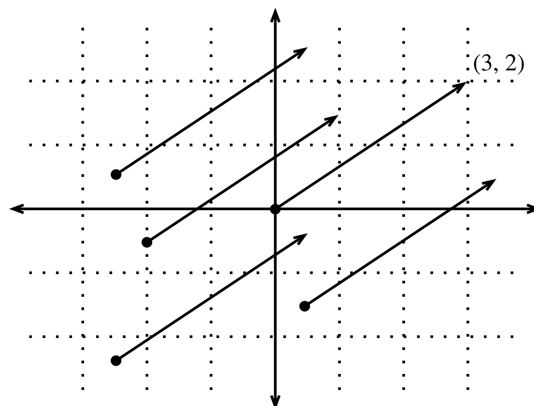


Fig. 18.1.2: Qualquer vetor pode ser visualizado como uma seta no plano. Nesse caso, todo vetor desenhado é uma representação do vetor $(3, 2)^\top$.

Um dos benefícios dessa mudança é que podemos dar sentido visual ao ato de adição de vetores. Em particular, seguimos as instruções dadas por um vetor, e então siga as instruções dadas pelo outro, como pode ser visto em Fig. 18.1.3.

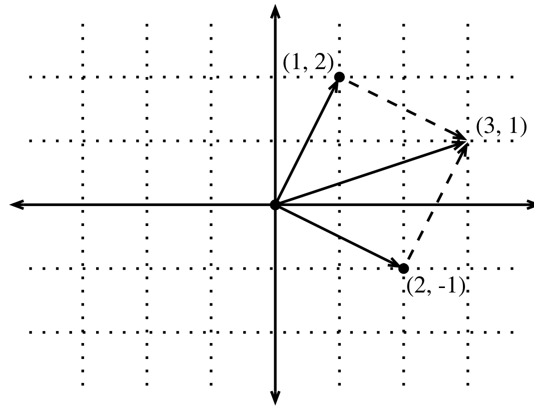


Fig. 18.1.3: Podemos visualizar a adição de vetores seguindo primeiro um vetor e depois outro.

A subtração de vetores tem uma interpretação semelhante. Considerando a identidade que $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$, vemos que o vetor $\mathbf{u} - \mathbf{v}$ é a direção que nos leva do ponto $\mathbf{v}$ ao ponto $\mathbf{u}$.

18.1.2 Produto Escalar e Ângulos

Como vimos em Section 2.3, se tomarmos dois vetores de coluna $\mathbf{u}$ and $\mathbf{v}$, podemos formar seu produto escalar computando:

$$\mathbf{u}^\top\mathbf{v} = \sum_i u_i \cdot v_i. \tag{18.1.3}$$

Porque (18.1.3) é simétrico, iremos espelhar a notação de multiplicação clássica e escrita

$$\mathbf{u}\cdot\mathbf{v} = \mathbf{u}^\top\mathbf{v} = \mathbf{v}^\top\mathbf{u}, \tag{18.1.4}$$

para destacar o fato de que a troca da ordem dos vetores produzirá a mesma resposta.

The dot product (18.1.3) also admits a geometric interpretation: it is closely related to the angle between two vectors. Consider the angle shown in Fig. 18.1.4.

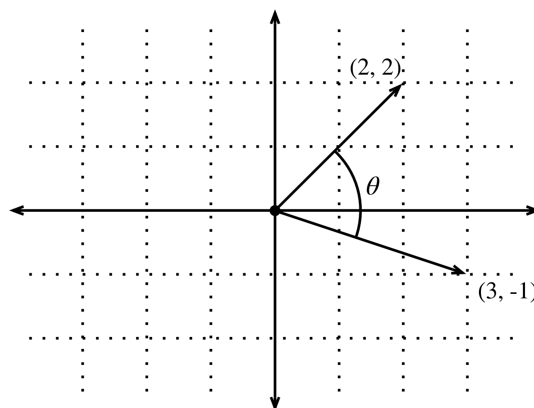


Fig. 18.1.4: Entre quaisquer dois vetores no plano, existe um ângulo bem definido θ. Veremos que esse ângulo está intimamente ligado ao produto escalar.

Para começar, vamos considerar dois vetores específicos:

$$\mathbf{v} = (r, 0) \text{ and } \mathbf{w} = (s\cos(\theta), s\sin(\theta)). \tag{18.1.5}$$

O vetor $\mathbf{v}$ tem comprimento r e corre paralelo ao eixo x, e o vetor $\mathbf{w}$ tem comprimento s e está no ângulo θ com o eixo x. Se calcularmos o produto escalar desses dois vetores, vemos que

$$\mathbf{v} \cdot \mathbf{w} = rs\cos(\theta) = \|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta). \tag{18.1.6}$$

Com alguma manipulação algébrica simples, podemos reorganizar os termos para obter

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{18.1.7}$$

Em suma, para esses dois vetores específicos, o produto escalar combinado com as normas nos informa o ângulo entre os dois vetores. Este mesmo fato é verdade em geral. Não iremos derivar a expressão aqui, no entanto, se considerarmos escrever $\|\mathbf{v} - \mathbf{w}\|^2$ de duas maneiras: um com o produto escalar e o outro geometricamente usando a lei dos cossenos, podemos obter o relacionamento completo. Na verdade, para quaisquer dois vetores $\mathbf{v}$ e $\mathbf{w}$, o ângulo entre os dois vetores é

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{18.1.8}$$

Este é um bom resultado, pois nada no cálculo faz referência a duas dimensões. Na verdade, podemos usar isso em três ou três milhões de dimensões sem problemas.

Como um exemplo simples, vamos ver como calcular o ângulo entre um par de vetores:

```
%matplotlib inline
import torch
import torchvision
from IPython import display
from torchvision import transforms
from d2l import torch as d2l


def angle(v, w):
    return torch.acos(v.dot(w) / (torch.norm(v) * torch.norm(w)))

angle(torch.tensor([0, 1, 2], dtype=torch.float32), torch.tensor([2.0, 3, 4]))
```

```
tensor(0.4190)
```

Não o usaremos agora, mas é útil saber que iremos nos referir a vetores para os quais o ângulo é $\pi/2$ (ou equivalentemente $90°$) como sendo *ortogonal*. Examinando a equação acima, vemos que isso acontece quando $\theta = \pi/2$, que é a mesma coisa que $\cos(\theta) = 0$. A única maneira de isso acontecer é se o produto escalar em si for zero, e dois vetores são ortogonais se e somente se $\mathbf{v} \cdot \mathbf{w} = 0$. Esta será uma fórmula útil para compreender objetos geometricamente.

É razoável perguntar: por que calcular o ângulo é útil? A resposta vem no tipo de invariância que esperamos que os dados tenham. Considere uma imagem e uma imagem duplicada, onde cada valor de pixel é o mesmo, mas com 10% do brilho. Os valores dos pixels individuais estão geralmente longe dos valores originais. Assim, se computarmos a distância entre a imagem original e a mais escura, a distância pode ser grande. No entanto, para a maioria dos aplicativos de ML, o *conteúdo* é o mesmo — ainda é uma imagem de um gato no que diz respeito a um classificador gato / cão. No entanto, se considerarmos o ângulo, não é difícil ver que para qualquer vetor $\mathbf{v}$, o ângulo entre $\mathbf{v}$ e $0.1 \cdot \mathbf{v}$ é zero. Isso corresponde ao fato de que os vetores de escala mantém a mesma direção e apenas altera o comprimento. O ângulo considera a imagem mais escura idêntica.

Exemplos como este estão por toda parte. No texto, podemos querer que o tópico seja discutido para não mudar se escrevermos o dobro do tamanho do documento que diz a mesma coisa. Para algumas codificações (como contar o número de ocorrências de palavras em algum vocabulário), isso corresponde a uma duplicação do vetor que codifica o documento, então, novamente, podemos usar o ângulo.

Semelhança de Cosseno

Em contextos de ML onde o ângulo é empregado para medir a proximidade de dois vetores, os profissionais adotam o termo *semelhança de cosseno* para se referir à porção

$$\cos(\theta) = \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}. \tag{18.1.9}$$

O cosseno assume um valor máximo de 1 quando os dois vetores apontam na mesma direção, um valor mínimo de -1 quando apontam em direções opostas, e um valor de 0 quando os dois vetores são ortogonais. Observe que se os componentes de vetores de alta dimensão são amostrados aleatoriamente com 0 médio, seu cosseno será quase sempre próximo a 0.

18.1.3 Hiperplanos

Além de trabalhar com vetores, outro objeto-chave que você deve entender para ir longe na álgebra linear é o *hiperplano*, uma generalização para dimensões superiores de uma linha (duas dimensões) ou de um plano (três dimensões). Em um espaço vetorial d-dimensional, um hiperplano tem $d-1$ dimensões e divide o espaço em dois meios-espaços.

Vamos começar com um exemplo. Suponha que temos um vetor coluna $\mathbf{w} = [2,1]^\top$. Queremos saber, "quais são os pontos $\mathbf{v}$ com $\mathbf{w}\cdot\mathbf{v} = 1$?" Ao relembrar a conexão entre produtos escalares e ângulos acima (18.1.8), podemos ver que isso é equivalente a

$$\|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta) = 1 \iff \|\mathbf{v}\|\cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \tag{18.1.10}$$

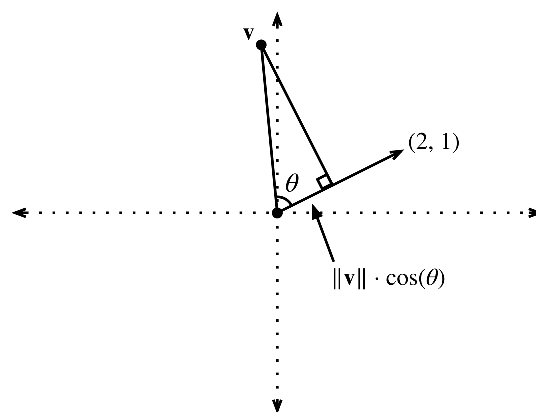


Fig. 18.1.5: Relembrando a trigonometria, vemos que a fórmula $\|\mathbf{v}\|\cos(\theta)$ é o comprimento da projeção do vetor $\mathbf{v}$ na direção de $\mathbf{w}$

Se considerarmos o significado geométrico desta expressão, vemos que isso é equivalente a dizer que o comprimento da projeção de $\mathbf{v}$ na direção de $\mathbf{w}$ é exatamente $1/\|\mathbf{w}\|$, como é mostrado em

Fig. 18.1.5. O conjunto de todos os pontos onde isso é verdade é uma linha perpendicularmente ao vetor **w**. Se quiséssemos, poderíamos encontrar a equação para esta linha e veja que é $2x + y = 1$ ou equivalentemente $y = 1 - 2x$.

Se agora olharmos para o que acontece quando perguntamos sobre o conjunto de pontos com $\mathbf{w} \cdot \mathbf{v} > 1$ ou $\mathbf{w} \cdot \mathbf{v} < 1$, podemos ver que estes são casos em que as projeções são maiores ou menores que $1/\|\mathbf{w}\|$, respectivamente. Portanto, essas duas desigualdades definem os dois lados da linha. Desta forma, descobrimos uma maneira de cortar nosso espaço em duas metades, onde todos os pontos de um lado têm produto escalar abaixo de um limite, e o outro lado acima como vemos em Fig. 18.1.6.

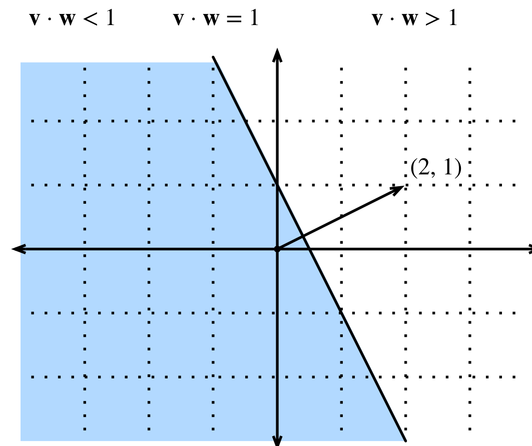


Fig. 18.1.6: Se considerarmos agora a versão da desigualdade da expressão, vemos que nosso hiperplano (neste caso: apenas uma linha) separa o espaço em duas metades.

A história em uma dimensão superior é praticamente a mesma. Se agora tomarmos $\mathbf{w} = [1, 2, 3]^\top$ e perguntarmos sobre os pontos em três dimensões com $\mathbf{w} \cdot \mathbf{v} = 1$, obtemos um plano perpendicular ao vetor dado **w**. As duas desigualdades definem novamente os dois lados do plano como é mostrado em Fig. 18.1.7.

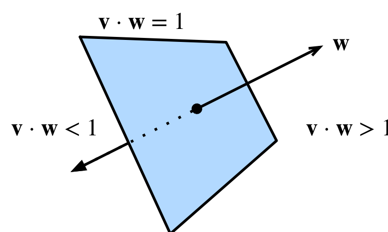


Fig. 18.1.7: Hiperplanos em qualquer dimensão separam o espaço em duas metades.

Embora nossa capacidade de visualizar se esgote neste ponto, nada nos impede de fazer isso em dezenas, centenas ou bilhões de dimensões. Isso ocorre frequentemente quando se pensa em modelos aprendidos por máquina. Por exemplo, podemos entender modelos de classificação linear como aqueles de Section 3.4, como métodos para encontrar hiperplanos que separam as diferentes classes de destino. Nesse contexto, esses hiperplanos são freqüentemente chamados de *planos de decisão*. A maioria dos modelos de classificação profundamente aprendidos termina com uma camada linear alimentada em um *softmax*, para que se possa interpretar o papel da rede neural profunda encontrar uma incorporação não linear de modo que as classes de destino podem

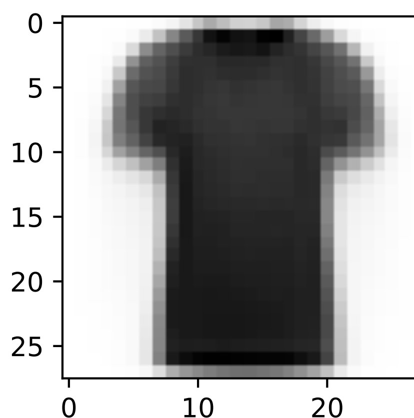
ser separados de forma limpa por hiperplanos.

Para dar um exemplo feito à mão, observe que podemos produzir um modelo razoável para classificar pequenas imagens de camisetas e calças do conjunto de dados do Fashion MNIST (visto em Section 3.5) apenas pegando o vetor entre seus meios para definir o plano de decisão e olho um limiar bruto. Primeiro, carregaremos os dados e calcularemos as médias.

```
# Load in the dataset
trans = []
trans.append(transforms.ToTensor())
trans = transforms.Compose(trans)
train = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                          train=True, download=True)
test = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                         train=False, download=True)

X_train_0 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 0]).type(torch.float32)
X_train_1 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 1]).type(torch.float32)
X_test = torch.stack(
    [x[0] * 256 for x in test if x[1] == 0 or x[1] == 1]).type(torch.float32)
y_test = torch.stack([torch.tensor(x[1]) for x in test
                      if x[1] == 0 or x[1] == 1]).type(torch.float32)

# Compute averages
ave_0 = torch.mean(X_train_0, axis=0)
ave_1 = torch.mean(X_train_1, axis=0)
```

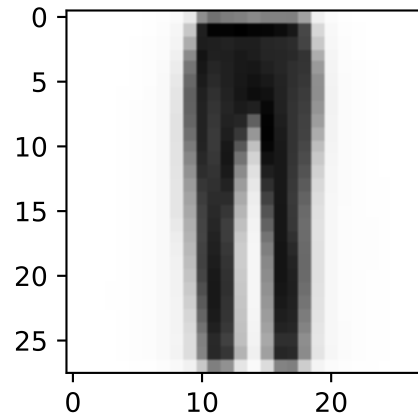
Pode ser informativo examinar essas médias em detalhes, portanto, vamos representar graficamente sua aparência. Nesse caso, vemos que a média realmente se assemelha a uma imagem borrada de uma camiseta.

```
# Plot average t-shirt
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



No segundo caso, vemos novamente que a média se assemelha a uma imagem borrada de calças.

```python
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



Em uma solução totalmente aprendida pela máquina, aprenderíamos o limite do conjunto de dados. Nesse caso, simplesmente analisamos um limite que parecia bom nos dados de treinamento à mão.

```python
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
# '@' is Matrix Multiplication operator in pytorch.
predictions = X_test.reshape(2000, -1) @ (w.flatten()) > -1500000

# Accuracy
torch.mean(predictions.type(y_test.dtype) == y_test, dtype=torch.float64)
```

```
tensor(0.7870, dtype=torch.float64)
```

18.1.4 Geometria de Transformações Lineares

Por meio de Section 2.3 e das discussões acima, temos um conhecimento sólido da geometria de vetores, comprimentos e ângulos. No entanto, há um objeto importante que omitimos de discutir, e essa é uma compreensão geométrica das transformações lineares representadas por matrizes. Totalmente internalizando o que as matrizes podem fazer para transformar dados entre dois espaços de dimensões elevadas potencialmente diferentes requer prática significativa, e está além do escopo deste apêndice. No entanto, podemos começar a construir a intuição em duas dimensões.

Suponha que temos alguma matriz:

$$\mathbf{A} = \begin{bmatrix}
a & b \\ c & d
\end{bmatrix}. \tag{18.1.11}$$

Se quisermos aplicar isso a um vetor arbitrário $\mathbf{v} = [x, y]^\top$, nós nos multiplicamos e vemos que

$$\begin{aligned}
\mathbf{A}\mathbf{v} &= \begin{bmatrix}a & b\\ c & d\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix} \\
&= \begin{bmatrix}ax+by\\ cx+dy\end{bmatrix} \\
&= x\begin{bmatrix}a\\ c\end{bmatrix} + y\begin{bmatrix}b\\ d\end{bmatrix} \\
&= x\left\{\mathbf{A}\begin{bmatrix}1\\ 0\end{bmatrix}\right\} + y\left\{\mathbf{A}\begin{bmatrix}0\\ 1\end{bmatrix}\right\}.
\end{aligned} \tag{18.1.12}$$

Isso pode parecer um cálculo estranho, onde algo claro se tornou algo impenetrável. No entanto, isso nos diz que podemos escrever da maneira que uma matriz transforma *qualquer* vetor em termos de como ele transforma *dois vetores específicos*: $[1,0]^\top$ and $[0,1]^\top$. Vale a pena considerar isso por um momento. Nós essencialmente reduzimos um problema infinito (o que acontece com qualquer par de números reais) para um finito (o que acontece com esses vetores específicos). Esses vetores são um exemplo de *vetores canônicos*, onde podemos escrever qualquer vetor em nosso espaço como uma soma ponderada desses *vetores canônicos*.

Vamos desenhar o que acontece quando usamos a matriz específica

$$\mathbf{A} = \begin{bmatrix}1 & 2\\ -1 & 3\end{bmatrix}. \tag{18.1.13}$$

Se olharmos para o vetor específico $\mathbf{v} = [2, -1]^\top$, vemos que é $2 \cdot [1,0]^\top + -1 \cdot [0,1]^\top$, e assim sabemos que a matriz A irá enviar isso para $2(\mathbf{A}[1,0]^\top) + -1(\mathbf{A}[0,1])^\top = 2[1,-1]^\top - [2,3]^\top = [0,-5]^\top$. Se seguirmos essa lógica com cuidado, digamos, considerando a grade de todos os pares inteiros de pontos, vemos que o que acontece é que a multiplicação da matriz pode inclinar, girar e dimensionar a grade, mas a estrutura da grade deve permanecer como você vê em Fig. 18.1.8.

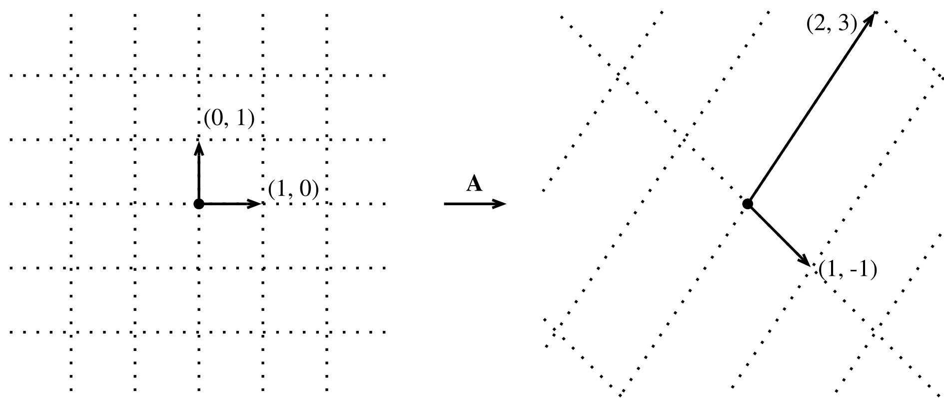


Fig. 18.1.8: A matriz **A** agindo nos vetores de base dados. Observe como toda a grade é transportada junto com ela.

Este é o ponto intuitivo mais importante para internalizar sobre transformações lineares representadas por matrizes. As matrizes são incapazes de distorcer algumas partes do espaço de maneira diferente de outras. Tudo o que elas podem fazer é pegar as coordenadas originais em nosso espaço e inclinar, girar e dimensioná-las.

Algumas distorções podem ser graves. Por exemplo, a matriz

$$\mathbf{B} = \begin{bmatrix}2 & -1\\ 4 & -2\end{bmatrix}, \tag{18.1.14}$$

comprime todo o plano bidimensional em uma única linha. Identificar e trabalhar com essas transformações é o tópico de uma seção posterior, mas geometricamente podemos ver que isso é fundamentalmente diferente dos tipos de transformações que vimos acima. Por exemplo, o resultado da matriz **A** pode ser "dobrado" para a grade original. Os resultados da matriz **B** não podem porque nunca saberemos de onde o vetor $[1, 2]^\top$ veio — estava it $[1, 1]^\top$ ou $[0, -1]^\top$?

Embora esta imagem fosse para uma matriz 2×2, nada nos impede de levar as lições aprendidas para dimensões superiores. Se tomarmos vetores de base semelhantes como $[1, 0, \ldots, 0]$ e ver para onde nossa matriz os envia, podemos começar a ter uma ideia de como a multiplicação da matriz distorce todo o espaço em qualquer dimensão de espaço com a qual estamos lidando.

18.1.5 Dependência Linear

Considere novamente a matriz

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}. \tag{18.1.15}$$

Isso comprime todo o plano para viver na única linha $y = 2x$. A questão agora surge: há alguma maneira de detectarmos isso apenas olhando para a própria matriz? A resposta é que realmente podemos. Tomemos $\mathbf{b}_1 = [2, 4]^\top$ e $\mathbf{b}_2 = [-1, -2]^\top$ sejam as duas colunas de **B**. Lembre-se de que podemos escrever tudo transformado pela matriz **B** como uma soma ponderada das colunas da matriz: como $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$. Chamamos isso de *combinação linear*. O fato de $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ significa que podemos escrever qualquer combinação linear dessas duas colunas inteiramente em termos de, digamos, $\mathbf{b}_2$ desde

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \tag{18.1.16}$$

Isso significa que uma das colunas é, de certo modo, redundante porque não define uma direção única no espaço. Isso não deve nos surpreender muito pois já vimos que esta matriz reduz o plano inteiro em uma única linha. Além disso, vemos que a dependência linear $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ captura isso. Para tornar isso mais simétrico entre os dois vetores, vamos escrever isso como

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \tag{18.1.17}$$

Em geral, diremos que uma coleção de vetores $\mathbf{v}_1, \ldots, \mathbf{v}_k$ são *linearmente dependentes* se existirem coeficientes $a_1, \ldots, a_k$ *nem todos iguais a zero* de modo que

$$\sum_{i=1}^{k} a_i\mathbf{v_i} = 0. \tag{18.1.18}$$

Neste caso, podemos resolver para um dos vetores em termos de alguma combinação dos outros, e efetivamente torná-lo redundante. Assim, uma dependência linear nas colunas de uma matriz é uma testemunha do fato de que nossa matriz está comprimindo o espaço para alguma dimensão inferior. Se não houver dependência linear, dizemos que os vetores são *linearmente independentes*. Se as colunas de uma matriz são linearmente independentes, nenhuma compressão ocorre e a operação pode ser desfeita.

18.1.6 Classificação

Se tivermos uma matriz geral $n \times m$, é razoável perguntar em qual espaço de dimensão a matriz mapeia. Um conceito conhecido como classificação será a nossa resposta. Na seção anterior, notamos que uma dependência linear testemunha a compressão do espaço em uma dimensão inferior e assim seremos capazes de usar isso para definir a noção de posto. Em particular, a classificação de uma matriz $\mathbf{A}$ é o maior número de colunas linearmente independentes entre todos os subconjuntos de colunas. Por exemplo, a matriz

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}, \tag{18.1.19}$$

tem $\operatorname{rank}(B) = 1$, uma vez que as duas colunas são linearmente dependentes, mas qualquer coluna por si só não é linearmente dependente. Para um exemplo mais desafiador, podemos considerar

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}, \tag{18.1.20}$$

e mostrar que $\mathbf{C}$ tem classificação dois, uma vez que, por exemplo, as duas primeiras colunas são linearmente independentes, entretanto, qualquer uma das quatro coleções de três colunas é dependente.

Este procedimento, conforme descrito, é muito ineficiente. Requer olhar para cada subconjunto das colunas de nossa matriz, e, portanto, é potencialmente exponencial no número de colunas. Mais tarde, veremos uma forma mais eficiente do ponto de vista computacional para calcular a classificação de uma matriz, mas por enquanto, isso é suficiente para ver que o conceito está bem definido e compreende o significado.

18.1.7 Invertibilidade

Vimos acima que a multiplicação por uma matriz com colunas linearmente dependentes não pode ser desfeita, ou seja, não há operação inversa que sempre pode recuperar a entrada. No entanto, a multiplicação por uma matriz de classificação completa (ou seja, algum $\mathbf{A}$ que é $n \times n$ matriz com classificação n), devemos sempre poder desfazê-lo. Considere a matriz

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{18.1.21}$$

que é a matriz com uns ao longo da diagonal e zeros em outros lugares. Chamamos isso de matriz de * identidade *. É a matriz que deixa nossos dados inalterados quando aplicados. Para encontrar uma matriz que desfaça o que nossa matriz $\mathbf{A}$ fez, queremos encontrar uma matriz $\mathbf{A}^{-1}$ tal que

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{18.1.22}$$

Se olharmos para isso como um sistema, temos $n \times n$ incógnitas (as entradas de $\mathbf{A}^{-1}$) e $n \times n$ equações (a igualdade que precisa ser mantida entre cada entrada do produto $\mathbf{A}^{-1}\mathbf{A}$ e cada entrada de $\mathbf{I}$) portanto, devemos genericamente esperar que exista uma solução. Na verdade, na próxima

seção, veremos uma quantidade chamada de *determinante*, que tem a propriedade de que, desde que o determinante não seja zero, podemos encontrar uma solução. Chamamos tal matriz $\mathbf{A}^{-1}$ de matriz *inversa*. Por exemplo, se $\mathbf{A}$ é a matriz 2×2 geral

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{18.1.23}$$

então podemos ver que a inversa é

$$\frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{18.1.24}$$

Podemos testar para ver isso vendo que a multiplicação pela inversa dado pela fórmula acima funciona na prática.

```
M = torch.tensor([[1, 2], [1, 4]], dtype=torch.float32)
M_inv = torch.tensor([[2, -1], [-0.5, 0.5]])
M_inv @ M
```

```
tensor([[1., 0.],
        [0., 1.]])
```

Problemas Numéricos

Embora o inverso de uma matriz seja útil em teoria, devemos dizer que na maioria das vezes não desejamos *usar* a matriz inversa para resolver um problema na prática. Em geral, existem algoritmos muito mais estáveis numericamente para resolver equações lineares como

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{18.1.25}$$

do que calcular a inversa e multiplicar para obter

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{18.1.26}$$

Assim como a divisão por um pequeno número pode levar à instabilidade numérica, o mesmo pode acontecer com a inversão de uma matriz que está perto de ter uma classificação baixa.

Além disso, é comum que a matriz $\mathbf{A}$ seja *esparsa*, o que significa que ele contém apenas um pequeno número de valores diferentes de zero. Se fossemos explorar exemplos, veríamos que isso não significa que o inverso é esparso. Mesmo se $\mathbf{A}$ fosse uma matriz de 1 milhão por 1 milhão com apenas 5 milhões de entradas diferentes de zero (e, portanto, precisamos apenas armazenar aqueles 5 milhões), a inversa normalmente terá quase todas as entradas não negativas, exigindo que armazenemos todas as 1M^2 de entradas — isto é 1 trilhão de entradas!

Embora não tenhamos tempo para mergulhar totalmente nas espinhosas questões numéricas frequentemente encontrados ao trabalhar com álgebra linear, queremos fornecer-lhe alguma intuição sobre quando proceder com cautela, e geralmente evitar a inversão na prática é uma boa regra prática.

18.1.8 Determinante

A visão geométrica da álgebra linear oferece uma maneira intuitiva para interpretar uma quantidade fundamental conhecida como *determinante*. Considere a imagem da grade de antes, mas agora com uma região destacada (`fig_grid-fill`).

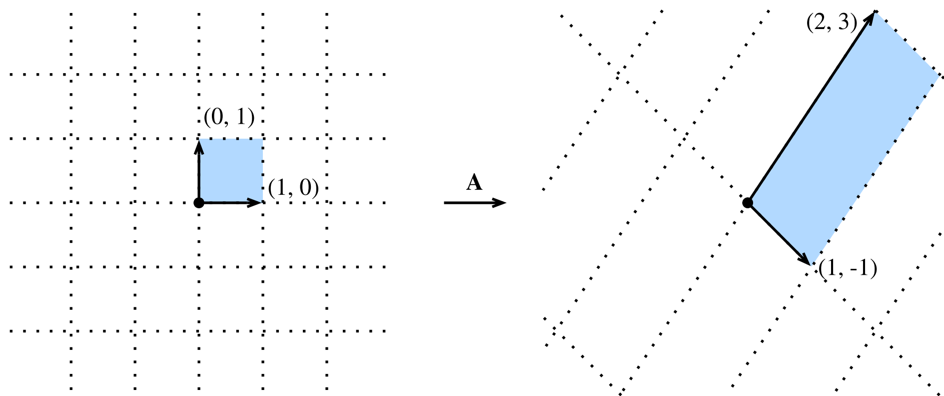


Fig. 18.1.9: A matriz **A** novamente distorcendo a grade. Desta vez, quero chamar a atenção em particular para o que acontece com o quadrado destacado.

Olhe para o quadrado destacado. Este é um quadrado com bordas fornecidas por $(0, 1)$ e $(1, 0)$ e, portanto, tem área um. Depois que **A** transforma este quadrado, vemos que se torna um paralelogramo. Não há razão para este paralelogramo ter a mesma área com que começamos, e de fato no caso específico mostrado aqui de

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}, \tag{18.1.27}$$

é um exercício de geometria coordenada para calcular a área deste paralelogramo e obtenha que a área é 5.

Em geral, se tivermos uma matriz

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{18.1.28}$$

podemos ver com alguns cálculos que a área do paralelogramo resultante é $ad - bc$. Essa área é chamada de *determinante*.

Vamos verificar isso rapidamente com algum código de exemplo.

```
torch.det(torch.tensor([[1, -1], [2, 3]], dtype=torch.float32))
```

```
tensor(5.)
```

Os olhos de águia entre nós notarão que esta expressão pode ser zero ou mesmo negativa. Para o termo negativo, isso é uma questão de convenção geralmente considerado em matemática: se a matriz inverte a figura, dizemos que a área está negada. Vejamos agora que quando o determinante é zero, aprendemos mais.

Vamos considerar

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \tag{18.1.29}$$

Se calcularmos o determinante desta matriz, obtemos $2 \cdot (-2) - 4 \cdot (-1) = 0$. Dado o nosso entendimento acima, isso faz sentido. **B** comprime o quadrado da imagem original até um segmento de linha, que tem área zero. E, de fato, sendo comprimido em um espaço dimensional inferior é a única maneira de ter área zero após a transformação. Assim, vemos que o seguinte resultado é verdadeiro: uma matriz A é invertível se e somente se o determinante não é igual a zero.

Como comentário final, imagine que temos alguma figura desenhada no avião. Pensando como cientistas da computação, podemos decompor aquela figura em uma coleção de pequenos quadrados de modo que a área da figura é em essência apenas o número de quadrados na decomposição. Se agora transformarmos essa figura em uma matriz, enviamos cada um desses quadrados para paralelogramos, cada um deles tem área dada pelo determinante. Vemos que para qualquer figura, o determinante dá o número (com sinal) que uma matriz dimensiona a área de qualquer figura.

Determinantes de computação para matrizes maiores podem ser trabalhosos, mas a intuição é a mesma. O determinante continua sendo o fator que $n \times n$ matrizes escalam volumes n-dimensionais.

18.1.9 Tensores e Operações de Álgebra Linear Comum

Em Section 2.3 o conceito de tensores foi introduzido. Nesta seção, vamos mergulhar mais profundamente nas contrações tensoras (o tensor equivalente à multiplicação da matriz), e ver como pode fornecer uma visão unificada em uma série de operações de matriz e vetor.

Com matrizes e vetores, sabíamos como multiplicá-los para transformar os dados. Precisamos ter uma definição semelhante para tensores se eles forem úteis para nós. Pense na multiplicação de matrizes:

$$\mathbf{C} = \mathbf{AB}, \tag{18.1.30}$$

ou equivalente

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \tag{18.1.31}$$

Esse padrão pode ser repetido para tensores. Para tensores, não há um caso de qual para somar isso pode ser universalmente escolhido, portanto, precisamos especificar exatamente quais índices queremos somar. Por exemplo, podemos considerar

$$y_{il} = \sum_{jk} x_{ijkl} a_{jk}. \tag{18.1.32}$$

Essa transformação é chamada de *contração tensorial*. Pode representar uma família de transformações muito mais flexível que a multiplicação de matriz sozinha.

Como uma simplificação de notação frequentemente usada, podemos notar que a soma está exatamente acima desses índices que ocorrem mais de uma vez na expressão, assim, as pessoas costumam trabalhar com *notação de Einstein*, onde o somatório é implicitamente assumido sobre todos os índices repetidos. Isso dá a expressão compacta:

$$y_{il} = x_{ijkl} a_{jk}. \tag{18.1.33}$$

Exemplos Comuns de Álgebra Linear

Vamos ver quantas das definições algébricas lineares que vimos antes pode ser expresso nesta notação de tensor compactado:

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_i v_i$
- $(\mathbf{A}\mathbf{v})_i = \sum_j a_{ij} v_j$
- $(\mathbf{A}\mathbf{B})_{ik} = \sum_j a_{ij} b_{jk}$
- $\mathrm{tr}(\mathbf{A}) = \sum_i a_{ii}$

Dessa forma, podemos substituir uma miríade de notações especializadas por expressões tensoriais curtas.

Expressando em Código

Os tensores também podem ser operados com flexibilidade no código. Conforme visto em Section 2.3, podemos criar tensores como mostrado abaixo.

```
# Define tensors
B = torch.tensor([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = torch.tensor([[1, 2], [3, 4]])
v = torch.tensor([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
(torch.Size([2, 2]), torch.Size([2, 2, 3]), torch.Size([2]))
```

O somatório de Einstein foi implementado diretamente. Os índices que ocorrem na soma de Einstein podem ser passados como uma *string*, seguido pelos tensores que estão sofrendo ação. Por exemplo, para implementar a multiplicação de matrizes, podemos considerar o somatório de Einstein visto acima ($\mathbf{A}\mathbf{v} = a_{ij} v_j$) e retirar os próprios índices para obter a implementação:

```
# Reimplement matrix multiplication
torch.einsum("ij, j -> i", A, v), A@v
```

```
(tensor([ 5, 11]), tensor([ 5, 11]))
```

Esta é uma notação altamente flexível. Por exemplo, se quisermos calcular o que seria tradicionalmente escrito como

$$c_{kl} = \sum_{ij} \mathbf{b}_{ijk} \mathbf{a}_{il} v_j. \tag{18.1.34}$$

pode ser implementado via somatório de Einstein como:

```
torch.einsum("ijk, il, j -> kl", B, A, v)
```

```
tensor([[ 90, 126],
        [102, 144],
        [114, 162]])
```

Esta notação é legível e eficiente para humanos, por mais volumoso que seja por algum motivo precisamos gerar uma contração de tensor programaticamente. Por este motivo, `einsum` fornece uma notação alternativa fornecendo índices inteiros para cada tensor. Por exemplo, a mesma contração tensorial também pode ser escrita como:

```
# PyTorch doesn't support this type of notation.
```

Qualquer uma das notações permite uma representação concisa e eficiente das contrações do tensor no código.

18.1.10 Resumo

- Os vetores podem ser interpretados geometricamente como pontos ou direções no espaço.
- Os produtos escalares definem a noção de ângulo para espaços de dimensões arbitrariamente altas.
- Hiperplanos são generalizações dimensionais de linhas e planos. Eles podem ser usados para definir planos de decisão que geralmente são usados como a última etapa em uma tarefa de classificação.
- A multiplicação de matrizes pode ser interpretada geometricamente como distorções uniformes das coordenadas subjacentes. Eles representam uma maneira muito restrita, mas matematicamente limpa, de transformar vetores.
- Dependência linear é uma forma de saber quando uma coleção de vetores está em um espaço dimensional inferior do que esperaríamos (digamos que você tenha 3 vetores vivendo em um espaço 2-dimensional). A classificação de uma matriz é o tamanho do maior subconjunto de suas colunas que são linearmente independentes.
- Quando a inversa de uma matriz é definida, a inversão da matriz nos permite encontrar outra matriz que desfaça a ação da primeira. A inversão da matriz é útil na teoria, mas requer cuidado na prática devido à instabilidade numérica.
- Os determinantes nos permitem medir o quanto uma matriz se expande ou contrai em um espaço. Um determinante diferente de zero implica uma matriz invertível (não singular) e um determinante de valor zero significa que a matriz é não invertível (singular).
- As contrações do tensor e a soma de Einstein fornecem uma notação limpa e organizada para expressar muitos dos cálculos que são vistos no aprendizado de máquina.

18.1.11 Exercícios

1. Qual é o ângulo entre

$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \qquad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix}? \tag{18.1.35}$$

2. Verdadeiro ou falso: $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ e $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ são inversas uma da outra?
3. Suponha que desenhemos uma forma no plano com área $100\mathrm{m}^2$. Qual é a área depois de transformar a figura pela matriz

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{18.1.36}$$

4. Qual dos seguintes conjuntos de vetores são linearmente independentes?

- $\left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix} \right\}$
- $\left\{ \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$
- $\left\{ \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\}$

5. Suponha que você tenha uma matriz escrita como $A = \begin{bmatrix} c \\ d \end{bmatrix} \cdot \begin{bmatrix} a & b \end{bmatrix}$ para alguma escolha de valores a, b, c,e d. Verdadeiro ou falso: o determinante dessa matriz é sempre 0?
6. Os vetores $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ e $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ são ortogonais. Qual é a condição em uma matriz A para que Ae_1 e Ae_2 sejam ortogonais?
7. Como você pode escrever $\mathrm{tr}(\mathbf{A}^4)$ na notação de Einstein para uma matriz arbitrária A?

Discussões[200]

18.2 Autovalores e Autovetores

Os autovalores são frequentemente uma das noções mais úteis encontraremos ao estudar álgebra linear, entretanto, como um iniciante, é fácil ignorar sua importância. Abaixo, apresentamos a decomposção e cálculo destes, e tentamos transmitir algum sentido de por que é tão importante.

Suponha que tenhamos uma matriz A com as seguintes entradas:

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}. \tag{18.2.1}$$

[200] https://discuss.d2l.ai/t/1084

Se aplicarmos A a qualquer vetor $\mathbf{v} = [x, y]^\top$, obtemos um vetor $\mathbf{A}\mathbf{v} = [2x, -y]^\top$. Isso tem uma interpretação intuitiva: estique o vetor para ser duas vezes mais largo na direção x, e, em seguida, inverta-o na direção y-direcionamento.

No entanto, existem *alguns* vetores para os quais algo permanece inalterado. A saber, $[1, 0]^\top$ é enviado para $[2, 0]^\top$ e $[0, 1]^\top$ é enviado para $[0, -1]^\top$. Esses vetores ainda estão na mesma linha, e a única modificação é que a matriz os estica por um fator de 2 e -1 respectivamente. Chamamos esses vetores de *autovetores* e o fator eles são estendidos por *autovalores*.

Em geral, se pudermos encontrar um número λ e um vetor $\mathbf{v}$ tal que

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v}. \tag{18.2.2}$$

Dizemos que $\mathbf{v}$ é um autovetor para A e λ é um autovalor.

18.2.1 Encontrando Autovalores

Vamos descobrir como encontrá-los. Subtraindo $\lambda \mathbf{v}$ de ambos os lados, e então fatorar o vetor, vemos que o acima é equivalente a:

$$(\mathbf{A} - \lambda \mathbf{I})\mathbf{v} = 0. \tag{18.2.3}$$

Para (18.2.3) acontecer, vemos que $(\mathbf{A} - \lambda \mathbf{I})$ deve comprimir alguma direção até zero, portanto, não é invertível e, portanto, o determinante é zero. Assim, podemos encontrar os *valores próprios* descobrindo quanto λ is $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$. Depois de encontrar os valores próprios, podemos resolver $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ para encontrar os *autovetores* associados.

Um Exemplo

Vamos ver isso com uma matriz mais desafiadora

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \tag{18.2.4}$$

Se considerarmos $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$, vemos que isso é equivalente à equação polinomial $0 = (2 - \lambda)(3 - \lambda) - 2 = (4 - \lambda)(1 - \lambda)$. Assim, dois valores próprios são 4 e 1. Para encontrar os vetores associados, precisamos resolver

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \; \text{and} \; \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4x \\ 4y \end{bmatrix}. \tag{18.2.5}$$

Podemos resolver isso com os vetores $[1, -1]^\top$ and $[1, 2]^\top$ respectivamente.

Podemos verificar isso no código usando a rotina incorporada `numpy.linalg.eig`.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.eig(torch.tensor([[2, 1], [2, 3]], dtype=torch.float64),
          eigenvectors=True)
```

```
torch.return_types.eig(
eigenvalues=tensor([[1., 0.],
        [4., 0.]], dtype=torch.float64),
eigenvectors=tensor([[-0.7071, -0.4472],
        [ 0.7071, -0.8944]], dtype=torch.float64))
```

Observe que numpy normaliza os vetores próprios para ter comprimento um, ao passo que consideramos o nosso comprimento arbitrário. Além disso, a escolha do sinal é arbitrária. No entanto, os vetores calculados são paralelos aos que encontramos à mão com os mesmos autovalores.

18.2.2 Matrizes de Decomposição

Vamos continuar o exemplo anterior um passo adiante. Deixe

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \tag{18.2.6}$$

ser a matriz onde as colunas são os autovetores da matriz **A**. Deixe

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \tag{18.2.7}$$

ser a matriz com os autovalores associados na diagonal. Então, a definição de autovalores e autovetores nos diz que

$$\mathbf{A}\mathbf{W} = \mathbf{W}\boldsymbol{\Sigma}. \tag{18.2.8}$$

A matriz W é invertível, então podemos multiplicar ambos os lados por W^{-1} à direita, nós vemos que podemos escrever

$$\mathbf{A} = \mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^{-1}. \tag{18.2.9}$$

Na próxima seção, veremos algumas consequências interessantes disso, mas por agora só precisamos saber que tal decomposição existirá enquanto pudermos encontrar uma coleção completa de autovetores linearmente independentes (de forma que W seja invertível).

18.2.3 Operações em Autovalores e Autovetores

Uma coisa boa sobre autovalores e autovetores (18.2.9) é que podemos escrever muitas operações que geralmente encontramos de forma limpa em termos da decomposição automática. Como primeiro exemplo, considere:

$$\mathbf{A}^n = \overbrace{\mathbf{A}\cdots\mathbf{A}}^{n \text{ times}} = \overbrace{(\mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^{-1})\cdots(\mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^{-1})}^{n \text{ times}} = \mathbf{W}\overbrace{\boldsymbol{\Sigma}\cdots\boldsymbol{\Sigma}}^{n \text{ times}}\mathbf{W}^{-1} = \mathbf{W}\boldsymbol{\Sigma}^n\mathbf{W}^{-1}. \tag{18.2.10}$$

Isso nos diz que para qualquer poder positivo de uma matriz, a autodecomposição é obtida apenas elevando os autovalores à mesma potência. O mesmo pode ser mostrado para potências negativas, então, se quisermos inverter uma matriz, precisamos apenas considerar

$$\mathbf{A}^{-1} = \mathbf{W}\boldsymbol{\Sigma}^{-1}\mathbf{W}^{-1}, \tag{18.2.11}$$

ou em outras palavras, apenas inverta cada autovalor. Isso funcionará, desde que cada autovalor seja diferente de zero, portanto, vemos que invertível é o mesmo que não ter autovalores zero.

De fato, um trabalho adicional pode mostrar que se $\lambda_1, \ldots, \lambda_n$ são os autovalores de uma matriz, então o determinante dessa matriz é

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \tag{18.2.12}$$

ou o produto de todos os autovalores. Isso faz sentido intuitivamente porque qualquer coisa que esticar **W** faz, W^{-1} desfaz, então, no final, o único alongamento que acontece é por multiplicação pela matriz diagonal $\boldsymbol{\Sigma}$, que estica os volumes pelo produto dos elementos diagonais.

Por fim, lembre-se de que a classificação era o número máximo de colunas linearmente independentes de sua matriz. Examinando a decomposição de perto, podemos ver que a classificação é a mesma que o número de autovalores diferentes de zero de **A**.

Os exemplos podem continuar, mas espero que o ponto esteja claro: A autodecomposição pode simplificar muitos cálculos algébricos lineares e é uma operação fundamental subjacente a muitos algoritmos numéricos e muitas das análises que fazemos em álgebra linear.

18.2.4 Composições Originais de Matrizes Simétricas

Nem sempre é possível encontrar autovetores independentes linearmente suficientes para que o processo acima funcione. Por exemplo, a matriz

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \tag{18.2.13}$$

tem apenas um único autovetor, a saber $(1, 0)^\top$. Para lidar com essas matrizes, exigimos técnicas mais avançadas do que podemos cobrir (como a Forma Normal de Jordan ou Decomposição de Valor Singular). Frequentemente precisaremos restringir nossa atenção a essas matrizes onde podemos garantir a existência de um conjunto completo de autovetores.

A família mais comumente encontrada são as *matrizes simétricas*, que são aquelas matrizes onde $\mathbf{A} = \mathbf{A}^\top$.

Neste caso, podemos tomar W como uma *matriz ortogonal* - uma matriz cujas colunas são todas vetores de comprimento unitário que estão em ângulos retos entre si, onde $\mathbf{W}^\top = \mathbf{W}^{-1}$ – e todos os autovalores serão reais. Assim, neste caso especial, podemos escrever (18.2.9) como

$$\mathbf{A} = \mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^\top. \tag{18.2.14}$$

18.2.5 Teorema do Círculo de Gershgorin

Os valores próprios costumam ser difíceis de raciocinar intuitivamente. Se for apresentada uma matriz arbitrária, pouco pode ser dito sobre quais são os valores próprios sem computá-los. Há, no entanto, um teorema que pode facilitar uma boa aproximação se os maiores valores estiverem na diagonal.

Seja $\mathbf{A} = (a_{ij})$ qualquer matriz quadrada($n \times n$). Definiremos $r_i = \sum_{j \neq i} |a_{ij}|$. Deixe $\mathcal{D}_i$ representar o disco no plano complexo com centro a_{ii} radius r_i. Então, cada autovalor de **A** está contido em um dos $\mathcal{D}_i$.

Isso pode ser um pouco difícil de descompactar, então vejamos um exemplo. Considere a matriz:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \tag{18.2.15}$$

Temos $r_1 = 0.3$, $r_2 = 0.6$, $r_3 = 0.8$ e $r_4 = 0.9$. A matriz é simétrica, portanto, todos os autovalores são reais. Isso significa que todos os nossos valores próprios estarão em um dos intervalos de

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \tag{18.2.16}$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \tag{18.2.17}$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \tag{18.2.18}$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \tag{18.2.19}$$

Realizar o cálculo numérico mostra que os valores próprios são aproximadamente 0.99, 2.97, 4.95, 9.08, tudo confortavelmente dentro das faixas fornecidas.

```
A = torch.tensor([[1.0, 0.1, 0.1, 0.1],
              [0.1, 3.0, 0.2, 0.3],
              [0.1, 0.2, 5.0, 0.5],
              [0.1, 0.3, 0.5, 9.0]])

v, _ = torch.eig(A)
v
```

```
tensor([[0.9923, 0.0000],
        [9.0803, 0.0000],
        [4.9539, 0.0000],
        [2.9734, 0.0000]])
```

Desta forma, os autovalores podem ser aproximados, e as aproximações serão bastante precisas no caso em que a diagonal é significativamente maior do que todos os outros elementos.

É uma coisa pequena, mas com um complexo e um tópico sutil como; decomposição automática, é bom obter qualquer compreensão intuitiva possível.

18.2.6 Uma Aplicação Útil: o Crescimento de Mapas Iterados

Agora que entendemos o que são autovetores, em princípio, vamos ver como eles podem ser usados para fornecer um entendimento profundo de um problema central para o comportamento da rede neural: inicialização de peso adequada.

Autovetores como Comportamento de Longo Prazo

A investigação matemática completa da inicialização de redes neurais profundas está além do escopo do texto, mas podemos ver uma versão de brinquedo aqui para entender como os autovalores podem nos ajudar a ver como esses modelos funcionam. Como sabemos, as redes neurais operam por camadas intercaladas de transformações lineares com operações não lineares. Para simplificar aqui, vamos supor que não há não linearidade, e que a transformação é uma única operação de matriz repetida A, para que a saída do nosso modelo seja

$$\mathbf{v}_{out} = \mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A} \mathbf{v}_{in} = \mathbf{A}^N \mathbf{v}_{in}. \tag{18.2.20}$$

Quando esses modelos são inicializados, A é considerado uma matriz aleatória com entradas gaussianas, então vamos fazer uma delas. Para ser concreto, começamos com uma média zero, variância um Gaussiana distribuída 5×5 matriz.

```
torch.manual_seed(42)

k = 5
A = torch.randn(k, k, dtype=torch.float64)
A
```

```
tensor([[ 0.2996,  0.2424,  0.2832, -0.2329,  0.6712],
        [ 0.7818, -1.7903, -1.7484,  0.1735, -0.1182],
        [-1.7446, -0.4695,  0.4573,  0.5177, -0.2771],
        [-0.6641,  0.6551,  0.2616, -1.5265, -0.3311],
        [-0.6378,  0.1072,  0.7096,  0.3009, -0.2869]], dtype=torch.float64)
```

Behavior on Random Data

Para simplificar nosso modelo de brinquedo, vamos assumir que o vetor de dados que alimentamos em $\mathbf{v}_{in}$ é um vetor gaussiano aleatório de cinco dimensões. Vamos pensar sobre o que queremos que aconteça. Para contextualizar, vamos pensar em um problema genérico de ML, onde estamos tentando transformar dados de entrada, como uma imagem, em uma previsão, como a probabilidade de a imagem ser a foto de um gato. Se a aplicação repetida de $\mathbf{A}$ estende um vetor aleatório para ser muito longo, então, pequenas mudanças na entrada serão amplificadas em grandes mudanças na saída — pequenas modificações da imagem de entrada levaria a previsões muito diferentes. Isso não parece certo!

Por outro lado, se $\mathbf{A}$ encolhe vetores aleatórios para serem mais curtos, então, depois de passar por muitas camadas, o vetor irá essencialmente encolher a nada, e a saída não dependerá da entrada. Isso também claramente não está certo!

Precisamos andar na linha estreita entre o crescimento e a decadência para ter certeza de que nossa saída muda dependendo de nossa entrada, mas não muito!

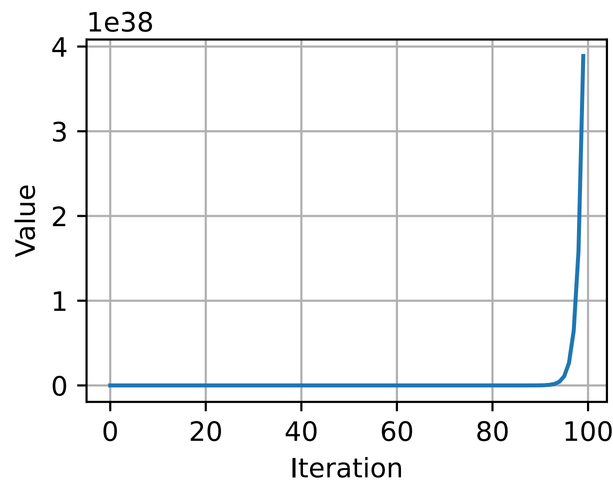
Vamos ver o que acontece quando multiplicamos repetidamente nossa matriz $\mathbf{A}$ contra um vetor de entrada aleatório e acompanhe a norma.

```
# Calculate the sequence of norms after repeatedly applying `A`
v_in = torch.randn(k, 1, dtype=torch.float64)
```

(continues on next page)

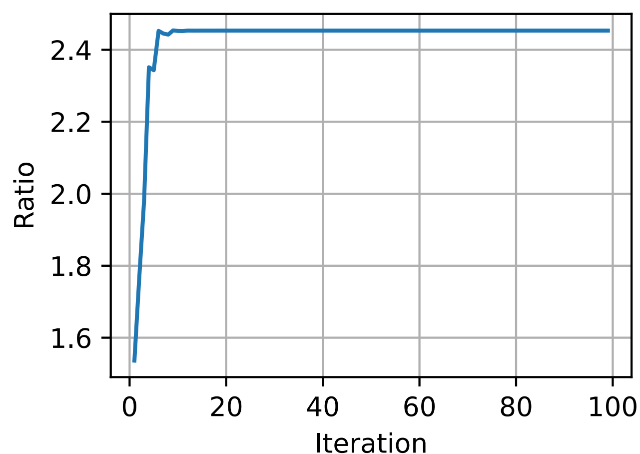
(continued from previous page)

```
norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



The norm is growing uncontrollably! Indeed if we take the list of quotients, we will see a pattern.

```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



Se olharmos para a última parte do cálculo acima, vemos que o vetor aleatório é alongado por um fator de `1.974459321485[...]`, onde a parte no final muda um pouco, mas o fator de alongamento é estável.

Relacionando-se com os Autovetores

Vimos que autovetores e autovalores correspondem na medida em que algo é esticado, mas isso era para vetores específicos e trechos específicos. Vamos dar uma olhada no que eles são para **A**. Uma pequena advertência aqui: acontece que, para ver todos eles, precisaremos ir para os números complexos. Você pode pensar nisso como alongamentos e rotações. Pegando a norma do número complexo (raiz quadrada das somas dos quadrados das partes reais e imaginárias) podemos medir esse fator de alongamento. Deixe-nos também classificá-los.

```
# Compute the eigenvalues
eigs = torch.eig(A)[0][:,0].tolist()
norm_eigs = [torch.abs(torch.tensor(x)) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')
```

```
norms of eigenvalues: [tensor(0.3490), tensor(0.5691), tensor(0.5691), tensor(1.1828),␣
↪tensor(2.4532)]
```

Uma Observação

Vemos algo um pouco inesperado acontecendo aqui: aquele número que identificamos antes para o alongamento de longo prazo de nossa matriz **A** aplicado a um vetor aleatório é *exatamente* (com precisão de treze casas decimais!) o maior autovalor de **A**. Isso claramente não é uma coincidência!

Mas, se agora pensarmos sobre o que está acontecendo geometricamente, isso começa a fazer sentido. Considere um vetor aleatório. Este vetor aleatório aponta um pouco em todas as direções, então, em particular, ele aponta pelo menos um pouco na mesma direção do vetor próprio de **A** associado ao maior autovalor. Isso é tão importante que se chama o *autovalor principal* e o *autovetor principal*. Depois de aplicar **A**, nosso vetor aleatório é esticado em todas as direções possíveis, como está associado a cada autovetor possível, mas é esticado principalmente na direção associado a este autovetor principal. O que isso significa é que depois de aplicar em A, nosso vetor aleatório é mais longo e aponta em uma direção mais perto de estar alinhado com o autovetor principal. Depois de aplicar a matriz várias vezes, o alinhamento com o autovetor principal torna-se cada vez mais próximo até que, para todos os efeitos práticos, nosso vetor aleatório foi transformado no autovetor principal! Na verdade, este algoritmo é a base para o que é conhecido como *iteração de energia* para encontrar o maior autovalor e autovetor de uma matriz. Para obter detalhes, consulte, por exemplo (VanLoan & Golub, 1983).

Corrigindo a Normalização

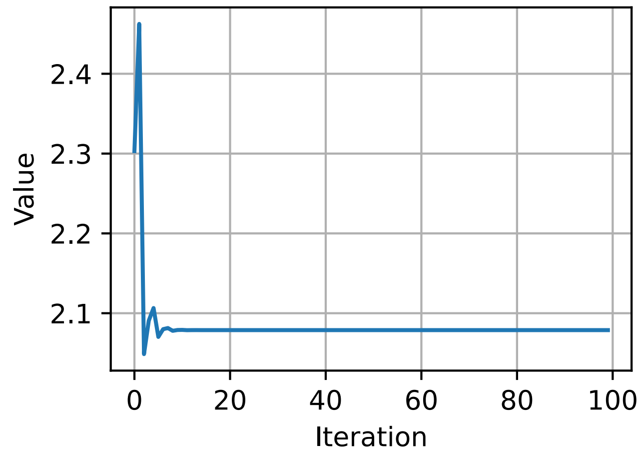
Agora, das discussões acima, concluímos que não queremos que um vetor aleatório seja esticado ou esmagado, gostaríamos que os vetores aleatórios permanecessem do mesmo tamanho durante todo o processo. Para fazer isso, agora redimensionamos nossa matriz por este autovalor principal de modo que o maior autovalor é agora apenas um. Vamos ver o que acontece neste caso.

```
# Rescale the matrix `A`
A /= norm_eigs[-1]
```

(continues on next page)

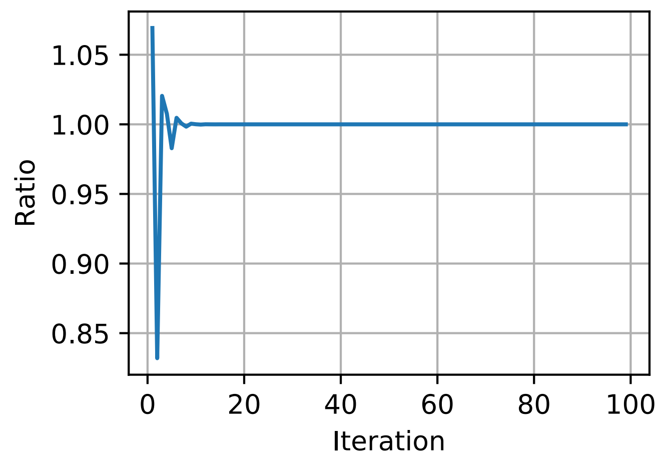
(continued from previous page)

```
# Do the same experiment again
v_in = torch.randn(k, 1, dtype=torch.float64)

norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



Também podemos traçar a proporção entre as normas consecutivas como antes e ver que de fato ela se estabiliza.

```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



18.2.7 Conclusões

Agora vemos exatamente o que esperávamos! Depois de normalizar as matrizes pelo autovalor principal, vemos que os dados aleatórios não explodem como antes, mas, em vez disso, eventualmente se equilibram com um valor específico. Seria bom ser capaz de fazer essas coisas desde os primeiros princípios, e acontece que se olharmos profundamente para a matemática disso, podemos ver que o maior autovalor de uma grande matriz aleatória com média independente de zero, variância de uma entrada gaussiana é em média cerca de $\sqrt{n}$, ou no nosso caso $\sqrt{5} \approx 2.2$, devido a um fato fascinante conhecido como a *lei circular* (Ginibre, 1965). A relação entre os valores próprios (e um objeto relacionado chamado valores singulares) de matrizes aleatórias demonstrou ter conexões profundas para a inicialização adequada de redes neurais, como foi discutido em (Pennington et al., 2017) e trabalhos subsequentes.

18.2.8 Resumo

- Autovetores são vetores que são alongados por uma matriz sem mudar de direção.
- Os autovalores são a quantidade em que os autovetores são alongados pela aplicação da matriz.
- A autodecomposição em autovalores e autovetores de uma matriz pode permitir que muitas operações sejam reduzidas a operações nos autovalores.
- O Teorema do Círculo de Gershgorin pode fornecer valores aproximados para os autovalores de uma matriz.
- O comportamento das potências da matriz iterada depende principalmente do tamanho do maior autovalor. Esse entendimento tem muitas aplicações na teoria de inicialização de redes neurais.

18.2.9 Exercícios

1. Quais são os autovalores e autovetores de

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}? \tag{18.2.21}$$

2. Quais são os autovalores e autovetores da matriz a seguir, e o que há de estranho neste exemplo em comparação com o anterior?

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}. \tag{18.2.22}$$

3. Sem calcular os autovalores, é possível que o menor autovalor da matriz a seguir seja menor que $0,5$? *Nota*: este problema pode ser feito na sua cabeça.

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix}. \tag{18.2.23}$$

Discussões[201]

[201] https://discuss.d2l.ai/t/1086

18.3 Cálculo de Variável Única

Em Section 2.4, vimos os elementos básicos do cálculo diferencial. Esta seção dá um mergulho mais profundo nos fundamentos do cálculo e como podemos entendê-lo e aplicá-lo no contexto do aprendizado de máquina.

18.3.1 Cálculo diferencial

O cálculo diferencial é fundamentalmente o estudo de como as funções se comportam sob pequenas mudanças. Para ver por que isso é tão importante para o*deep learning*, vamos considerar um exemplo.

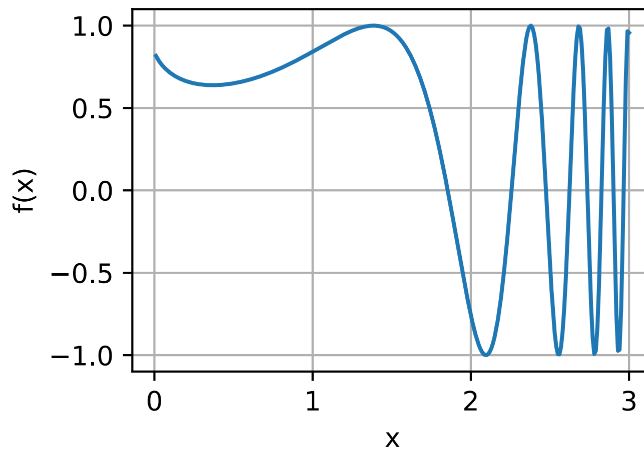
Suponha que temos uma rede neural profunda onde os pesos são, por conveniência, concatenados em um único vetor $\mathbf{w} = (w_1, \ldots, w_n)$. Dado um conjunto de dados de treinamento, consideramos a perda de nossa rede neural neste conjunto de dados, que escreveremos como $\mathcal{L}(\mathbf{w})$.

Esta função é extraordinariamente complexa, codificando o desempenho de todos os modelos possíveis da arquitetura dada neste conjunto de dados, então é quase impossível dizer qual conjunto de pesos $\mathbf{w}$ irá minimizar a perda. Assim, na prática, geralmente começamos inicializando nossos pesos *aleatoriamente* e, em seguida, damos passos pequenos iterativamente na direção que faz com que a perda diminua o mais rápido possível.

A questão então se torna algo que superficialmente não é mais fácil: como encontramos a direção que faz com que os pesos diminuam o mais rápido possível? Para nos aprofundarmos nisso, vamos primeiro examinar o caso com apenas um único peso: $L(\mathbf{w}) = L(x)$ para um único valor real x.

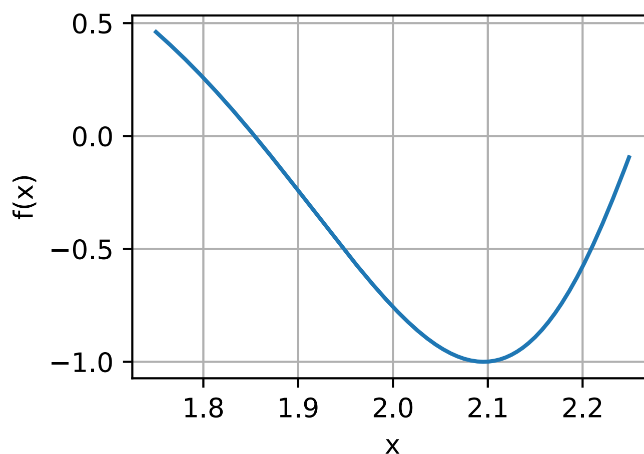
Vamos pegar x e tentar entender o que acontece quando o alteramos por uma pequena quantia para $x+\epsilon$. Se você deseja ser concreto, pense em um número como $\epsilon = 0.0000001$. Para nos ajudar a visualizar o que acontece, vamos representar graficamente uma função de exemplo, $f(x) = \sin(x^x)$, sobre $[0, 3]$.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Define pi in torch

# Plot a function in a normal range
x_big = torch.arange(0.01, 3.01, 0.01)
ys = torch.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```



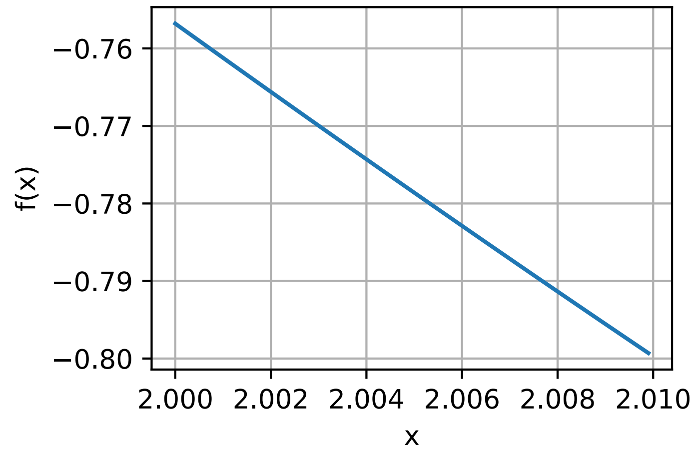
Em grande escala, o comportamento da função não é simples. No entanto, se reduzirmos nosso intervalo para algo menor como $[1.75, 2.25]$, vemos que o gráfico se torna muito mais simples.

```
# Plot a the same function in a tiny range
x_med = torch.arange(1.75, 2.25, 0.001)
ys = torch.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



Levando isso ao extremo, se ampliarmos em um segmento minúsculo, o comportamento se torna muito mais simples: é apenas uma linha reta.

```
# Plot a the same function in a tiny range
x_small = torch.arange(2.0, 2.01, 0.0001)
ys = torch.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```



Esta é a observação chave do cálculo de variável única: o comportamento de funções familiares pode ser modelado por uma linha em um intervalo pequeno o suficiente. Isso significa que, para a maioria das funções, é razoável esperar que, à medida que deslocamos um pouco o valor x da função, a saída $f(x)$ também seja deslocada um pouco. A única pergunta que precisamos responder é: "Qual é o tamanho da mudança na produção em comparação com a mudança na entrada? É a metade? Duas vezes maior?"

Assim, podemos considerar a razão da mudança na saída de uma função para uma pequena mudança na entrada da função. Podemos escrever isso formalmente como

$$\frac{L(x+\epsilon) - L(x)}{(x+\epsilon) - x} = \frac{L(x+\epsilon) - L(x)}{\epsilon}. \tag{18.3.1}$$

Isso já é o suficiente para começar a brincar com o código. Por exemplo, suponha que saibamos que $L(x) = x^2 + 1701(x-4)^3$, então podemos ver o quão grande é esse valor no ponto $x = 4$ como segue.

```
# Define our function
def L(x):
    return x**2 + 1701*(x-4)**3

# Print the difference divided by epsilon for several epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print(f'epsilon = {epsilon:.5f} -> {(L(4+epsilon) - L(4)) / epsilon:.5f}')
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

Agora, se formos observadores, notaremos que a saída desse número é suspeitamente próxima de 8. De fato, se diminuirmos ϵ, veremos que o valor se torna progressivamente mais próximo de 8. Assim, podemos concluir, corretamente, que o valor que buscamos (o grau em que uma mudança na entrada muda a saída) deve ser 8 no ponto $x = 4$. A forma como um matemático codifica este fato é

$$\lim_{\epsilon \rightarrow 0}\frac{L(4+\epsilon) - L(4)}{\epsilon} = 8. \tag{18.3.2}$$

Como uma pequena digressão histórica: nas primeiras décadas de pesquisa de redes neurais, os cientistas usaram este algoritmo (o *método das diferenças finitas*) para avaliar como uma função de perda mudou sob pequenas perturbações: basta alterar os pesos e ver como o perda mudou. Isso é computacionalmente ineficiente, exigindo duas avaliações da função de perda para ver como uma única mudança de uma variável influenciou a perda. Se tentássemos fazer isso mesmo com alguns poucos milhares de parâmetros, seriam necessários vários milhares de avaliações da rede em todo o conjunto de dados! Não foi resolvido até 1986 que o *algoritmo de retropropagação* introduzido em (Rumelhart et al., 1988) forneceu uma maneira de calcular como *qualquer* alteração dos pesos juntos mudaria a perda no mesmo cálculo tempo como uma única previsão da rede no conjunto de dados.

De volta ao nosso exemplo, este valor 8 é diferente para diferentes valores de x, então faz sentido defini-lo como uma função de x. Mais formalmente, esta taxa de variação dependente do valor é referida como a *derivada* que é escrita como

$$\frac{df}{dx}(x) = \lim_{\epsilon \rightarrow 0}\frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{18.3.3}$$

Textos diferentes usarão notações diferentes para a derivada. Por exemplo, todas as notações abaixo indicam a mesma coisa:

$$\frac{df}{dx} = \frac{d}{dx}f = f' = \nabla_xf = D_xf = f_x. \tag{18.3.4}$$

A maioria dos autores escolherá uma única notação e a manterá, porém nem isso é garantido. É melhor estar familiarizado com tudo isso. Usaremos a notação $\frac{df}{dx}$ ao longo deste texto, a menos que queiramos tirar a derivada de uma expressão complexa, caso em que usaremos $\frac{d}{dx}f$ para escrever expressões como

$$\frac{d}{dx}\left[x^4+\cos\left(\frac{x^2+1}{2x-1}\right)\right]. \tag{18.3.5}$$

Muitas vezes, é intuitivamente útil desvendar a definição de derivada (18.3.3) novamente para ver como uma função muda quando fazemos uma pequena mudança de x:

$$\begin{aligned} \frac{df}{dx}(x) = \lim_{\epsilon \rightarrow 0}\frac{f(x+\epsilon) - f(x)}{\epsilon} & \implies \frac{df}{dx}(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \\ & \implies \epsilon \frac{df}{dx}(x) \approx f(x+\epsilon) - f(x) \\ & \implies f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x). \end{aligned} \tag{18.3.6}$$

Vale a pena mencionar explicitamente a última equação. Isso nos diz que se você pegar qualquer função e alterar a entrada em um pequeno valor, a saída mudará nesse pequeno valor escalonado pela derivada.

Desta forma, podemos entender a derivada como o fator de escala que nos diz quão grande é a mudança que obtemos na saída de uma mudança na entrada.

18.3.2 Regras de Cálculo

Agora nos voltamos para a tarefa de entender como calcular a derivada de uma função explícita. Um tratamento formal completo do cálculo derivaria tudo dos primeiros princípios. Não vamos ceder a esta tentação aqui, mas sim fornecer uma compreensão das regras comuns encontradas.

Derivadas Comuns

Como foi visto em Section 2.4, ao calcular derivadas, muitas vezes pode-se usar uma série de regras para reduzir o cálculo a algumas funções básicas. Nós os repetimos aqui para facilitar a referência.

- **Derivada de constantes.** $\frac{d}{dx}c = 0$.
- **Derivada de funções lineares.** $\frac{d}{dx}(ax) = a$.
- **Regra de potência.** $\frac{d}{dx}x^n = nx^{n-1}$.
- **Derivada de exponenciais.** $\frac{d}{dx}e^x = e^x$.
- **Derivada do logaritmo.** $\frac{d}{dx}\log(x) = \frac{1}{x}$.

Regras de Derivadas

Se cada derivada precisasse ser calculada separadamente e armazenada em uma tabela, o cálculo diferencial seria quase impossível. É um presente da matemática que podemos generalizar as derivadas acima e calcular derivadas mais complexas, como encontrar a derivada de $f(x) = \log\left(1 + (x-1)^{10}\right)$. Como foi mencionado em Section 2.4, a chave para fazer isso é codificar o que acontece quando pegamos funções e as combinamos de várias maneiras, o mais importante: somas, produtos e composições.

- **Regra da soma.** $\frac{d}{dx}\left(g(x) + h(x)\right) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$.
- **Regra do produto.** $\frac{d}{dx}\left(g(x) \cdot h(x)\right) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$.
- **Regra da cadeia.** $\frac{d}{dx}g(h(x)) = \frac{dg}{dh}(h(x)) \cdot \frac{dh}{dx}(x)$.

Vamos ver como podemos usar (18.3.6) para entender essas regras. Para a regra da soma, considere a seguinte cadeia de raciocínio:

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon) + h(x+\epsilon) \\ &\approx g(x) + \epsilon\frac{dg}{dx}(x) + h(x) + \epsilon\frac{dh}{dx}(x) \\ &= g(x) + h(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right) \\ &= f(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right). \end{aligned} \tag{18.3.7}$$

Comparando este resultado com o fato de que $f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x)$ vemos que $\frac{df}{dx}(x) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$ conforme desejado. A intuição aqui é: quando mudamos a entrada x, g e h contribuem conjuntamente para a mudança da saída $\frac{dg}{dx}(x)$ e $\frac{dh}{dx}(x)$.

O produto é mais sutil e exigirá uma nova observação sobre como trabalhar com essas expressões. Começaremos como antes usando (18.3.6):

$$
\begin{aligned}
f(x+\epsilon) & = g(x+\epsilon)\cdot h(x+\epsilon) \\
& \approx \left(g(x) + \epsilon \frac{dg}{dx}(x)\right) \cdot \left(h(x) + \epsilon \frac{dh}{dx}(x)\right) \\
& = g(x)\cdot h(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x) \\
& = f(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x).
\end{aligned}
\tag{18.3.8}
$$

Isso se assemelha ao cálculo feito acima, e de fato vemos nossa resposta ($\frac{df}{dx}(x) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$) sentado ao lado de ϵ, mas há a questão desse termo de tamanho ϵ^2. Iremos nos referir a isso como um *termo de ordem superior*, uma vez que a potência de ϵ^2 é maior do que a potência de ϵ^1. Veremos em uma seção posterior que às vezes desejaremos mantê-los sob controle; no entanto, por enquanto, observe que se $\epsilon = 0.0000001$, então $\epsilon^2 = 0.0000000000001$, que é muito menor. Conforme enviamos $\epsilon \rightarrow 0$, podemos ignorar com segurança os termos de pedido superior. Como uma convenção geral neste apêndice, usaremos "$\approx$" para denotar que os dois termos são iguais até os termos de ordem superior. No entanto, se quisermos ser mais formais, podemos examinar o quociente de diferença

$$
\frac{f(x+\epsilon) - f(x)}{\epsilon} = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) + \epsilon\frac{dg}{dx}(x)\frac{dh}{dx}(x),
\tag{18.3.9}
$$

e veja que conforme enviamos $\epsilon \rightarrow 0$, o termo do lado direito vai para zero também.

Finalmente, com a regra da cadeia, podemos novamente progredir como antes usando (18.3.6) e ver que

$$
\begin{aligned}
f(x+\epsilon) & = g(h(x+\epsilon)) \\
& \approx g\left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\
& \approx g(h(x)) + \epsilon\frac{dh}{dx}(x)\frac{dg}{dh}(h(x)) \\
& = f(x) + \epsilon\frac{dg}{dh}(h(x))\frac{dh}{dx}(x),
\end{aligned}
\tag{18.3.10}
$$

onde na segunda linha vemos a função g como tendo sua entrada ($h(x)$) deslocada pela pequena quantidade $\epsilon\frac{dh}{dx}(x)$.

Essa regra nos fornece um conjunto flexível de ferramentas para calcular essencialmente qualquer expressão desejada. Por exemplo,

$$
\begin{aligned}
\frac{d}{dx}\left[\log\left(1+(x-1)^{10}\right)\right] & = \left(1+(x-1)^{10}\right)^{-1}\frac{d}{dx}\left[1+(x-1)^{10}\right] \\
& = \left(1+(x-1)^{10}\right)^{-1}\left(\frac{d}{dx}[1] + \frac{d}{dx}[(x-1)^{10}]\right) \\
& = \left(1+(x-1)^{10}\right)^{-1}\left(0 + 10(x-1)^9\frac{d}{dx}[x-1]\right) \\
& = 10\left(1+(x-1)^{10}\right)^{-1}(x-1)^9 \\
& = \frac{10(x-1)^9}{1+(x-1)^{10}}.
\end{aligned}
\tag{18.3.11}
$$

Onde cada linha usou as seguintes regras:

1. A regra da cadeia e derivada do logaritmo.
2. A regra da soma.
3. A derivada de constantes, regra da cadeia e regra de potência.
4. A regra da soma, derivada de funções lineares, derivada de constantes.

Duas coisas devem ficar claras depois de fazer este exemplo:

1. Qualquer função que possamos escrever usando somas, produtos, constantes, potências, exponenciais e logaritmos pode ter sua derivada calculada mecanicamente seguindo essas regras.
2. Fazer com que um humano siga essas regras pode ser entediante e sujeito a erros!

Felizmente, esses dois fatos juntos apontam para um caminho a seguir: este é um candidato perfeito para a mecanização! Na verdade, a retropropagação, que revisitaremos mais tarde nesta seção, é exatamente isso.

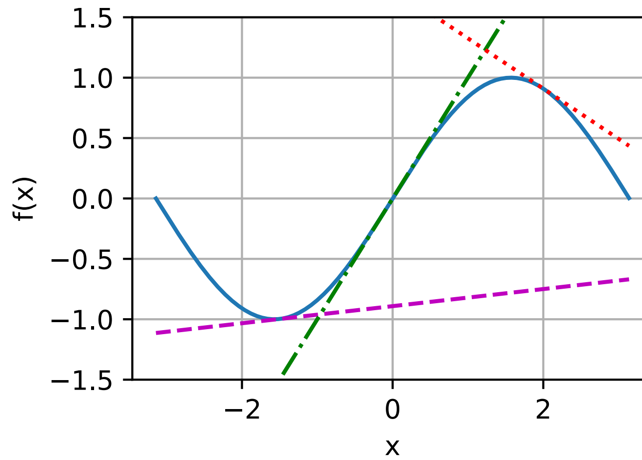
Aproximação Linear

Ao trabalhar com derivadas, geralmente é útil interpretar geometricamente a aproximação usada acima. Em particular, observe que a equação

$$f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x), \tag{18.3.12}$$

aproxima o valor de f por uma linha que passa pelo ponto $(x, f(x))$ e tem inclinação $\frac{df}{dx}(x)$. Desta forma, dizemos que a derivada dá uma aproximação linear para a função f, conforme ilustrado abaixo:

```
# Compute sin
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Compute some linear approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Derivadas de Ordem Superior

Vamos agora fazer algo que pode parecer estranho superficialmente. Pegue uma função f e calcule a derivada $\frac{df}{dx}$. Isso nos dá a taxa de variação de f em qualquer ponto.

No entanto, a derivada, $\frac{df}{dx}$, pode ser vista como uma função em si, então nada nos impede de calcular a derivada de $\frac{df}{dx}$ para obter $\frac{d^2f}{dx^2} = \frac{df}{dx}\left(\frac{df}{dx}\right)$.. Chamaremos isso de segunda derivada de f. Esta função é a taxa de variação da taxa de variação de f, ou em outras palavras, como a taxa de variação está mudando. Podemos aplicar a derivada qualquer número de vezes para obter o que é chamado de n-ésima derivada. Para manter a notação limpa, denotaremos a derivada n-ésima

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{18.3.13}$$

Vamos tentar entender *por que* essa noção é útil. Abaixo, visualizamos $f^{(2)}(x)$, $f^{(1)}(x)$, and $f(x)$.

Primeiro, considere o caso em que a segunda derivada $f^{(2)}(x)$ é uma constante positiva. Isso significa que a inclinação da primeira derivada é positiva. Como resultado, a primeira derivada $f^{(1)}(x)$ pode começar negativa, tornar-se zero em um ponto e então se tornar positiva no final. Isso nos diz a inclinação de nossa função original f e, portanto, a própria função f diminui, nivela e, em seguida, aumenta. Em outras palavras, a função f se curva para cima e tem um único mínimo como é mostrado em Fig. 18.3.1.

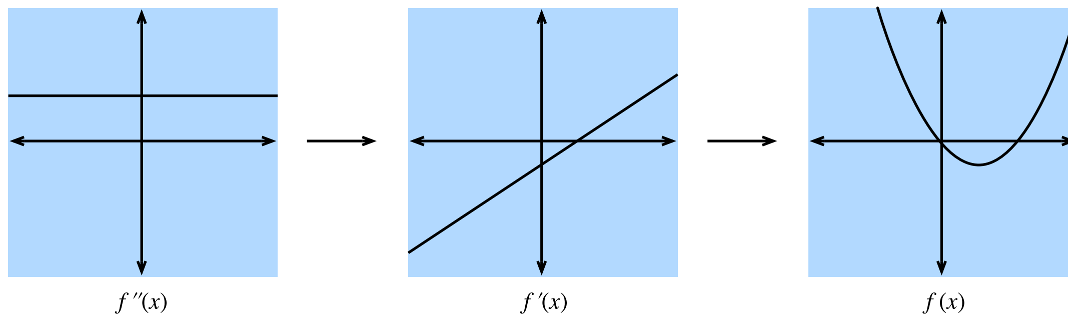


Fig. 18.3.1: Se assumirmos que a segunda derivada é uma constante positiva, então a primeira derivada está aumentando, o que implica que a própria função tem um mínimo.

Em segundo lugar, se a segunda derivada é uma constante negativa, isso significa que a primeira derivada está diminuindo. Isso implica que a primeira derivada pode começar positiva, tornar-se zero em um ponto e, em seguida, tornar-se negativa. Conseqüentemente, a própria função f aumenta, nivela e depois diminui. Em outras palavras, a função f se curva para baixo e tem um único máximo, conforme mostrado em Fig. 18.3.2.

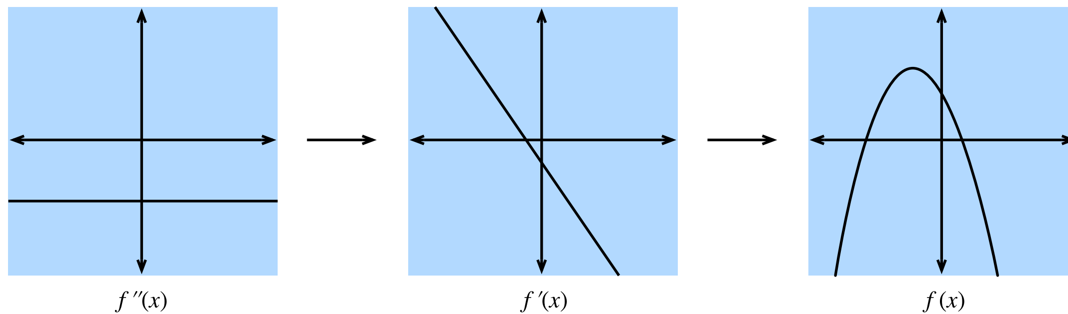


Fig. 18.3.2: Se assumirmos que a segunda derivada é uma constante negativa, então a primeira derivada decrescente, o que implica que a própria função tem um máximo.

Terceiro, se a segunda derivada é sempre zero, então a primeira derivada nunca mudará - ela é constante! Isso significa que f aumenta (ou diminui) a uma taxa fixa, e f é em si uma linha reta como mostrado em Fig. 18.3.3.

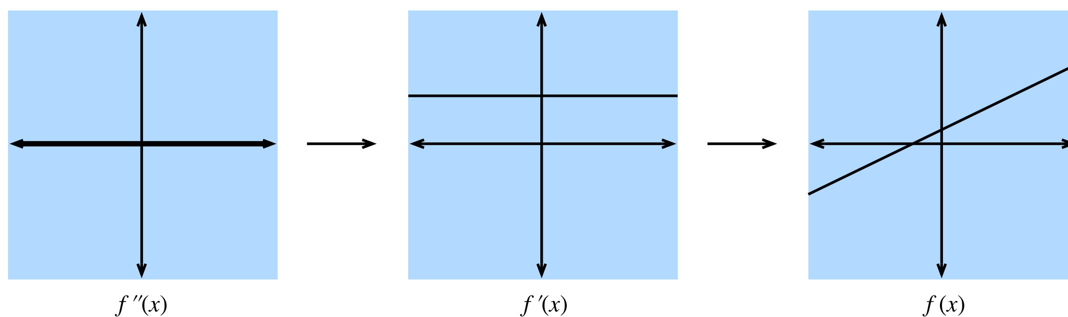


Fig. 18.3.3: Se assumirmos que a segunda derivada é zero, a primeira derivada é constante, o que implica que a própria função é uma linha reta.

Para resumir, a segunda derivada pode ser interpretada como descrevendo a forma como a função f curva. Uma segunda derivada positiva leva a uma curva para cima, enquanto uma segunda derivada negativa significa que f se curva para baixo, e uma segunda derivada zero significa que f não faz nenhuma curva.

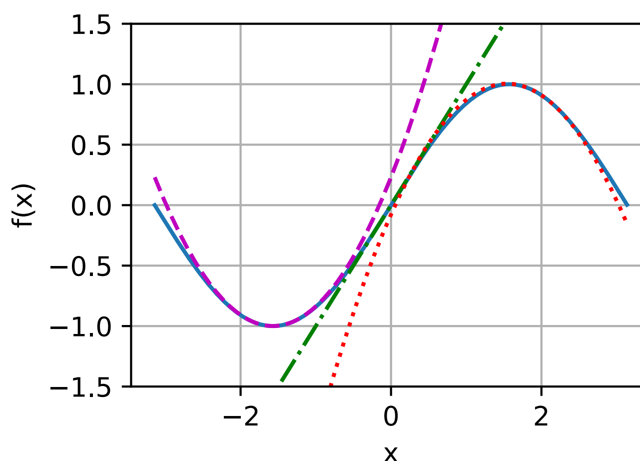
Vamos dar um passo adiante. Considere a função $g(x) = ax^2 + bx + c$. Podemos então calcular que

$$\begin{aligned} \frac{dg}{dx}(x) & = 2ax + b \\ \frac{d^2g}{dx^2}(x) & = 2a. \end{aligned} \tag{18.3.14}$$

Se tivermos alguma função original $f(x)$ em mente, podemos calcular as duas primeiras derivadas e encontrar os valores para a, b, e c que os fazem corresponder a este cálculo. Similarmente à seção anterior, onde vimos que a primeira derivada deu a melhor aproximação com uma linha

reta, esta construção fornece a melhor aproximação por uma quadrática. Vamos visualizar isso para $f(x) = \sin(x)$.

```
# Compute sin
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Compute some quadratic approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)) - (xs - x0)**2 *
                 torch.sin(torch.tensor(x0)) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Vamos estender essa ideia para a ideia de uma *série de Taylor* na próxima seção.

Séries de Taylor

A *série de Taylor* fornece um método para aproximar a função $f(x)$ se recebermos valores para as primeiros n derivadas em um ponto x_0, i.e., $\left\{ f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(n)}(x_0) \right\}$. A ideia será encontrar um polinômio de grau n que corresponda a todas as derivadas fornecidas em x_0.

Vimos o caso de $n = 2$ na seção anterior e um pouco de álgebra mostra que isso é

$$f(x) \approx \frac{1}{2} \frac{d^2 f}{dx^2}(x_0)(x - x_0)^{2} + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{18.3.15}$$

Como podemos ver acima, o denominador de 2 está aí para cancelar os 2 que obtemos quando tomamos duas derivadas de x^2, enquanto os outros termos são todos zero. A mesma lógica se aplica à primeira derivada e ao próprio valor.

Se empurrarmos a lógica ainda mais para $n = 3$, concluiremos que

$$f(x) \approx \frac{\frac{d^3 f}{dx^3}(x_0)}{6}(x - x_0)^3 + \frac{\frac{d^2 f}{dx^2}(x_0)}{2}(x - x_0)^{2} + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{18.3.16}$$

onde $6 = 3 \times 2 = 3!$ vem da constante que obtemos na frente se tomarmos três derivadas de x^3.

Além disso, podemos obter um polinômio de grau n por

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i. \tag{18.3.17}$$

onde a notação

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{18.3.18}$$

De fato, $P_n(x)$ pode ser visto como a melhor aproximação polinomial de n-ésimo grau para nossa função $f(x)$.

Embora não vamos mergulhar totalmente no erro das aproximações acima, vale a pena mencionar o limite infinito. Neste caso, para funções bem comportadas (conhecidas como funções analíticas reais) como $\cos(x)$ or e^x, podemos escrever o número infinito de termos e aproximar exatamente a mesma função

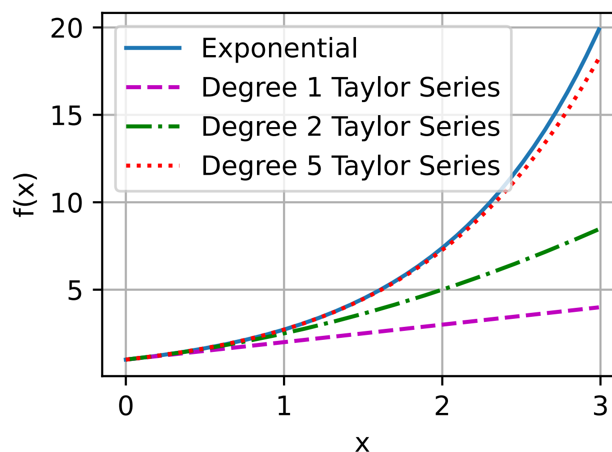
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{18.3.19}$$

Tome $f(x) = e^x$ como um exemplo. Como e^x é sua própria derivada, sabemos que $f^{(n)}(x) = e^x$. Portanto, e^x pode ser reconstruído tomando a série de Taylor em $x_0 = 0$, ou seja,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots. \tag{18.3.20}$$

Vamos ver como isso funciona no código e observar como o aumento do grau da aproximação de Taylor nos aproxima da função desejada e^x.

```
# Compute the exponential function
xs = torch.arange(0, 3, 0.01)
ys = torch.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```



A série Taylor tem duas aplicações principais:

1. *Aplicações teóricas*: Freqüentemente, quando tentamos entender uma função muito complexa, usar a série de Taylor nos permite transformá-la em um polinômio com o qual podemos trabalhar diretamente.
2. *Aplicações numéricas*: Algumas funções como e^x ou $\cos(x)$ são difíceis de serem computadas pelas máquinas. Eles podem armazenar tabelas de valores com uma precisão fixa (e isso geralmente é feito), mas ainda deixa questões em aberto como "Qual é o milésimo dígito de $\cos(1)$?" As séries de Taylor costumam ser úteis para responder a essas perguntas.

18.3.3 Resumo

- As derivadas podem ser usadas para expressar como as funções mudam quando alteramos a entrada em um pequeno valor.
- Derivadas elementares podem ser combinadas usando regras de derivadas para criar derivadas arbitrariamente complexas.
- As derivadas podem ser iteradas para obter derivadas de segunda ordem ou de ordem superior. Cada aumento na ordem fornece informações mais refinadas sobre o comportamento da função.
- Usando informações nas derivadas de um único exemplo de dados, podemos aproximar funções bem comportadas por polinômios obtidos da série de Taylor.

18.3.4 Exercícios

1. Qual é a derivada de $x^3 - 4x + 1$?
2. Qual é a derivada de $\log(\frac{1}{x})$?
3. Verdadeiro ou falso: Se $f'(x) = 0$ então f tem um máximo ou mínimo de x?
4. Onde está o mínimo de $f(x) = x\log(x)$ para $x \geq 0$ (onde assumimos que f assume o valor limite de 0 em $f(0)$)?

Discussões[202]

18.4 Cálculo Multivariável

Agora que temos um entendimento bastante forte das derivadas de uma função de uma única variável, vamos voltar à nossa questão original, onde estávamos considerando uma função de perda de potencialmente bilhões de pesos.

[202] https://discuss.d2l.ai/t/1088

18.4.1 Diferenciação de Dimensões Superiores

O que Section 18.3 nos diz é que se mudarmos um desses bilhões de pesos deixando todos os outros fixos, sabemos o que vai acontecer! Isso nada mais é do que uma função de uma única variável, então podemos escrever

$$L(w_1+\epsilon_1, w_2, \ldots, w_N) \approx L(w_1, w_2, \ldots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \ldots, w_N). \tag{18.4.1}$$

Chamaremos a derivada em uma variável enquanto fixamos a outra *derivada parcial*, e usaremos a notação $\frac{\partial}{\partial w_1}$ para a derivada em (18.4.1).

Agora, vamos pegar isso e mudar $ w_2 $ um pouco para $w_2 + \epsilon_2$:

$$\begin{aligned}
L(w_1+\epsilon_1, w_2+\epsilon_2, \ldots, w_N) & \approx L(w_1, w_2+\epsilon_2, \ldots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2+\epsilon_2, \ldots, w_N+\epsilon_N) \\
& \approx L(w_1, w_2, \ldots, w_N) \\
& \quad + \epsilon_2\frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
& \quad + \epsilon_1\frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
& \quad + \epsilon_1\epsilon_2\frac{\partial}{\partial w_2}\frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
& \approx L(w_1, w_2, \ldots, w_N) \\
& \quad + \epsilon_2\frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
& \quad + \epsilon_1\frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N).
\end{aligned} \tag{18.4.2}$$

Usamos novamente a ideia de que $\epsilon_1\epsilon_2$ é um termo de ordem superior que podemos descartar da mesma forma que descartamos ϵ^2 na seção anterior, junto com o que vimos em (18.4.1). Continuando dessa maneira, podemos escrever que

$$L(w_1+\epsilon_1, w_2+\epsilon_2, \ldots, w_N+\epsilon_N) \approx L(w_1, w_2, \ldots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \ldots, w_N). \tag{18.4.3}$$

Isso pode parecer uma bagunça, mas podemos tornar isso mais familiar observando que a soma à direita parece exatamente com um produto escalar, então, se deixarmos

$$\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\top \text{ and } \nabla_{\mathbf{x}} L = \left[\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N}\right]^\top, \tag{18.4.4}$$

então

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon}\cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{18.4.5}$$

Chamaremos o vetor $\nabla_{\mathbf{w}} L$ de *gradiente* de L.

Equação (18.4.5) vale a pena ponderar por um momento. Tem exatamente o formato que encontramos em uma dimensão, apenas convertemos tudo para vetores e produtos escalares. Isso nos permite dizer aproximadamente como a função L mudará dada qualquer perturbação na entrada. Como veremos na próxima seção, isso nos fornecerá uma ferramenta importante para compreender geometricamente como podemos aprender usando as informações contidas no gradiente.

Mas, primeiro, vejamos essa aproximação em funcionamento com um exemplo. Suponha que estejamos trabalhando com a função

$$f(x, y) = \log(e^x + e^y) \text{ with gradient } \nabla f(x, y) = \left[\frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y}\right]. \tag{18.4.6}$$

Se olharmos para um ponto como $(0, \log(2))$, vemos que

$$f(x, y) = \log(3) \text{ with gradient } \nabla f(x, y) = \left[\frac{1}{3}, \frac{2}{3}\right]. \tag{18.4.7}$$

Assim, se quisermos aproximar f em $(\epsilon_1, \log(2) + \epsilon_2)$, vemos que devemos ter a instância específica de (18.4.5):

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \tag{18.4.8}$$

Podemos testar isso no código para ver o quão boa é a aproximação.

```
%matplotlib inline
import numpy as np
import torch
from IPython import display
from mpl_toolkits import mplot3d
from d2l import torch as d2l


def f(x, y):
    return torch.log(torch.exp(x) + torch.exp(y))
def grad_f(x, y):
    return torch.tensor([torch.exp(x) / (torch.exp(x) + torch.exp(y)),
                         torch.exp(y) / (torch.exp(x) + torch.exp(y))])

epsilon = torch.tensor([0.01, -0.03])
grad_approx = f(torch.tensor([0.]), torch.log(
    torch.tensor([2.]))) + epsilon.dot(
    grad_f(torch.tensor([0.]), torch.log(torch.tensor(2.))))
true_value = f(torch.tensor([0.]) + epsilon[0], torch.log(
    torch.tensor([2.])) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

```
'approximation: tensor([1.0819]), true Value: tensor([1.0821])'
```

18.4.2 Geometria de Gradientes e Gradiente Descendente

Considere novamente (18.4.5):

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{18.4.9}$$

Suponhamos que eu queira usar isso para ajudar a minimizar nossa perda L. Vamos entender geometricamente o algoritmo de gradiente descendente descrito pela primeira vez em Section 2.5. O que faremos é o seguinte:

1. Comece com uma escolha aleatória para os parâmetros iniciais $\mathbf{w}$.

2. Encontre a direção $\mathbf{v}$ que faz L diminuir mais rapidamente em $\mathbf{w}$.
3. Dê um pequeno passo nessa direção: $\mathbf{w} \rightarrow \mathbf{w} + \epsilon\mathbf{v}$.
4. Repita.

A única coisa que não sabemos exatamente como fazer é calcular o vetor $\mathbf{v}$ no segundo passo. Chamaremos tal direção de *direção da descida mais íngreme*. Usando o entendimento geométrico de produtos escalares de Section 18.1, vemos que podemos reescrever (18.4.5) como

$$L(\mathbf{w} + \mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = L(\mathbf{w}) + \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \tag{18.4.10}$$

Observe que seguimos nossa orientação para ter comprimento um por conveniência, e usamos θ para o ângulo entre $\mathbf{v}$ e $\nabla_{\mathbf{w}} L(\mathbf{w})$ Se quisermos encontrar a direção que diminui L o mais rápido possível, queremos tornar essa expressão o mais negativa possível. A única maneira pela qual a direção que escolhemos entra nesta equação é através de $\cos(\theta)$ e, portanto, desejamos tornar esse cosseno o mais negativo possível. Agora, relembrando a forma do cosseno, podemos torná-lo o mais negativo possível, tornando $\cos(\theta) = -1$ ou equivalentemente tornando o ângulo entre o gradiente e nossa direção escolhida em π radianos, ou equivalentemente 180 graus. A única maneira de conseguir isso é seguir na direção oposta exata: escolha $\mathbf{v}$ para apontar na direção oposta exata para $\nabla_{\mathbf{w}} L(\mathbf{w})$!

Isso nos leva a um dos conceitos matemáticos mais importantes no aprendizado de máquina: a direção dos pontos decentes mais íngremes na direção de $-\nabla_{\mathbf{w}} L(\mathbf{w})$. Assim, nosso algoritmo informal pode ser reescrito da seguinte maneira.

1. Comece com uma escolha aleatória para os parâmetros iniciais $\mathbf{w}$.
2. Calcule $\nabla_{\mathbf{w}} L(\mathbf{w})$.
3. Dê um pequeno passo na direção oposta: $\mathbf{w} \rightarrow \mathbf{w} - \epsilon\nabla_{\mathbf{w}} L(\mathbf{w})$.
4. Repita.

Este algoritmo básico foi modificado e adaptado de várias maneiras por muitos pesquisadores, mas o conceito central permanece o mesmo em todos eles. Usar o gradiente para encontrar a direção que diminui a perda o mais rápido possível e atualizar os parâmetros para dar um passo nessa direção.

18.4.3 Uma Nota Sobre Otimização Matemática

Ao longo deste livro, enfocamos diretamente as técnicas de otimização numérica pela razão prática de que todas as funções que encontramos no ambiente de aprendizado profundo são muito complexas para serem minimizadas explicitamente.

No entanto, é um exercício útil considerar o que a compreensão geométrica que obtivemos acima nos diz sobre como otimizar funções diretamente.

Suponha que desejamos encontrar o valor de $\mathbf{x}_0$ que minimiza alguma função $L(\mathbf{x})$. Suponhamos que, além disso, alguém nos dê um valor e nos diga que é o valor que minimiza L. Existe algo que possamos verificar para ver se a resposta deles é plausível?

Considere novamente (18.4.5):

$$L(\mathbf{x}_0 + \boldsymbol{\epsilon}) \approx L(\mathbf{x}_0) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{x}} L(\mathbf{x}_0). \tag{18.4.11}$$

Se o gradiente não for zero, sabemos que podemos dar um passo na direção $-\epsilon\nabla_{\mathbf{x}}L(\mathbf{x}_0)$ para encontrar um valor de L que é menor. Portanto, se realmente estamos no mínimo, não pode ser esse o caso! Podemos concluir que se $\mathbf{x}_0$ é um mínimo, então $\nabla_{\mathbf{x}}L(\mathbf{x}_0) = 0$. Chamamos pontos com $\nabla_{\mathbf{x}}L(\mathbf{x}_0) = 0$ *pontos críticos*.

Isso é bom, porque em algumas configurações raras, nós *podemos* encontrar explicitamente todos os pontos onde o gradiente é zero e encontrar aquele com o menor valor.

Para um exemplo concreto, considere a função

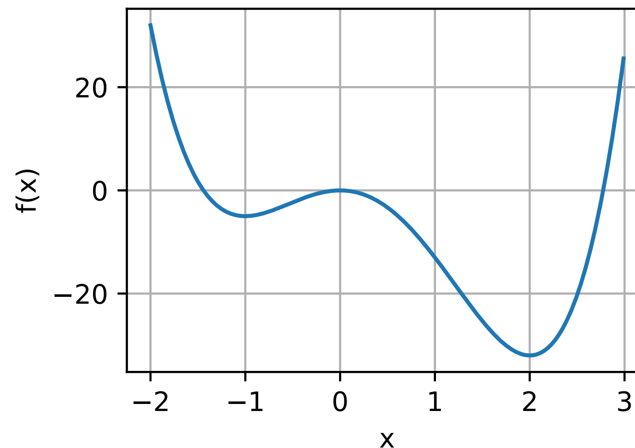
$$f(x) = 3x^4 - 4x^3 - 12x^2. \tag{18.4.12}$$

Esta função tem derivada

$$\frac{df}{dx} = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \tag{18.4.13}$$

A única localização possível dos mínimos está em $x = -1, 0, 2$, onde a função assume os valores $-5, 0, -32$ respectivamente, e assim podemos concluir que minimizamos nossa função quando $x = 2$. Um gráfico rápido confirma isso.

```
x = torch.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```



Isso destaca um fato importante a saber ao trabalhar teoricamente ou numericamente: os únicos pontos possíveis onde podemos minimizar (ou maximizar) uma função terão gradiente igual a zero, no entanto, nem todo ponto com gradiente zero é o verdadeiro *global* mínimo (ou máximo).

18.4.4 Regra da Cadeia Multivariada

Vamos supor que temos uma função de quatro variáveis (w, x, y, e z) que podemos fazer compondo muitos termos:

$$\begin{aligned}f(u, v) & = (u+v)^{2} \\u(a, b) & = (a+b)^{2}, \qquad v(a, b) = (a-b)^{2}, \\a(w, x, y, z) & = (w+x+y+z)^{2}, \qquad b(w, x, y, z) = (w+x-y-z)^2.\end{aligned} \tag{18.4.14}$$

Essas cadeias de equações são comuns ao trabalhar com redes neurais, portanto, tentar entender como calcular gradientes de tais funções é fundamental. Podemos começar a ver dicas visuais dessa conexão em Fig. 18.4.1 se dermos uma olhada em quais variáveis se relacionam diretamente entre si.

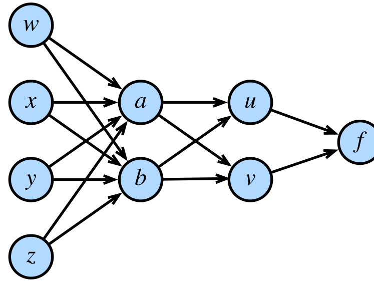


Fig. 18.4.1: As relações de função acima, onde os nós representam valores e as arestas mostram dependência funcional.

Nada nos impede de apenas compor tudo de (18.4.14) e escrever isso

$$f(w, x, y, z) = \left(\left((w+x+y+z)^2+(w+x-y-z)^2\right)^2+\left((w+x+y+z)^2-(w+x-y-z)^2\right)^2\right)^2. \tag{18.4.15}$$

Podemos então tirar a derivada usando apenas derivadas de variável única, mas se fizéssemos isso, rapidamente nos veríamos inundados com termos, muitos dos quais são repetições! Na verdade, pode-se ver que, por exemplo:

$$\begin{aligned}\frac{\partial f}{\partial w} = 2 & \left(2\left(2(w+x+y+z)-2(w+x-y-z)\right)\left((w+x+y+z)^2-(w+x-y-z)^2\right)+\right.\\ & \left. 2\left(2(w+x-y-z)+2(w+x+y+z)\right)\left((w+x-y-z)^2+(w+x+y+z)^2\right)\right)\times \\ & \left(\left((w+x+y+z)^2-(w+x-y-z)^2\right)^2+\left((w+x-y-z)^2+(w+x+y+z)^2\right)^2\right).\end{aligned} \tag{18.4.16}$$

Se também quiséssemos calcular $\frac{\partial f}{\partial x}$, acabaríamos com uma equação semelhante novamente com muitos termos repetidos e muitos termos repetidos *compartilhados* entre as duas derivadas. Isso representa uma enorme quantidade de trabalho desperdiçado e, se precisássemos calcular as derivadas dessa forma, toda a revolução do aprendizado profundo teria estagnado antes de começar!

Vamos resolver o problema. Começaremos tentando entender como f muda quando mudamos a, essencialmente supondo que $w, x, y,$ e z não existem. Vamos raciocinar como fazíamos quando trabalhamos com gradiente pela primeira vez. Vamos pegar a e adicionar uma pequena quantia ϵ a ele.

$$\begin{aligned} & f(u(a+\epsilon, b), v(a+\epsilon, b)) \\ \approx & f\left(u(a, b) + \epsilon\frac{\partial u}{\partial a}(a, b), v(a, b) + \epsilon\frac{\partial v}{\partial a}(a, b)\right) \\ \approx & f(u(a, b), v(a, b)) + \epsilon\left[\frac{\partial f}{\partial u}(u(a, b), v(a, b))\frac{\partial u}{\partial a}(a, b) + \frac{\partial f}{\partial v}(u(a, b), v(a, b))\frac{\partial v}{\partial a}(a, b)\right]. \end{aligned} \tag{18.4.17}$$

A primeira linha segue da definição de derivada parcial e a segunda segue da definição de gradiente. É notacionalmente pesado rastrear exatamente onde avaliamos cada derivada, como na

expressão $\frac{\partial f}{\partial u}(u(a, b), v(a, b))$, então frequentemente abreviamos para muito mais memorável

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}. \tag{18.4.18}$$

É útil pensar sobre o significado do processo. Estamos tentando entender como uma função da forma $f(u(a, b), v(a, b))$ muda seu valor com uma mudança em a. Isso pode ocorrer de duas maneiras: há o caminho onde $a \to u \to f$ e onde $a \to v \to f$. Podemos calcular ambas as contribuições por meio da regra da cadeia: $\frac{\partial w}{\partial u} \cdot \frac{\partial u}{\partial x}$ e $\frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x}$ respectivamente, e somados.

Imagine que temos uma rede diferente de funções onde as funções à direita dependem daquelas que estão conectadas à esquerda, como mostrado em Fig. 18.4.2.

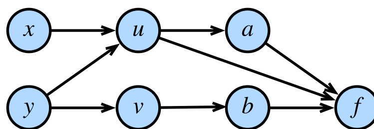


Fig. 18.4.2: Outro exemplo mais sutil da regra da cadeia.

Para calcular algo como $\frac{\partial f}{\partial y}$, precisamos somar todos (neste caso 3) caminhos de y a f dando

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}\frac{\partial v}{\partial y}. \tag{18.4.19}$$

Entender a regra da cadeia desta forma renderá grandes dividendos ao tentar entender como os gradientes fluem através das redes, e por que várias escolhas arquitetônicas como aquelas em LSTMs (Section 9.2) ou camadas residuais (Section 7.6) podem ajudam a moldar o processo de aprendizagem, controlando o fluxo gradiente.

18.4.5 O Algoritmo de Retropropagação

Vamos retornar ao exemplo de (18.4.14) a seção anterior onde

$$\begin{split}\begin{aligned}
f(u, v) & = (u+v)^{2} \\
u(a, b) & = (a+b)^{2}, \qquad v(a, b) = (a-b)^{2}, \\
a(w, x, y, z) & = (w+x+y+z)^{2}, \qquad b(w, x, y, z) = (w+x-y-z)^2.
\end{aligned}\end{split} \tag{18.4.20}$$

Se quisermos calcular, digamos $\frac{\partial f}{\partial w}$, podemos aplicar a regra da cadeia multivariada para ver:

$$\begin{split}\begin{aligned}
\frac{\partial f}{\partial w} & = \frac{\partial f}{\partial u}\frac{\partial u}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial w}, \\
\frac{\partial u}{\partial w} & = \frac{\partial u}{\partial a}\frac{\partial a}{\partial w}+\frac{\partial u}{\partial b}\frac{\partial b}{\partial w}, \\
\frac{\partial v}{\partial w} & = \frac{\partial v}{\partial a}\frac{\partial a}{\partial w}+\frac{\partial v}{\partial b}\frac{\partial b}{\partial w}.
\end{aligned}\end{split} \tag{18.4.21}$$

Vamos tentar usar esta decomposição para calcular $\frac{\partial f}{\partial w}$. Observe que tudo o que precisamos aqui são as várias parciais de etapa única:

$$
\begin{aligned}
\frac{\partial f}{\partial u} = 2(u+v), & \quad \frac{\partial f}{\partial v} = 2(u+v), \\
\frac{\partial u}{\partial a} = 2(a+b), & \quad \frac{\partial u}{\partial b} = 2(a+b), \\
\frac{\partial v}{\partial a} = 2(a-b), & \quad \frac{\partial v}{\partial b} = -2(a-b), \\
\frac{\partial a}{\partial w} = 2(w+x+y+z), & \quad \frac{\partial b}{\partial w} = 2(w+x-y-z).
\end{aligned}
\tag{18.4.22}
$$

Se escrevermos isso no código, isso se tornará uma expressão bastante gerenciável.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'    f at {w}, {x}, {y}, {z} is {f}')

# Compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Compute the final result from inputs to outputs
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
```

No entanto, observe que isso ainda não facilita o cálculo de algo como $\frac{\partial f}{\partial x}$. A razão para isso é a *forma* que escolhemos para aplicar a regra da cadeia. Se observarmos o que fizemos acima, sempre mantivemos ∂w no denominador quando podíamos. Desta forma, optamos por aplicar a regra da cadeia vendo como w mudou todas as outras variáveis. Se é isso que queríamos, seria uma boa ideia. No entanto, pense em nossa motivação com o aprendizado profundo: queremos ver como cada parâmetro altera a *perda*. Em essência, queremos aplicar a regra da cadeia mantendo ∂f no numerador sempre que pudermos!

Para ser mais explícito, observe que podemos escrever

$$
\begin{aligned}
\frac{\partial f}{\partial w} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w}, \\
\frac{\partial f}{\partial a} & = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}, \\
\frac{\partial f}{\partial b} & = \frac{\partial f}{\partial u}\frac{\partial u}{\partial b} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}.
\end{aligned}
\tag{18.4.23}
$$

Observe que esta aplicação da regra da cadeia nos faz computar explicitamente

$\frac{\partial f}{\partial u}, \frac{\partial f}{\partial v}, \frac{\partial f}{\partial a}, \frac{\partial f}{\partial b},$ e $\frac{\partial f}{\partial w}$. Nada nos impede de incluir também as equações:

$$\begin{aligned}
\frac{\partial f}{\partial x} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x}+\frac{\partial f}{\partial b}\frac{\partial b}{\partial x}, \\
\frac{\partial f}{\partial y} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y}+\frac{\partial f}{\partial b}\frac{\partial b}{\partial y}, \\
\frac{\partial f}{\partial z} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial z}+\frac{\partial f}{\partial b}\frac{\partial b}{\partial z}.
\end{aligned} \tag{18.4.24}$$

e acompanhar como f muda quando mudamos *qualquer* nó em toda a rede. Vamos implementar.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'f at {w}, {x}, {y}, {z} is {f}')

# Compute the derivative using the decomposition above
# First compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)

# Now compute how f changes when we change any value from output to input
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz

print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
print(f'df/dx at {w}, {x}, {y}, {z} is {df_dx}')
print(f'df/dy at {w}, {x}, {y}, {z} is {df_dy}')
print(f'df/dz at {w}, {x}, {y}, {z} is {df_dz}')
```

```
f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
df/dx at -1, 0, -2, 1 is -4096
df/dy at -1, 0, -2, 1 is -4096
df/dz at -1, 0, -2, 1 is -4096
```

O fato de calcularmos derivadas de f de volta para as entradas em vez de das entradas para as saídas (como fizemos no primeiro trecho de código acima) é o que dá a esse algoritmo seu nome: *retropropagação*. Observe que existem duas etapas: 1. Calcular o valor da função e as parciais de etapa única da frente para trás. Embora não feito acima, isso pode ser combinado em um único *passe para frente*. 2. Calcule o gradiente de f de trás para frente. Chamamos isso de *passe para trás*.

Isso é precisamente o que todo algoritmo de aprendizado profundo implementa para permitir o cálculo do gradiente da perda em relação a cada peso na rede em uma passagem. É um fato surpreendente que tenhamos tal decomposição.

Para ver como encapsular isso, vamos dar uma olhada rápida neste exemplo.

```
# Initialize as ndarrays, then attach gradients
w = torch.tensor([-1.], requires_grad=True)
x = torch.tensor([0.], requires_grad=True)
y = torch.tensor([-2.], requires_grad=True)
z = torch.tensor([1.], requires_grad=True)
# Do the computation like usual, tracking gradients
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2

# Execute backward pass
f.backward()

print(f'df/dw at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {w.grad.data.item()}')
print(f'df/dx at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {x.grad.data.item()}')
print(f'df/dy at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {y.grad.data.item()}')
print(f'df/dz at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {z.grad.data.item()}')
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

Tudo o que fizemos acima pode ser feito automaticamente chamando `f.backwards ()`.

18.4.6 Hessians

Como no cálculo de variável única, é útil considerar derivadas de ordem superior para entender como podemos obter uma melhor aproximação de uma função do que usar apenas o gradiente.

Há um problema imediato que se encontra ao trabalhar com derivadas de funções de várias variáveis de ordem superior, que é o grande número delas. Se temos uma função $f(x_1, \ldots, x_n)$ de n variáveis, então podemos tomar n^2 derivadas secundas, nomeadamente para qualquer escolha de i e j:

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i}\left(\frac{d}{dx_j}f\right). \tag{18.4.25}$$

Isso é tradicionalmente montado em uma matriz chamada *Hessian*:

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \end{bmatrix}. \tag{18.4.26}$$

Nem todas as entradas desta matriz são independentes. Na verdade, podemos mostrar que, enquanto ambas * parciais mistas * (derivadas parciais em relação a mais de uma variável) existem e são contínuos, podemos dizer que para qualquer i e j,

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \tag{18.4.27}$$

Isto segue considerando primeiro perturbar uma função na direção de x_i, e então perturbá-la em x_j e então comparar o resultado disso com o que acontece se perturbarmos primeiro x_j e então x_i, com o conhecimento que ambos os pedidos levam à mesma mudança final na produção de f.

Assim como acontece com variáveis únicas, podemos usar essas derivadas para ter uma ideia muito melhor de como a função se comporta perto de um ponto. Em particular, podemos usar isso para encontrar a quadrática de melhor ajuste próximo a um ponto $\mathbf{x}_0$, como vimos em uma única variável.

Vejamos um exemplo. Suponha que $f(x_1, x_2) = a + b_1x_1 + b_2x_2 + c_{11}x_1^2 + c_{12}x_1x_2 + c_{22}x_2^2$. Esta é a forma geral de uma quadrática em duas variáveis. Se olharmos para o valor da função, seu gradiente e seu Hessian (18.4.26), tudo no ponto zero:

$$\begin{aligned} f(0,0) & = a, \\ \nabla f(0,0) & = \begin{bmatrix}b_1 \\ b_2\end{bmatrix}, \\ \mathbf{H} f(0,0) & = \begin{bmatrix}2 c_{11} & c_{12} \\ c_{12} & 2c_{22}\end{bmatrix}, \end{aligned} \tag{18.4.28}$$

podemos obter nosso polinômio original de volta, dizendo

$$f(\mathbf{x}) = f(0) + \nabla f (0) \cdot \mathbf{x} + \frac{1}{2}\mathbf{x}^\top \mathbf{H} f(0) \mathbf{x}. \tag{18.4.29}$$

In general, if we computed this expansion any point $\mathbf{x}_0$, we see that

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f (\mathbf{x}_0) \cdot (\mathbf{x}-\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^\top \mathbf{H} f(\mathbf{x}_0) (\mathbf{x}-\mathbf{x}_0). \tag{18.4.30}$$

Isso funciona para entradas de qualquer dimensão e fornece a melhor aproximação quadrática para qualquer função em um ponto. Para dar um exemplo, vamos representar graficamente a função

$$f(x, y) = xe^{-x^2-y^2}. \tag{18.4.31}$$

Pode-se calcular que o gradiente e Hessian são

$$\nabla f(x, y) = e^{-x^2-y^2}\begin{pmatrix}1-2x^2 \\ -2xy\end{pmatrix} \; \text{and} \; \mathbf{H}f(x, y) = e^{-x^2-y^2}\begin{pmatrix} 4x^3 - 6x & 4x^2y - 2y \\ 4x^2y-2y &4xy^2-2x\end{pmatrix}. \tag{18.4.32}$$

E assim, com um pouco de álgebra, veja que a aproximação quadrática em $[-1,0]^\top$ é

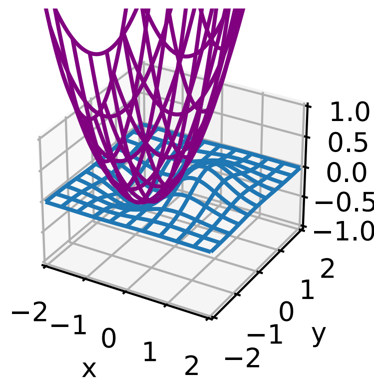
$$f(x, y) \approx e^{-1}\left(-1 - (x+1) +(x+1)^2+y^2\right). \tag{18.4.33}$$

```
# Construct grid and compute function
x, y = torch.meshgrid(torch.linspace(-2, 2, 101),
                   torch.linspace(-2, 2, 101))

z = x*torch.exp(- x**2 - y**2)

# Compute approximating quadratic with gradient and Hessian at (1, 0)
w = torch.exp(torch.tensor([-1.]))*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

# Plot function
```

(continues on next page)

(continued from previous page)

```
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.numpy(), y.numpy(), z.numpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x.numpy(), y.numpy(), w.numpy(),
                  **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```



Isso forma a base para o Algoritmo de Newton discutido em Section 11.3, onde realizamos a otimização numérica encontrando iterativamente a quadrática de melhor ajuste e, em seguida, minimizando exatamente essa quadrática.

18.4.7 Um Pouco de Cálculo Matricial

Derivadas de funções envolvendo matrizes revelaram-se particularmente interessantes. Esta seção pode se tornar notacionalmente pesada, portanto, pode ser ignorada em uma primeira leitura, mas é útil saber como as derivadas de funções que envolvem operações de matriz comum são muitas vezes muito mais limpas do que se poderia prever inicialmente, especialmente considerando como as operações de matriz centrais são para o aprendizado profundo aplicações.

Vamos começar com um exemplo. Suponha que temos algum vetor de coluna fixo $\boldsymbol{\beta}$, e queremos obter a função de produto $f(\mathbf{x}) = \boldsymbol{\beta}^\top\mathbf{x}$, e entender como o produto escalar muda quando mudamos $\mathbf{x}$.

Um pouco de notação que será útil ao trabalhar com derivadas de matriz em ML é chamado de *derivada de matriz de layout de denominador*, onde montamos nossas derivadas parciais na forma de qualquer vetor, matriz ou tensor que esteja no denominador do diferencial. Neste caso, vamos escrever

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}, \tag{18.4.34}$$

onde combinamos a forma do vetor coluna $\mathbf{x}$.

Se escrevermos nossa função em componentes, isso é

$$f(\mathbf{x}) = \sum_{i=1}^{n} \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \tag{18.4.35}$$

Se agora tomarmos a derivada parcial em relação a β_1, note que tudo é zero, exceto o primeiro termo, que é apenas x_1 multiplicado por β_1,, então obtemos isso

$$\frac{df}{dx_1} = \beta_1, \tag{18.4.36}$$

ou mais geralmente isso

$$\frac{df}{dx_i} = \beta_i. \tag{18.4.37}$$

Agora podemos remontar isso em uma matriz para ver

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}. \tag{18.4.38}$$

Isso ilustra alguns fatores sobre o cálculo de matriz que muitas vezes iremos contrariar ao longo desta seção:

- Primeiro, os cálculos ficarão bastante complicados.
- Em segundo lugar, os resultados finais são muito mais limpos do que o processo intermediário e sempre serão semelhantes ao caso de uma única variável. Neste caso, observe que $\frac{d}{dx}(bx) = b$ and $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}^\top \mathbf{x}) = \boldsymbol{\beta}$ são ambos semelhantes.
- Terceiro, as transpostas muitas vezes podem aparecer aparentemente do nada. A principal razão para isso é a convenção de que combinamos a forma do denominador, portanto, quando multiplicamos as matrizes, precisaremos fazer transposições para corresponder à forma do termo original.

Para continuar construindo a intuição, vamos tentar um cálculo um pouco mais difícil. Suponha que temos um vetor coluna $\mathbf{x}$ e uma matriz quadrada A e queremos calcular

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A \mathbf{x}). \tag{18.4.39}$$

Para direcionar para uma notação mais fácil de manipular, vamos considerar este problema usando a notação de Einstein. Neste caso, podemos escrever a função como

$$\mathbf{x}^\top A \mathbf{x} = x_i a_{ij} x_j. \tag{18.4.40}$$

Para calcular nossa derivada, precisamos entender para cada k, qual é o valor de

$$\frac{d}{dx_k}(\mathbf{x}^\top A \mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \tag{18.4.41}$$

Pela regra do produto, isso é

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \tag{18.4.42}$$

Para um termo como $\frac{dx_i}{dx_k}$, não é difícil ver que este é um quando $i = k$ e zero caso contrário. Isso significa que todos os termos em que i e k são diferentes desaparecem dessa soma, de modo que os únicos termos que permanecem nessa primeira soma são aqueles em que $i = k$. O mesmo raciocínio vale para o segundo termo em que precisamos de $j = k$. Isto dá

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \tag{18.4.43}$$

Agora, os nomes dos índices na notação de Einstein são arbitrários — o fato de que i e j são diferentes é irrelevante para este cálculo neste ponto, então podemos reindexar para que ambos usem i para ver isso

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \tag{18.4.44}$$

Agora, é aqui que precisamos de um pouco de prática para ir mais longe. Vamos tentar identificar esse resultado em termos de operações de matriz. $a_{ki} + a_{ik}$ é o k, i-ésimo componente de $\mathbf{A} + \mathbf{A}^\top$. Isto dá

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \tag{18.4.45}$$

Da mesma forma, este termo é agora o produto da matriz $\mathbf{A} + \mathbf{A}^\top$ pelo vetor $\mathbf{x}$, então vemos que

$$\left[\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x})\right]_k = \frac{d}{dx_k} x_i a_{ij} x_j = [(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}]_k. \tag{18.4.46}$$

Assim, vemos que a k -ésima entrada da derivada desejada de (18.4.39) é apenas a k-ésima entrada do vetor à direita e, portanto, os dois são iguais. Assim

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{18.4.47}$$

Isso exigiu muito mais trabalho do que o anterior, mas o resultado final é pequeno. Mais do que isso, considere o seguinte cálculo para derivadas de variável única tradicionais:

$$\frac{d}{dx}(xax) = \frac{dx}{dx} ax + xa\frac{dx}{dx} = (a + a)x. \tag{18.4.48}$$

Equivalentemente $\frac{d}{dx}(ax^2) = 2ax = (a + a)x$. Novamente, obtemos um resultado que se parece bastante com o resultado de uma única variável, mas com uma transposição inserida.

Neste ponto, o padrão deve parecer bastante suspeito, então vamos tentar descobrir o porquê. Quando tomamos derivadas de matriz como esta, vamos primeiro supor que a expressão que obtemos será outra expressão de matriz: uma expressão que podemos escrevê-la em termos de produtos e somas de matrizes e suas transposições. Se tal expressão existir, ela precisará ser verdadeira para todas as matrizes. Em particular, ele precisará ser verdadeiro para matrizes 1×1, caso em que o produto da matriz é apenas o produto dos números, a soma da matriz é apenas a soma e a transposta não faz nada! Em outras palavras, qualquer expressão que obtivermos *deve* corresponder à expressão de variável única. Isso significa que, com alguma prática, muitas vezes pode-se adivinhar as derivadas de matriz apenas por saber como a expressão da única variável associada deve se parecer!

Vamos tentar fazer isso. Suponha que $\mathbf{X}$ é uma matriz $n \times m$, $\mathbf{U}$ é uma $n \times r$ e $\mathbf{V}$ é uma $r \times m$. Vamos tentar calcular

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = \,? \tag{18.4.49}$$

Este cálculo é importante em uma área chamada fatoração de matrizes. Para nós, no entanto, é apenas uma derivada para calcular. Vamos tentar imaginar o que seria para as matrizes 1×1. Nesse caso, obtemos a expressão

$$\frac{d}{dv}(x - uv)^2 = -2(x - uv)u, \tag{18.4.50}$$

onde, a derivada é bastante padrão. Se tentarmos converter isso de volta em uma expressão de matriz, obteremos

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2(\mathbf{X} - \mathbf{U}\mathbf{V})\mathbf{U}. \tag{18.4.51}$$

No entanto, se olharmos para isso, não funciona bem. Lembre-se de que **X** é $n \times m$, assim como **UV**, então a matriz $2(\mathbf{X} - \mathbf{U}\mathbf{V})$ é $n \times m$. Por outro lado **U** é $n \times r$, e não podemos multiplicar uma matriz $n \times m$ e uma $n \times r$ porque as dimensões não combinam!

Queremos obter $\frac{d}{d\mathbf{V}}$, que tem a mesma forma de **V**, que é $r \times m$. Então, de alguma forma, precisamos pegar uma matriz $n \times m$ e uma matriz $n \times r$, multiplicá-las juntas (talvez com algumas transposições) para obter uma $r \times m$. Podemos fazer isso multiplicando $U^\top$ por $(\mathbf{X} - \mathbf{U}\mathbf{V})$.. Assim, podemos adivinhar a solução para (18.4.49) é

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V}). \tag{18.4.52}$$

Para mostrar que isso funciona, seríamos negligentes em não fornecer um cálculo detalhado. Se já acreditamos que essa regra prática funciona, fique à vontade para pular esta derivação. Para calcular

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2, \tag{18.4.53}$$

devemos encontrar para cada a e b

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = \frac{d}{dv_{ab}}\sum_{i,j}\left(x_{ij} - \sum_k u_{ik}v_{kj}\right)^2. \tag{18.4.54}$$

Lembrando que todas as entradas de **X** e **U** são constantes no que diz respeito a $\frac{d}{dv_{ab}}$, podemos colocar a derivada dentro da soma, e aplicar a regra da cadeia ao quadrado para obter

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = \sum_{i,j}2\left(x_{ij} - \sum_k u_{ik}v_{kj}\right)\left(-\sum_k u_{ik}\frac{dv_{kj}}{dv_{ab}}\right). \tag{18.4.55}$$

Como na derivação anterior, podemos notar que $\frac{dv_{kj}}{dv_{ab}}$ só é diferente de zero se $k = a$ and $j = b$. Se qualquer uma dessas condições não for válida, o termo na soma é zero e podemos descartá-lo livremente. Nós vemos que

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i\left(x_{ib} - \sum_k u_{ik}v_{kb}\right)u_{ia}. \tag{18.4.56}$$

Uma sutileza importante aqui é que o requisito de que $k = a$ não ocorre dentro da soma interna, uma vez que k é uma variável que estamos somando dentro do termo interno. Para um exemplo notacionalmente mais limpo, considere por que

$$\frac{d}{dx_1}\left(\sum_i x_i\right)^2 = 2\left(\sum_i x_i\right). \tag{18.4.57}$$

A partir deste ponto, podemos começar a identificar os componentes da soma. Primeiro,

$$\sum_k u_{ik} v_{kb} = [\mathbf{U}\mathbf{V}]_{ib}. \tag{18.4.58}$$

Portanto, toda a expressão no interior da soma é

$$x_{ib} - \sum_k u_{ik} v_{kb} = [\mathbf{X} - \mathbf{U}\mathbf{V}]_{ib}. \tag{18.4.59}$$

Isso significa que agora podemos escrever nossa derivada como

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i [\mathbf{X} - \mathbf{U}\mathbf{V}]_{ib} u_{ia}. \tag{18.4.60}$$

Queremos que se pareça com o elemento a, b de uma matriz, para que possamos usar a técnica como no exemplo anterior para chegar a uma expressão de matriz, o que significa que precisamos trocar a ordem dos índices em u_{ia}. Se notarmos que $u_{ia} = [\mathbf{U}^\top]_{ai}$, podemos escrever

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i [\mathbf{U}^\top]_{ai} [\mathbf{X} - \mathbf{U}\mathbf{V}]_{ib}. \tag{18.4.61}$$

Este é um produto de matriz e, portanto, podemos concluir que

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2[\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V})]_{ab}. \tag{18.4.62}$$

e assim podemos escrever a solução para (18.4.49)

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V}). \tag{18.4.63}$$

Isso corresponde à solução que adivinhamos acima!

É razoável perguntar neste ponto: "Por que não posso simplesmente escrever versões de matriz de todas as regras de cálculo que aprendi? Está claro que isso ainda é mecânico. Por que não simplesmente acabamos com isso!" E de fato existem tais regras e (Petersen et al., 2008) fornece um excelente resumo. No entanto, devido à infinidade de maneiras pelas quais as operações de matriz podem ser combinadas em comparação com valores únicos, há muito mais regras de derivadas de matriz do que regras de variável única. Geralmente, é melhor trabalhar com os índices ou deixar para a diferenciação automática, quando apropriado.

18.4.8 Resumo

- Em dimensões superiores, podemos definir gradientes que têm o mesmo propósito que os derivadas em uma dimensão. Isso nos permite ver como uma função multivariável muda quando fazemos uma pequena mudança arbitrária nas entradas.
- O algoritmo de retropropagação pode ser visto como um método de organizar a regra da cadeia multivariável para permitir o cálculo eficiente de muitas derivadas parciais.
- O cálculo matricial nos permite escrever as derivadas das expressões matriciais de maneiras concisas.

18.4.9 Exercícios

1. Dado um vetor de coluna $\boldsymbol{\beta}$, calcule as derivadas de $f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{x}$ e $g(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\beta}$. Por que você obtém a mesma resposta?
2. Seja $\mathbf{v}$ um vetor de dimensão n. O que é $\frac{\partial}{\partial \mathbf{v}} \|\mathbf{v}\|_2$?
3. Seja $L(x, y) = \log(e^x + e^y)$. Calcule o gradiente. Qual é a soma dos componentes do gradiente?
4. Seja $f(x, y) = x^2 y + x y^2$. Mostre que o único ponto crítico é $(0,0)$. Considerando $ f(x, x)$, determine se $(0,0)$ é máximo, mínimo ou nenhum.
5. Suponha que estejamos minimizando uma função $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$. Como podemos interpretar geometricamente a condição de $\nabla f = 0$ em termos de g e h?

Discussões[203]

18.5 Cálculo Integral

A diferenciação representa apenas metade do conteúdo de uma educação tradicional de cálculo. O outro pilar, integração, começa parecendo uma pergunta um tanto desconexa: "Qual é a área abaixo desta curva?" Embora aparentemente não relacionada, a integração está intimamente ligada à diferenciação por meio do que é conhecido como *teorema fundamental do cálculo*.

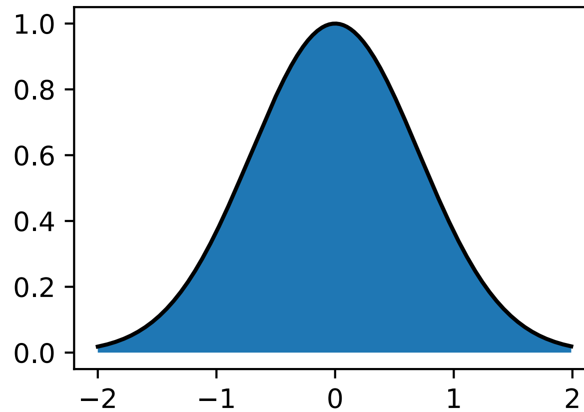
No nível de *machine learning* que discutimos neste livro, não precisaremos ter um conhecimento profundo de integração. No entanto, forneceremos uma breve introdução para estabelecer as bases para quaisquer outras aplicações que encontraremos mais tarde.

18.5.1 Interpretação Geométrica

Suponha que tenhamos uma função $f(x)$. Para simplificar, vamos supor que $f(x)$ não seja negativa (nunca assume um valor menor que zero). O que queremos tentar entender é: qual é a área contida entre $f(x)$ e o eixo x?

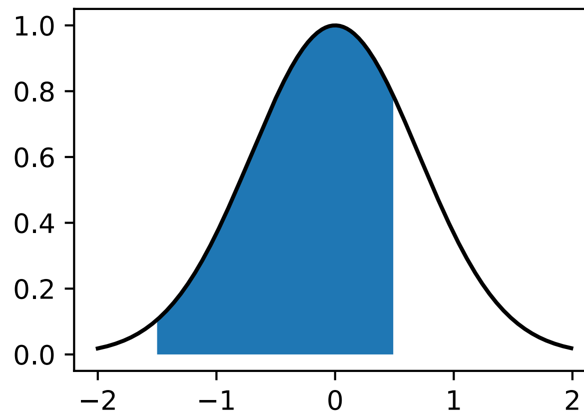
```
%matplotlib inline
import torch
from IPython import display
from mpl_toolkits import mplot3d
from d2l import torch as d2l

x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```

[203] https://discuss.d2l.ai/t/1090



Na maioria dos casos, esta área será infinita ou indefinida (considere a área sob $f(x) = x^2$), então as pessoas frequentemente falarão sobre a área entre um par de pontas, digamos a e b.

```
x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```



Iremos denotar esta área pelo símbolo integral abaixo:

$$\textrm{Area}(\mathcal{A}) = \int_a^b f(x) \;dx. \tag{18.5.1}$$

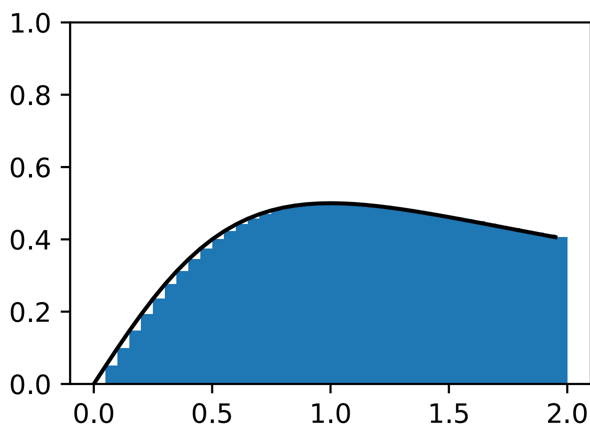
A variável interna é uma variável fictícia, muito parecida com o índice de uma soma em $\sum$ e, portanto, pode ser escrita de forma equivalente com qualquer valor interno que desejarmos:

$$\int_a^b f(x) \;dx = \int_a^b f(z) \;dz. \tag{18.5.2}$$

Há uma maneira tradicional de tentar entender como podemos tentar aproximar essas integrais: podemos imaginar pegar a região entre a e b e dividi-la em fatias verticais de N. Se N for grande, podemos aproximar a área de cada fatia por um retângulo e, em seguida, somar as áreas para

obter a área total sob a curva. Vamos dar uma olhada em um exemplo fazendo isso no código. Veremos como obter o valor verdadeiro em uma seção posterior.

```
epsilon = 0.05
a = 0
b = 2

x = torch.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = torch.sum(epsilon*f)
true = torch.log(torch.tensor([5.])) / 2

d2l.set_figsize()
d2l.plt.bar(x, f, width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

f'approximation: {approx}, truth: {true}'
```




```
'approximation: 0.7944855690002441, truth: tensor([0.8047])'
```

O problema é que, embora possa ser feito numericamente, podemos fazer essa abordagem analiticamente apenas para as funções mais simples, como

$$\int_a^b x \; dx. \tag{18.5.3}$$

Qualquer coisa um pouco mais complexa como nosso exemplo do código acima

$$\int_a^b \frac{x}{1+x^2} \; dx. \tag{18.5.4}$$

está além do que podemos resolver com um método tão direto.

Em vez disso, faremos uma abordagem diferente. Trabalharemos intuitivamente com a noção da área, e aprenderemos a principal ferramenta computacional usada para encontrar integrais: o *teorema fundamental do cálculo*. Esta será a base para nosso estudo de integração.

18.5.2 O Teorema Fundamental do Cálculo

Para mergulhar mais fundo na teoria da integração, vamos apresentar uma função

$$F(x) = \int_0^x f(y) dy. \tag{18.5.5}$$

Esta função mede a área entre 0 e x dependendo de como alteramos x. Observe que isso é tudo de que precisamos, já que

$$\int_a^b f(x) \; dx = F(b) - F(a). \tag{18.5.6}$$

Esta é uma codificação matemática do fato de que podemos medir a área até o ponto final distante e então subtrair a área até o ponto final próximo, conforme indicado em Fig. 18.5.1.

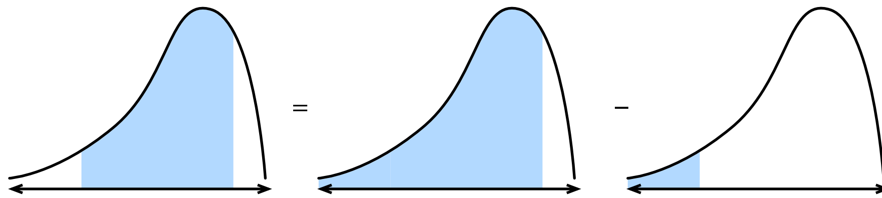


Fig. 18.5.1: Visualizando porque podemos reduzir o problema de calcular a área sob uma curva entre dois pontos para calcular a área à esquerda de um ponto.

Assim, podemos descobrir qual é a integral em qualquer intervalo, descobrindo o que é $F(x)$.

Para fazer isso, consideremos um experimento. Como costumamos fazer em cálculo, vamos imaginar o que acontece quando mudamos o valor um pouquinho. Pelo comentário acima, sabemos que

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y) \; dy. \tag{18.5.7}$$

Isso nos diz que a função muda de acordo com a área sob uma pequena porção de uma função.

Este é o ponto em que fazemos uma aproximação. Se olharmos para uma pequena porção de área como esta, parece que esta área está próxima da área retangular com a altura o valor de $f(x)$ e a largura da base ϵ De fato, pode-se mostrar que à medida que $\epsilon \rightarrow 0$ essa aproximação se torna cada vez melhor. Assim podemos concluir:

$$F(x+\epsilon) - F(x) \approx \epsilon f(x). \tag{18.5.8}$$

Porém, agora podemos notar: este é exatamente o padrão que esperamos se estivéssemos calculando a derivada de F! Assim, vemos o seguinte fato bastante surpreendente:

$$\frac{dF}{dx}(x) = f(x). \tag{18.5.9}$$

Este é o *teorema fundamental do cálculo*. Podemos escrever em forma expandida como

$$\frac{d}{dx}\int_{-\infty}^x f(y) \; dy = f(x). \tag{18.5.10}$$

Ele pega o conceito de localização de áreas (*a priori* bastante difícil) e o reduz a derivadas de uma instrução (algo muito mais completamente compreendido). Um último comentário que devemos

fazer é que isso não nos diz exatamente o que $F(x)$ é. Na verdade, $F(x) + C$ para qualquer C tem a mesma derivada. Este é um fato da vida na teoria da integração. Felizmente, observe que, ao trabalhar com integrais definidas, as constantes desaparecem e, portanto, são irrelevantes para o resultado.

$$\int_a^b f(x)\, dx = (F(b) + C) - (F(a) + C) = F(b) - F(a). \tag{18.5.11}$$

Isso pode parecer sem sentido abstrato, mas vamos parar um momento para apreciar que isso nos deu uma perspectiva totalmente nova sobre as integrais computacionais. Nosso objetivo não é mais fazer algum tipo de processo de corte e soma para tentar recuperar a área, ao invés disso, precisamos apenas encontrar uma função cuja derivada é a função que temos! Isso é incrível, pois agora podemos listar muitas integrais bastante difíceis apenas revertendo a tabela de Section 18.3.2. Por exemplo, sabemos que a derivada de x^n is nx^{n-1}. Assim, podemos dizer usando o teorema fundamental (18.5.10) que

$$\int_0^x ny^{n-1}\, dy = x^n - 0^n = x^n. \tag{18.5.12}$$

Da mesma forma, sabemos que a derivada de e^x é ela mesma, o que significa

$$\int_0^x e^x\, dx = e^x - e^0 = e^x - 1. \tag{18.5.13}$$

Desta forma, podemos desenvolver toda a teoria da integração aproveitando as idéias do cálculo diferencial livremente. Toda regra de integração deriva desse único fato.

18.5.3 Mudança de Variável

Assim como com a diferenciação, há várias regras que tornam o cálculo de integrais mais tratáveis. Na verdade, cada regra de cálculo diferencial (como a regra do produto, regra da soma e regra da cadeia) tem uma regra correspondente para o cálculo integral (integração por partes, linearidade de integração e fórmula de mudança de variáveis, respectivamente). Nesta seção, vamos mergulhar no que é indiscutivelmente o mais importante da lista: a fórmula de mudança de variáveis.

Primeiro, suponha que temos uma função que é ela mesma uma integral:

$$F(x) = \int_0^x f(y)\, dy. \tag{18.5.14}$$

Vamos supor que queremos saber como essa função se parece quando a compomos com outra para obter $F(u(x))$. Pela regra da cadeia, sabemos

$$\frac{d}{dx}F(u(x)) = \frac{dF}{du}(u(x)) \cdot \frac{du}{dx}. \tag{18.5.15}$$

Podemos transformar isso em uma declaração sobre integração usando o teorema fundamental (18.5.10) como acima. Isto dá

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy}\, dy. \tag{18.5.16}$$

Lembrando que F é em si uma integral dá que o lado esquerdo pode ser reescrito para ser

$$\int_{u(0)}^{u(x)} f(y)\, dy = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy}\, dy. \tag{18.5.17}$$

Da mesma forma, lembrar que F é uma integral nos permite reconhecer que $\frac{dF}{dx} = f$ usando o teorema fundamental (18.5.10), e assim podemos concluir

$$\int_{u(0)}^{u(x)} f(y)\; dy = \int_0^x f(u(y)) \cdot \frac{du}{dy}\; dy. \tag{18.5.18}$$

Esta é a fórmula de *mudança de variáveis*.

Para uma derivação mais intuitiva, considere o que acontece quando tomamos uma integral de $f(u(x))$ entre x e $x+\epsilon$. Para um pequeno ϵ, esta integral é aproximadamente $\epsilon f(u(x))$, a área do retângulo associado. Agora, vamos comparar isso com a integral de $f(y)$ de $u(x)$ a $u(x+\epsilon)$. Sabemos que $u(x+\epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$, então a área deste retângulo é aproximadamente $\epsilon \frac{du}{dx}(x) f(u(x))$. Assim, para fazer a área desses dois retângulos serem iguais, precisamos multiplicar o primeiro por $\frac{du}{dx}(x)$ como está ilustrado em Fig. 18.5.2.

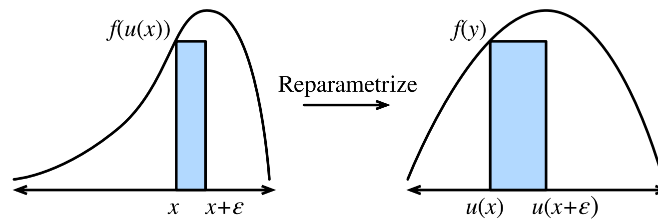


Fig. 18.5.2: Visualizando a transformação de um único retângulo fino sob a mudança de variáveis.

Isso nos diz que

$$\int_x^{x+\epsilon} f(u(y))\frac{du}{dy}(y)\; dy = \int_{u(x)}^{u(x+\epsilon)} f(y)\; dy. \tag{18.5.19}$$

Esta é a fórmula de mudança de variáveis expressa para um único retângulo pequeno.

Se $u(x)$ e $f(x)$ forem escolhidos corretamente, isso pode permitir o cálculo de integrais incrivelmente complexas. Por exemplo, se escolhermos $f(y) = 1$ e $u(x) = e^{-x^2}$ (o que significa $\frac{du}{dx}(x) = -2xe^{-x^2}$), isso pode mostrar, por exemplo, que

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1\; dy = -2\int_0^1 ye^{-y^2}\; dy, \tag{18.5.20}$$

e, assim, reorganizando isso

$$\int_0^1 ye^{-y^2}\; dy = \frac{1-e^{-1}}{2}. \tag{18.5.21}$$

18.5.4 Um Comentário Sobre as Convenções de Sinais

Os leitores mais atentos irão observar algo estranho nos cálculos acima. Ou seja, cálculos como

$$\int_{e^{-0}}^{e^{-1}} 1\; dy = e^{-1} - 1 < 0, \tag{18.5.22}$$

pode produzir números negativos. Ao pensar sobre áreas, pode ser estranho ver um valor negativo, por isso vale a pena aprofundar no que é a convenção.

Os matemáticos assumem a noção de áreas sinalizadas. Isso se manifesta de duas maneiras. Primeiro, se considerarmos uma função $f(x)$ que às vezes é menor que zero, então a área também será negativa. Então por exemplo

$$\int_0^1 (-1)\;dx = -1. \tag{18.5.23}$$

Da mesma forma, integrais que progridem da direita para a esquerda, ao invés da esquerda para a direita também são consideradas áreas negativas

$$\int_0^{-1} 1\;dx = -1. \tag{18.5.24}$$

A área padrão (da esquerda para a direita de uma função positiva) é sempre positiva. Qualquer coisa obtida ao invertê-lo (digamos, inverter o eixo x para obter a integral de um número negativo ou inverter o eixo y para obter uma integral na ordem errada) produzirá uma área negativa. E, de fato, girar duas vezes dará um par de sinais negativos que se cancelam para ter área positiva

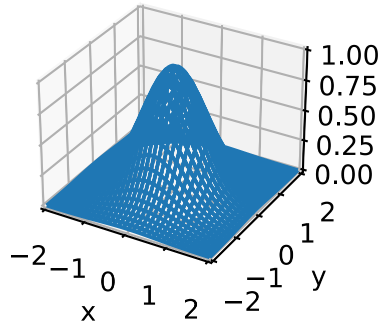
$$\int_0^{-1} (-1)\;dx = 1. \tag{18.5.25}$$

Se essa discussão parece familiar, é! Em Section 18.1 discutimos como o determinante representava a área sinalizada da mesma maneira.

18.5.5 Integrais Múltiplas

Em alguns casos, precisaremos trabalhar em dimensões superiores. Por exemplo, suponha que temos uma função de duas variáveis, como $f(x, y)$ e queremos saber o volume sob f quando x varia entre $[a, b]$ e y varia acima de $[c, d]$.

```
# Construct grid and compute function
x, y = torch.meshgrid(torch.linspace(-2, 2, 101), torch.linspace(-2, 2, 101))
z = torch.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```



Nós escrevemos isso como

$$\int_{[a,b]\times[c,d]} f(x,y)\;dx\;dy. \tag{18.5.26}$$

Suponha que desejamos calcular essa integral. Minha alegação é que podemos fazer isso calculando iterativamente primeiro a integral em x e, em seguida, mudando para a integral em y, ou seja,

$$\int_{[a,b]\times[c,d]} f(x,y)\;dx\;dy = \int_c^d \left(\int_a^b f(x,y)\;dx\right)\;dy. \tag{18.5.27}$$

Vamos ver por que isso acontece.

Considere a figura acima, onde dividimos a função em $\epsilon \times \epsilon$ quadrados que indexaremos com coordenadas inteiras i, j. Neste caso, nossa integral é aproximadamente

$$\sum_{i,j} \epsilon^{2} f(\epsilon i, \epsilon j). \tag{18.5.28}$$

Depois de discretizar o problema, podemos somar os valores nesses quadrados na ordem que quisermos e não nos preocupar em alterar os valores. Isso é ilustrado em Fig. 18.5.3. Em particular, podemos dizer que

$$\sum_{j} \epsilon \left(\sum_{i} \epsilon f(\epsilon i, \epsilon j)\right). \tag{18.5.29}$$

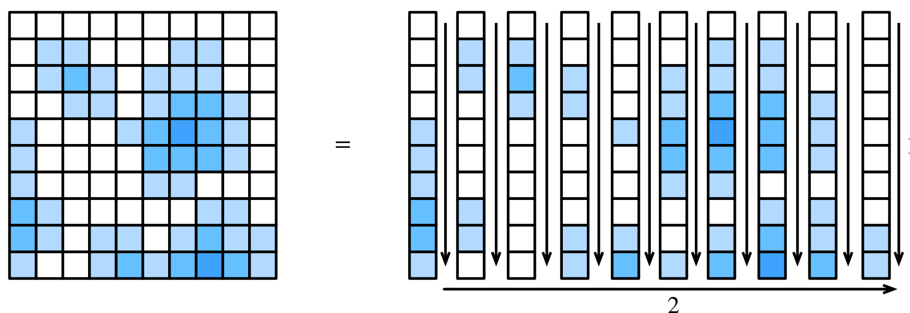


Fig. 18.5.3: Ilustrando como decompor uma soma em muitos quadrados como uma soma nas primeiras colunas (1), depois somando as somas das colunas (2).

A soma do interior é precisamente a discretização da integral

$$G(\epsilon j) = \int_a^b f(x, \epsilon j) \; dx. \tag{18.5.30}$$

Finalmente, observe que se combinarmos essas duas expressões, obteremos

$$\sum_j \epsilon G(\epsilon j) \approx \int_c^d G(y) \; dy = \int_{[a,b]\times[c,d]} f(x, y) \; dx \; dy. \tag{18.5.31}$$

Assim, juntando tudo, temos que

$$\int_{[a,b]\times[c,d]} f(x, y) \; dx \; dy = \int_c^d \left(\int_a^b f(x, y) \; dx \right) \; dy. \tag{18.5.32}$$

Observe que, uma vez discretizado, tudo o que fizemos foi reorganizar a ordem em que adicionamos uma lista de números. Isso pode fazer parecer que não é nada, mas esse resultado (chamado *Teorema de Fubini*) nem sempre é verdadeiro! Para o tipo de matemática encontrada ao fazer o aprendizado de máquina (funções contínuas), não há preocupação, no entanto, é possível criar exemplos onde ela falha (por exemplo, a função $f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3$ sobre o retângulo $[0, 2] \times [0, 1]$).

Observe que a escolha de fazer a integral em x primeiro e, em seguida, a integral em y foi arbitrária. Poderíamos ter igualmente escolhido fazer y primeiro e, em seguida, x para ver

$$\int_{[a,b]\times[c,d]} f(x, y) \; dx \; dy = \int_a^b \left(\int_c^d f(x, y) \; dy \right) \; dx. \tag{18.5.33}$$

Muitas vezes, vamos condensar em notação vetorial e dizer que para $U = [a, b] \times [c, d]$, isso é

$$\int_U f(\mathbf{x}) \; d\mathbf{x}. \tag{18.5.34}$$

18.5.6 Mudança de Variáveis em Integrais Múltiplas

Tal como acontece com variáveis únicas em (18.5.18), a capacidade de alterar variáveis dentro de uma integral de dimensão superior é uma ferramenta chave. Vamos resumir o resultado sem derivação.

Precisamos de uma função que reparameterize nosso domínio de integração. Podemos considerar isso como $\phi : \mathbb{R}^n \to \mathbb{R}^n$, ou seja, qualquer função que recebe n variáveis reais e retorna outro n. Para manter as expressões limpas, assumiremos que ϕ é *injetotora*, o que quer dizer que nunca se dobra ($\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$).

Neste caso, podemos dizer que

$$\int_{\phi(U)} f(\mathbf{x}) \; d\mathbf{x} = \int_U f(\phi(\mathbf{x})) \left|\det(D\phi(\mathbf{x}))\right| \; d\mathbf{x}. \tag{18.5.35}$$

onde $D\phi$ é o *Jacobiano* de ϕ, que é a matriz das derivadas parciais de $\boldsymbol{\phi} = (\phi_1(x_1, \ldots, x_n), \ldots, \phi_n(x_1, \ldots, x_n))$,

$$D\boldsymbol{\phi} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \tag{18.5.36}$$

Olhando de perto, vemos que isso é semelhante à regra de cadeia de única variável (18.5.18), exceto que substituímos o termo $\frac{du}{dx}(x)$ por $|\det(D\phi(\mathbf{x}))|$. Vamos ver como podemos interpretar este termo. Lembre-se de que o termo $\frac{du}{dx}(x)$ existia para dizer o quanto esticamos nosso eixo x aplicando u. O mesmo processo em dimensões superiores é determinar quanto esticamos a área (ou volume, ou hipervolume) de um pequeno quadrado (ou pequeno *hipercubo*) aplicando $\boldsymbol{\phi}$. Se $\boldsymbol{\phi}$ era a multiplicação por uma matriz, então sabemos como o determinante já dá a resposta.

Com algum trabalho, pode-se mostrar que o *Jacobiano* fornece a melhor aproximação para uma função multivariável $\boldsymbol{\phi}$ em um ponto por uma matriz da mesma forma que poderíamos aproximar por retas ou planos com derivados e gradientes. Assim, o determinante do Jacobiano reflete exatamente o fator de escala que identificamos em uma dimensão.

É preciso algum trabalho para preencher os detalhes, então não se preocupe se eles não estiverem claros agora. Vejamos pelo menos um exemplo que usaremos mais tarde. Considere a integral

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} \; dx \; dy. \tag{18.5.37}$$

Brincar com essa integral diretamente não nos levará a lugar nenhum, mas se mudarmos as variáveis, podemos fazer um progresso significativo. $\boldsymbol{\phi}(r, \theta) = (r\cos(\theta), r\sin(\theta))$ (o que significa que $x = r\cos(\theta)$, $y = r\sin(\theta)$), então podemos aplicar a fórmula de mudança da variável para ver que isso é a mesma coisa que

$$\int_0^{\infty} \int_0^{2\pi} e^{-r^2} \, |\det(D\phi(\mathbf{x}))| \; d\theta \; dr, \tag{18.5.38}$$

onde

$$|\det(D\phi(\mathbf{x}))| = \left|\det \begin{bmatrix} \cos(\theta) & -r\sin(\theta) \\ \sin(\theta) & r\cos(\theta) \end{bmatrix}\right| = r(\cos^2(\theta) + \sin^2(\theta)) = r. \tag{18.5.39}$$

Assim, a integral é

$$\int_0^{\infty} \int_0^{2\pi} re^{-r^2} \; d\theta \; dr = 2\pi \int_0^{\infty} re^{-r^2} \; dr = \pi, \tag{18.5.40}$$

onde a igualdade final segue pelo mesmo cálculo que usamos na seção Section 18.5.3.

Encontraremos essa integral novamente quando estudarmos variáveis aleatórias contínuas em Section 18.6.

18.5.7 Resumo

- A teoria da integração nos permite responder a perguntas sobre áreas ou volumes.
- O teorema fundamental do cálculo nos permite alavancar o conhecimento sobre derivadas para calcular áreas através da observação de que a derivada da área até certo ponto é dada pelo valor da função que está sendo integrada.
- Integrais em dimensões superiores podem ser calculados iterando integrais de variável única.

18.5.8 Exercícios

1. Quanto é $\int_1^2 \frac{1}{x} \; dx$?
2. Use a fórmula de mudança de variáveis para integrar $\int_0^{\sqrt{\pi}} x \sin(x^2) \; dx$.
3. Quanto é $\int_{[0,1]^2} xy \quad dx \quad dy$? 4.Use a fórmula de mudança de variáveis para calcular $\int_0^2 \int_0^1 xy(x^2 - y^2)/(x^2 + y^2)^3 \;\; dy \;\; dx$ e $\int_0^1 \int_0^2 f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3 \;\; dx \;\; dy$ para ver que elas são diferentes.

Discussões[204]

18.6 Variáveis Aleatórias

Em Section 2.6 vimos o básico de como trabalhar com variáveis aleatórias discretas, que em nosso caso se referem àquelas variáveis aleatórias que tomam um conjunto finito de valores possíveis, ou os inteiros. Nesta seção, desenvolvemos a teoria das *variáveis aleatórias contínuas*, que são variáveis aleatórias que podem assumir qualquer valor real.

18.6.1 Variáveis Aleatórias Contínuas

Variáveis aleatórias contínuas são um tópico significativamente mais sutil do que variáveis aleatórias discretas. Uma boa analogia a ser feita é que o salto técnico é comparável ao salto entre adicionar listas de números e integrar funções. Como tal, precisaremos levar algum tempo para desenvolver a teoria.

Do Discreto ao Contínuo

Para entender os desafios técnicos adicionais encontrados ao trabalhar com variáveis aleatórias contínuas, vamos realizar um experimento de pensamento. Suponha que estejamos jogando um dardo no alvo e queremos saber a probabilidade de que ele atinja exatamente 2cm do centro do alvo.

Para começar, imaginamos medir um único dígito de precisão, ou seja, com valores para 0cm, 1cm, 2cm e assim por diante. Jogamos, digamos, 100 dardos no alvo de dardos, e se 20 deles caem no valor de 2cm, concluímos que 20% dos dardos que jogamos atingem o tabuleiro 2cm longe do centro.

No entanto, quando olhamos mais de perto, isso não corresponde à nossa pergunta! Queríamos igualdade exata, ao passo que essas caixas mantêm tudo o que está entre, digamos, 1.5cm e 2.5cm.

Sem desanimar, continuamos adiante. Medimos ainda mais precisamente, digamos 1.9cm, 2.0cm, 2.1cm, e agora vemos que talvez 3 dos 100 dardos atingiram o tabuleiro no 2.0cm balde. Assim, concluímos que a probabilidade é 3%.

Porém, isso não resolve nada! Apenas resolvemos o problema um dígito mais. Vamos abstrair um pouco. Imagine que sabemos a probabilidade de que os primeiros k dígitos correspondam a $2.00000\ldots$ e queremos saber a probabilidade de que correspondam aos primeiros $k + 1$ dígitos. É bastante razoável supor que o dígito $k + 1^{\text{th}}$ é essencialmente uma escolha aleatória do conjunto

[204] https://discuss.d2l.ai/t/1092

$\{0, 1, 2, \ldots, 9\}$. Pelo menos, não podemos conceber um processo fisicamente significativo que forçaria o número de micrômetros a se afastar do centro a preferir terminar em 7 vs 3.

O que isso significa é que, em essência, cada dígito adicional de precisão que exigimos deve diminuir a probabilidade de correspondência por um fator de 10. Ou dito de outra forma, poderíamos esperar que

$$P(\text{distance is } 2.00\ldots, \text{ to } k \text{ digits}) \approx p \cdot 10^{-k}. \tag{18.6.1}$$

O valor p essencialmente codifica o que acontece com os primeiros dígitos, e 10^{-k} trata do resto.

Observe que, se conhecermos a posição com precisão de $k = 4$ dígitos após o decimal. isso significa que sabemos que o valor está dentro do intervalo, digamos $[(1.99995, 2.00005]$, que é um intervalo de comprimento $2.00005 - 1.99995 = 10^{-4}$. Portanto, se chamarmos o comprimento deste intervalo de ϵ, podemos dizer

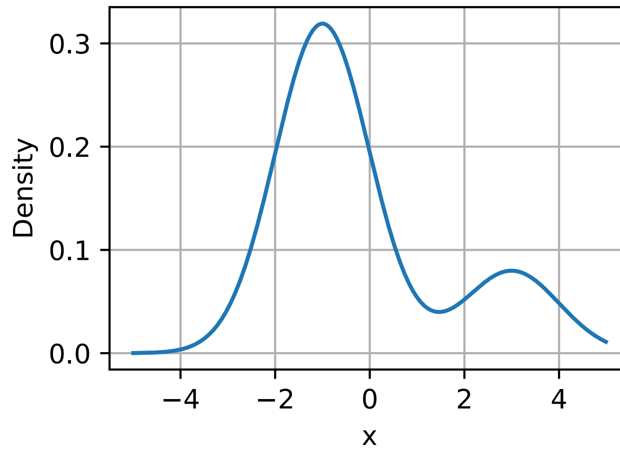
$$P(\text{distance is in an } \epsilon\text{-sized interval around } 2) \approx \epsilon \cdot p. \tag{18.6.2}$$

Vamos dar mais um passo final. Temos pensado nesse ponto 2 o tempo todo, mas nunca pensando em outros pontos. Nada é fundamentalmente diferente aqui, mas é o caso que o valor p provavelmente será diferente. Esperaríamos pelo menos que um arremessador de dardo tivesse mais probabilidade de acertar um ponto próximo ao centro, como 2cm. em vez de 20cm. Assim, o valor p não é fixo, mas deve depender do ponto x. Isso nos diz que devemos esperar

$$P(\text{distance is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{18.6.3}$$

Na verdade, (18.6.3) define com precisão a *função de densidade de probabilidade*. É uma função $p(x)$ que codifica a probabilidade relativa de acertar perto de um ponto em relação a outro. Vamos visualizar a aparência de tal função.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Define pi in torch

# Plot the probability density function for some random variable
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi)) + \
    0.8*torch.exp(-(x + 1)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi))

d2l.plot(x, p, 'x', 'Density')
```



Os locais onde o valor da função é grande indicam regiões onde é mais provável encontrar o valor aleatório. As porções baixas são áreas onde é improvável que encontremos o valor aleatório.

Funções de Densidade de Probabilidade

Vamos agora investigar isso mais detalhadamente. Já vimos o que é uma função de densidade de probabilidade intuitivamente para uma variável aleatória X, ou seja, a função de densidade é uma função $p(x)$ de modo que

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{18.6.4}$$

Mas o que isso implica para as propriedades de $p(x)$?

Primeiro, as probabilidades nunca são negativas, portanto, devemos esperar que $p(x) \geq 0$ também.

Em segundo lugar, vamos imaginar que dividimos $\mathbb{R}$ em um número infinito de fatias de ϵ de largura, digamos com fatias $(\epsilon \cdot i, \epsilon \cdot (i+1)]$. Para cada um deles, sabemos de (18.6.4) que a probabilidade é de aproximadamente

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(\epsilon \cdot i), \tag{18.6.5}$$

então, resumido sobre todos eles, deveria ser

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \tag{18.6.6}$$

Isso nada mais é do que a aproximação de uma integral discutida em Section 18.5, portanto, podemos dizer que

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x) \; dx. \tag{18.6.7}$$

Sabemos que $P(X \in \mathbb{R}) = 1$, uma vez que a variável aleatória deve assumir *algum* número, podemos concluir que para qualquer densidade

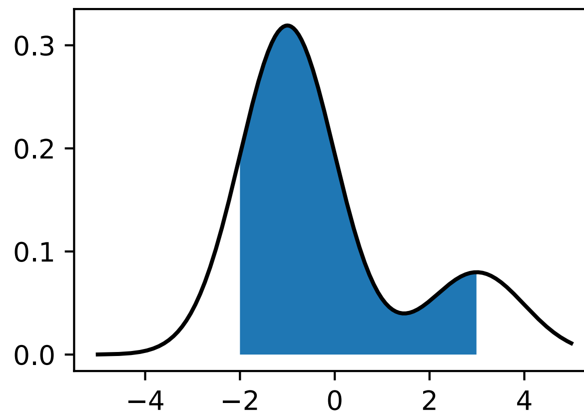
$$\int_{-\infty}^{\infty} p(x) \; dx = 1. \tag{18.6.8}$$

Na verdade, aprofundarmos mais nisso mostra que para qualquer a e b, vemos que

$$P(X \in (a, b]) = \int_{a}^{b} p(x) \; dx. \tag{18.6.9}$$

Podemos aproximar isso no código usando os mesmos métodos de aproximação discretos de antes. Nesse caso, podemos aproximar a probabilidade de cair na região azul.

```
# Approximate probability using numerical integration
epsilon = 0.01
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi)) +\
    0.8*torch.exp(-(x + 1)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi))

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

f'approximate Probability: {torch.sum(epsilon*p[300:800])}'
```




```
'approximate Probability: 0.773617148399353'
```

Acontece que essas duas propriedades descrevem exatamente o espaço das funções de densidade de probabilidade possíveis (ou *f.d.f.* 'S para a abreviação comumente encontrada). Eles são funções não negativas $p(x) \geq 0$ tais que

$$\int_{-\infty}^{\infty} p(x) \; dx = 1. \tag{18.6.10}$$

Interpretamos essa função usando integração para obter a probabilidade de nossa variável aleatória estar em um intervalo específico:

$$P(X \in (a, b]) = \int_{a}^{b} p(x) \; dx. \tag{18.6.11}$$

Em Section 18.8 veremos uma série de distribuições comuns, mas vamos continuar trabalhando de forma abstrata.

Funções de Distribuição Cumulativas

Na seção anterior, vimos a noção de f.d.p. Na prática, esse é um método comumente encontrado para discutir variáveis aleatórias contínuas, mas tem uma armadilha significativa: os valores de f.d.p. não são em si probabilidades, mas sim uma função que devemos integrar para produzir probabilidades. Não há nada de errado em uma densidade ser maior que 10, desde que não seja maior que 10 por mais de um intervalo de comprimento $1/10$. Isso pode ser contra-intuitivo, então as pessoas muitas vezes também pensam em termos da *função de distribuição cumulativa*, ou f.d.c., que *é* uma probabilidade.

Em particular, usando (18.6.11), definimos o f.d.c. para uma variável aleatória X com densidade $p(x)$ por

$$F(x) = \int_{-\infty}^{x} p(x) \; dx = P(X \leq x). \tag{18.6.12}$$

Vamos observar algumas propriedades.

- $F(x) \rightarrow 0$ as $x \rightarrow -\infty$.
- $F(x) \rightarrow 1$ as $x \rightarrow \infty$.
- $F(x)$ is non-decreasing ($y > x \implies F(y) \geq F(x)$).
- $F(x)$ is continuous (has no jumps) if X is a continuous random variable.

Com o quarto marcador, observe que isso não seria verdade se X fosse discreto, digamos, tomando os valores 0 e 1 ambos com probabilidade $1/2$. Nesse caso

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \geq 1. \end{cases} \tag{18.6.13}$$

Neste exemplo, vemos um dos benefícios de trabalhar com o cdf, a capacidade de lidar com variáveis aleatórias contínuas ou discretas na mesma estrutura, ou mesmo misturas das duas (lance uma moeda: se cara retornar o lançamento de um dado , se a cauda retornar a distância de um lançamento de dardo do centro de um alvo de dardo).

Médias

Suponha que estamos lidando com variáveis aleatórias X. A distribuição em si pode ser difícil de interpretar. Muitas vezes é útil ser capaz de resumir o comportamento de uma variável aleatória de forma concisa. Os números que nos ajudam a capturar o comportamento de uma variável aleatória são chamados de *estatísticas resumidas*. Os mais comumente encontrados são a *média*, a *variância* e o *desvio padrão*.

A *média* codifica o valor médio de uma variável aleatória. Se tivermos uma variável aleatória discreta X, que assume os valores x_i com probabilidades p_i, então a média é dada pela média ponderada: some os valores vezes a probabilidade de que a variável aleatória assuma esse valor:

$$\mu_X = E[X] = \sum_i x_i p_i. \tag{18.6.14}$$

A maneira como devemos interpretar a média (embora com cautela) é que ela nos diz essencialmente onde a variável aleatória tende a estar localizada.

Como um exemplo minimalista que examinaremos ao longo desta seção, tomemos X como a variável aleatória que assume o valor $a-2$ com probabilidade p, $a+2$ com probabilidade p e a com probabilidade $1-2p$. Podemos calcular usando (18.6.14) que, para qualquer escolha possível de a e p, a média é

$$\mu_X = E[X] = \sum_i x_i p_i = (a-2)p + a(1-2p) + (a+2)p = a. \tag{18.6.15}$$

Assim, vemos que a média é a. Isso corresponde à intuição, já que a é o local em torno do qual centralizamos nossa variável aleatória.

Por serem úteis, vamos resumir algumas propriedades.

- Para qualquer variável aleatória X e números a e b, temos que $\mu_{aX+b} = a\mu_X + b$.
- Se temos duas variáveis aleatórias X e Y, temos $\mu_{X+Y} = \mu_X + \mu_Y$.

As médias são úteis para entender o comportamento médio de uma variável aleatória, porém a média não é suficiente para ter um entendimento intuitivo completo. Ter um lucro de $\$10 \pm \$1$ por venda é muito diferente de ganhar $\$10 \pm \$15$ por venda, apesar de ter o mesmo valor médio. O segundo tem um grau de flutuação muito maior e, portanto, representa um risco muito maior. Assim, para entender o comportamento de uma variável aleatória, precisaremos, no mínimo, de mais uma medida: alguma medida de quão amplamente uma variável aleatória flutua.

Variâncias

Isso nos leva a considerar a *variância* de uma variável aleatória. Esta é uma medida quantitativa de quão longe uma variável aleatória se desvia da média. Considere a expressão $X - \mu_X$. Este é o desvio da variável aleatória de sua média. Esse valor pode ser positivo ou negativo, portanto, precisamos fazer algo para torná-lo positivo para que possamos medir a magnitude do desvio.

Uma coisa razoável a tentar é olhar para $|X - \mu_X|$, e de fato isso leva a uma quantidade útil chamada *desvio médio absoluto*, no entanto, devido a conexões com outras áreas da matemática e estatística, as pessoas costumam usar uma solução diferente.

Em particular, eles olham para $(X - \mu_X)^2.$. Se olharmos para o tamanho típico desta quantidade tomando a média, chegamos à variância

$$\sigma_X^2 = \mathrm{Var}(X) = E\left[(X-\mu_X)^2\right] = E[X^2] - \mu_X^2. \tag{18.6.16}$$

A última igualdade em (18.6.16) se mantém expandindo a definição no meio e aplicando as propriedades de expectativa.

Vejamos nosso exemplo onde X é a variável aleatória que assume o valor $a-2$ com probabilidade p, $a+2$ com probabilidade p e a com probabilidade $1-2p$. Neste caso $\mu_X = a$, então tudo que precisamos calcular é $E\left[X^2\right]$. Isso pode ser feito prontamente:

$$E\left[X^2\right] = (a-2)^2p + a^2(1-2p) + (a+2)^2p = a^2 + 8p. \tag{18.6.17}$$

Assim, vemos que por (18.6.16) nossa variância é

$$\sigma_X^2 = \mathrm{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \tag{18.6.18}$$

Este resultado novamente faz sentido. O maior p pode ser $1/2$ que corresponde a escolher $a-2$ ou $a+2$ com um cara ou coroa. A variação de ser 4 corresponde ao fato de que $a-2$ e $a+2$ estão 2

unidades de distância da média e $2^2 = 4$. Na outra extremidade do espectro, se $p = 0$, essa variável aleatória sempre assume o valor 0 e, portanto, não tem nenhuma variação.

Listaremos algumas propriedades de variação abaixo:

- Para qualquer variável aleatória X, $\mathrm{Var}(X) \geq 0$, com $\mathrm{Var}(X) = 0$ se e somente se X for uma constante.
- Para qualquer variável aleatória X e números a e b, temos que $\mathrm{Var}(aX + b) = a^2\mathrm{Var}(X)$.
- Se temos duas variáveis aleatórias *independentes* X e Y, temos $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$.

Ao interpretar esses valores, pode haver um pequeno soluço. Em particular, vamos tentar imaginar o que acontece se rastrearmos as unidades por meio desse cálculo. Suponha que estejamos trabalhando com a classificação por estrelas atribuída a um produto na página da web. Então a, $a-2$ e $a+2$ são medidos em unidades de estrelas. Da mesma forma, a média μ_X também é medida em estrelas (sendo uma média ponderada). No entanto, se chegarmos à variância, imediatamente encontramos um problema, que é queremos olhar para $(X - \mu_X)^2$, que está em unidades de *estrelas ao quadrado*. Isso significa que a própria variação não é comparável às medições originais. Para torná-lo interpretável, precisaremos retornar às nossas unidades originais.

Desvio Padrão

Essas estatísticas resumidas sempre podem ser deduzidas da variância calculando a raiz quadrada! Assim, definimos o *desvio padrão* como sendo

$$\sigma_X = \sqrt{\mathrm{Var}(X)}. \tag{18.6.19}$$

Em nosso exemplo, isso significa que agora temos o desvio padrão $\sigma_X = 2\sqrt{2p}$. Se estamos lidando com unidades de estrelas para nosso exemplo de revisão, σ_X está novamente em unidades de estrelas.

As propriedades que tínhamos para a variância podem ser reapresentadas para o desvio padrão.

- Para qualquer variável aleatória X, $\sigma_X \geq 0$.
- Para qualquer variável aleatória X e números a e b, temos que $\sigma_{aX+b} = |a|\sigma_X$
- Se temos duas variáveis aleatórias *independentes* X e Y, temos $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$.

É natural, neste momento, perguntar: "Se o desvio padrão está nas unidades de nossa variável aleatória original, ele representa algo que podemos desenhar em relação a essa variável aleatória?" A resposta é um sim retumbante! Na verdade, muito parecido com a média informada sobre a localização típica de nossa variável aleatória, o desvio padrão fornece a faixa típica de variação dessa variável aleatória. Podemos tornar isso rigoroso com o que é conhecido como desigualdade de Chebyshev:

$$P\left(X \notin [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]\right) \leq \frac{1}{\alpha^2}. \tag{18.6.20}$$

Ou, para declará-lo verbalmente no caso de $\alpha = 10$, 99% das amostras de qualquer variável aleatória se enquadram nos desvios padrão de 10 da média. Isso dá uma interpretação imediata de nossas estatísticas de resumo padrão.

Para ver como essa afirmação é bastante sutil, vamos dar uma olhada em nosso exemplo em execução novamente, onde X é a variável aleatória que assume o valor $a-2$ com probabilidade p, $a+2$

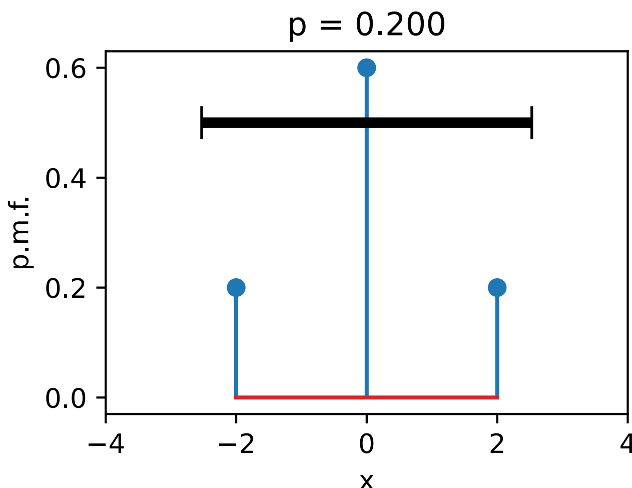
com probabilidade p e a com probabilidade $1-2p$. Vimos que a média era a e o desvio padrão era $2\sqrt{2p}$. Isso significa que, se tomarmos a desigualdade de Chebyshev (18.6.20) com $\alpha = 2$, vemos que a expressão é

$$P\left(X \not\in [a - 4\sqrt{2p}, a + 4\sqrt{2p}]\right) \leq \frac{1}{4}. \tag{18.6.21}$$

Isso significa que 75% do tempo, essa variável aleatória ficará dentro desse intervalo para qualquer valor de p. Agora, observe que como $p \rightarrow 0$, este intervalo também converge para o único ponto a. Mas sabemos que nossa variável aleatória assume os valores $a-2, a$ e $a+2$ apenas então, eventualmente, podemos ter certeza de que $a-2$ e $a+2$ ficarão fora do intervalo! A questão é: em que p isso acontece. Então, queremos resolver: para que p $a+4\sqrt{2p} = a+2$, que é resolvido quando $p = 1/8$, que é *exatamente* o primeiro p onde isso poderia acontecer sem violar nossa alegação de que não mais do que $1/4$ de amostras da distribuição ficariam fora do intervalo ($1/8$ à esquerda e $1/8$ à direita).

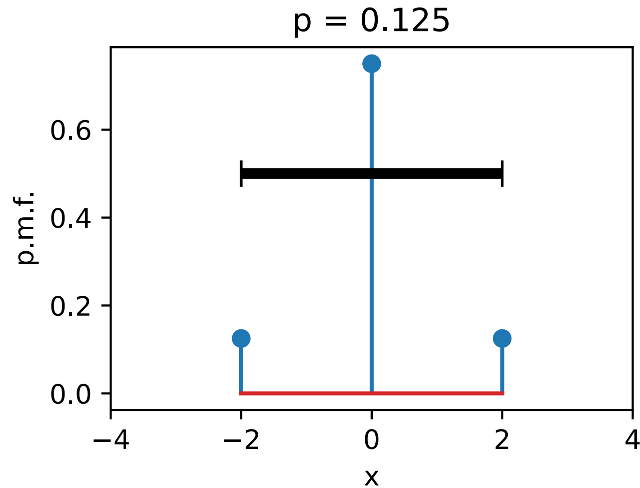
Vamos visualizar isso. Mostraremos a probabilidade de obter os três valores como três barras verticais com altura proporcional à probabilidade. O intervalo será desenhado como uma linha horizontal no meio. O primeiro gráfico mostra o que acontece para $p > 1/8$ onde o intervalo contém com segurança todos os pontos.

```python
# Define a helper to plot these figures
def plot_chebyshev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * torch.sqrt(2 * p),
                   a + 4 * torch.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(f'p = {p:.3f}')

    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebyshev(0.0, torch.tensor(0.2))
```



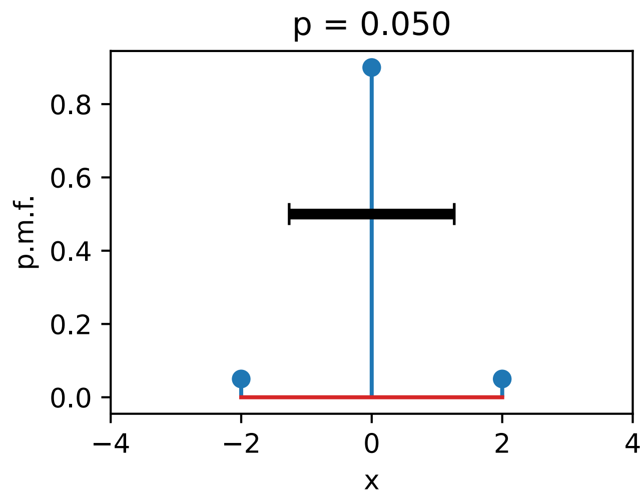
O segundo mostra que em $p = 1/8$, o intervalo toca exatamente os dois pontos. Isso mostra que a desigualdade é *nítida*, uma vez que nenhum intervalo menor pode ser obtido mantendo a desigualdade verdadeira.

```
# Plot interval when p = 1/8
plot_chebyshev(0.0, torch.tensor(0.125))
```



O terceiro mostra que para $p < 1/8$ o intervalo contém apenas o centro. Isso não invalida a desigualdade, pois só precisamos garantir que não mais do que $1/4$ da probabilidade caia fora do intervalo, o que significa que uma vez $p < 1/8$, os dois pontos em $a - 2$ e $a + 2$ podem ser descartados.

```
# Plot interval when p < 1/8
plot_chebyshev(0.0, torch.tensor(0.05))
```



Médias e Variância no Tempo Contínuo

Tudo isso em termos de variáveis aleatórias discretas, mas o caso das variáveis aleatórias contínuas é semelhante. Para entender intuitivamente como isso funciona, imagine que dividimos a reta do número real em intervalos de comprimento ϵ dados por $(\epsilon i, \epsilon(i+1)]$. Depois de fazermos isso, nossa variável aleatória contínua foi discretizada e podemos usar (18.6.14) dizer que

$$\begin{aligned}
\mu_X & \approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\
& \approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon,
\end{aligned} \tag{18.6.22}$$

onde p_X é a densidade de X. Esta é uma aproximação da integral de $xp_X(x)$, então podemos concluir que

$$\mu_X = \int_{-\infty}^\infty x p_X(x) \; dx. \tag{18.6.23}$$

Da mesma forma, usando (18.6.16) a variância pode ser escrita como

$$\sigma^2_X = E[X^2] - \mu_X^2 = \int_{-\infty}^\infty x^2 p_X(x) \; dx - \left(\int_{-\infty}^\infty x p_X(x) \; dx\right)^2. \tag{18.6.24}$$

Tudo o que foi dito acima sobre a média, a variância e o desvio padrão ainda se aplica neste caso. Por exemplo, se considerarmos a variável aleatória com densidade

$$p(x) = \begin{cases}
1 & x \in [0,1], \\
0 & \text{otherwise}.
\end{cases} \tag{18.6.25}$$

nós podemos computar

$$\mu_X = \int_{-\infty}^\infty x p(x) \; dx = \int_0^1 x \; dx = \frac{1}{2}. \tag{18.6.26}$$

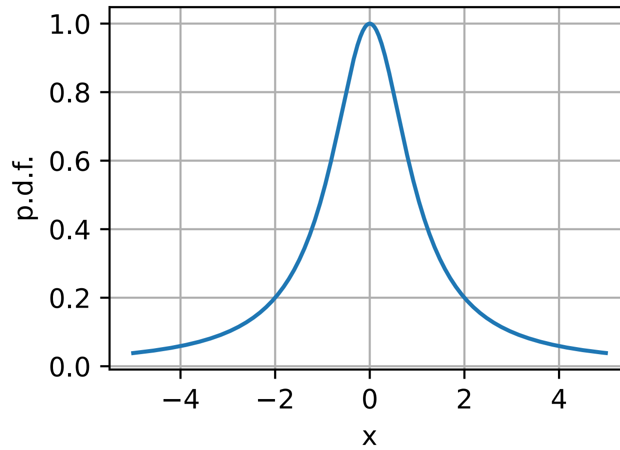
e

$$\sigma_X^2 = \int_{-\infty}^\infty x^2 p(x) \; dx - \left(\frac{1}{2}\right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{18.6.27}$$

Como um aviso, vamos examinar mais um exemplo, conhecido como a *Distribuição Cauchy*. Esta é a distribuição com f.d.p. dada por

$$p(x) = \frac{1}{1+x^2}. \tag{18.6.28}$$

```
# Plot the Cauchy distribution p.d.f.
x = torch.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```



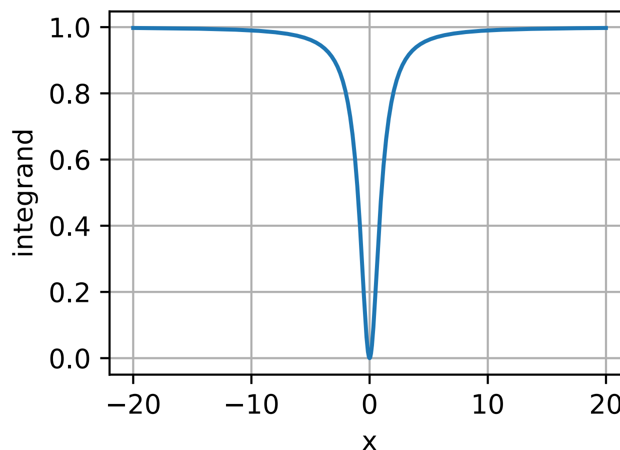
Esta função parece inocente e, de fato, consultar uma tabela de integrais mostrará que ela tem a área um abaixo dela e, portanto, define uma variável aleatória contínua.

Para ver o que está errado, vamos tentar calcular a variância disso. Isso envolveria o uso de computação (18.6.16)

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\, dx. \tag{18.6.29}$$

A função interna é semelhante a esta:

```
# Plot the integrand needed to compute the variance
x = torch.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```



Esta função tem claramente uma área infinita sob ela, uma vez que é essencialmente a constante com um pequeno mergulho próximo a zero, e de fato podemos mostrar que

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\, dx = \infty. \tag{18.6.30}$$

Isso significa que não tem uma variância finita bem definida.

No entanto, olhar mais a fundo mostra um resultado ainda mais perturbador. Vamos tentar calcular a média usando (18.6.14). Usando a fórmula de mudança de variáveis, vemos

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2} \; dx = \frac{1}{2}\int_1^\infty \frac{1}{u} \; du. \tag{18.6.31}$$

A integral interna é a definição do logaritmo, então isso é em essência $\log(\infty) = \infty$, então também não há um valor médio bem definido!

Os cientistas do aprendizado de máquina definem seus modelos de modo que, na maioria das vezes, não precisemos lidar com essas questões e, na grande maioria dos casos, lidaremos com variáveis aleatórias com médias e variações bem definidas. No entanto, de vez em quando variáveis aleatórias com *caudas pesadas* (ou seja, aquelas variáveis aleatórias onde as probabilidades de obter grandes valores são grandes o suficiente para tornar coisas como a média ou variância indefinidas) são úteis na modelagem de sistemas físicos, portanto, vale a pena saber que elas existem.

Funções de Densidade Conjunta

Todo o trabalho acima assume que estamos trabalhando com uma única variável aleatória de valor real. Mas e se estivermos lidando com duas ou mais variáveis aleatórias potencialmente altamente correlacionadas? Esta circunstância é a norma no *machine learning*: imagine variáveis aleatórias como $R_{i,j}$ que codificam o valor vermelho do pixel na coordenada (i,j) em uma imagem, ou P_t que é um variável aleatória dada por um preço de ação no momento t. Pixels próximos tendem a ter cores semelhantes e tempos próximos tendem a ter preços semelhantes. Não podemos tratá-los como variáveis aleatórias separadas e esperar criar um modelo de sucesso (veremos em Section 18.9 um modelo com desempenho inferior devido a tal suposição). Precisamos desenvolver a linguagem matemática para lidar com essas variáveis aleatórias contínuas correlacionadas.

Felizmente, com as integrais múltiplas em Section 18.5 podemos desenvolver tal linguagem. Suponha que temos, para simplificar, duas variáveis aleatórias X, Y que podem ser correlacionadas. Então, semelhante ao caso de uma única variável, podemos fazer a pergunta:

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y). \tag{18.6.32}$$

Raciocínio semelhante ao caso de variável única mostra que isso deve ser aproximadamente

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y) \approx \epsilon^{2}p(x,y), \tag{18.6.33}$$

para alguma função $p(x,y)$. Isso é conhecido como densidade conjunta de X e Y. Propriedades semelhantes são verdadeiras para isso, como vimos no caso de variável única. Nomeadamente:

- $p(x,y) \ge 0$;
- $\int_{\mathbb{R}^2} p(x,y) \; dx \; dy = 1$;
- $P((X,Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x,y) \; dx \; dy$.

Dessa forma, podemos lidar com múltiplas variáveis aleatórias potencialmente correlacionadas. Se quisermos trabalhar com mais de duas variáveis aleatórias, podemos estender a densidade multivariada para tantas coordenadas quantas desejar, considerando $p(\mathbf{x}) = p(x_1, \ldots, x_n)$. As mesmas propriedades de ser não negativo e ter integral total de um ainda são válidas.

Distribuições Marginais

Ao lidar com várias variáveis, muitas vezes queremos ser capazes de ignorar os relacionamentos e perguntar: "como essa variável é distribuída?" Essa distribuição é chamada de *distribuição marginal*.

Para ser concreto, vamos supor que temos duas variáveis aleatórias X, Y com densidade conjunta dada por $p_{X,Y}(x, y)$. Estaremos usando o subscrito para indicar para quais variáveis aleatórias se destina a densidade. A questão de encontrar a distribuição marginal é pegar essa função e usá-la para encontrar $p_X(x)$.

Como acontece com a maioria das coisas, é melhor retornar ao quadro intuitivo para descobrir o que deve ser verdade. Lembre-se de que a densidade é a função p_X para que

$$P(X \in [x, x+\epsilon]) \approx \epsilon \cdot p_X(x). \tag{18.6.34}$$

Não há menção de Y, mas se tudo o que recebemos é $p_{X,Y}$, precisamos incluir Y de alguma forma. Podemos primeiro observar que isso é o mesmo que

$$P(X \in [x, x+\epsilon], \text{and } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \tag{18.6.35}$$

Nossa densidade não nos diz diretamente sobre o que acontece neste caso, precisamos dividir em pequenos intervalos em y também, para que possamos escrever isso como

$$\begin{aligned}\epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x+\epsilon], \text{and } Y \in [\epsilon \cdot i, \epsilon \cdot (i+1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i).\end{aligned} \tag{18.6.36}$$

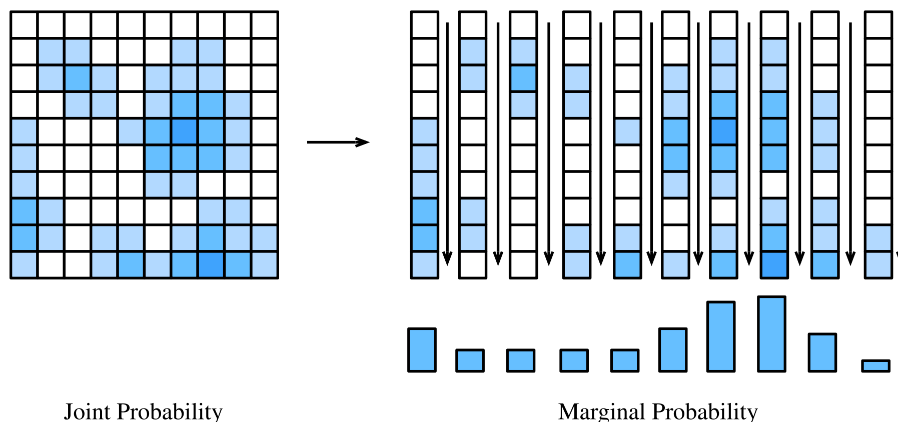


Fig. 18.6.1: Somando ao longo das colunas de nosso conjunto de probabilidades, podemos obter a distribuição marginal apenas para a variável aleatória representada ao longo do eixo x.

Isso nos diz para somar o valor da densidade ao longo de uma série de quadrados em uma linha, como é mostrado em Fig. 18.6.1. De fato, depois de cancelar um fator de épsilon de ambos os lados, e reconhecer que a soma à direita é a integral sobre y, podemos concluir que

$$\begin{aligned}p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y) \; dy.\end{aligned} \tag{18.6.37}$$

Assim vemos

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x,y) \; dy. \tag{18.6.38}$$

Isso nos diz que, para obter uma distribuição marginal, integramos as variáveis com as quais não nos importamos. Este processo é frequentemente referido como *integração* ou *marginalização* das variáveis desnecessárias.

Covariância

Ao lidar com várias variáveis aleatórias, há uma estatística de resumo adicional que é útil saber: a *covariância*. Isso mede o grau em que duas variáveis aleatórias flutuam juntas.

Suponha que temos duas variáveis aleatórias X e Y, para começar, suponhamos que sejam discretas, assumindo os valores (x_i, y_j) com probabilidade p_{ij}. Neste caso, a covariância é definida como

$$\sigma_{XY} = \textrm{Cov}(X,Y) = \sum_{i,j} (x_i - \mu_X) (y_j-\mu_Y) p_{ij}. = E[XY] - E[X]E[Y]. \tag{18.6.39}$$

Para pensar sobre isso intuitivamente: considere o seguinte par de variáveis aleatórias. Suponha que X assume os valores 1 e 3, e Y assume os valores -1 e 3. Suponha que temos as seguintes probabilidades

$$\begin{aligned}
P(X = 1 \; \textrm{and} \; Y = -1) & = \frac{p}{2}, \\
P(X = 1 \; \textrm{and} \; Y = 3) & = \frac{1-p}{2}, \\
P(X = 3 \; \textrm{and} \; Y = -1) & = \frac{1-p}{2}, \\
P(X = 3 \; \textrm{and} \; Y = 3) & = \frac{p}{2},
\end{aligned} \tag{18.6.40}$$

onde p é um parâmetro em $[0,1]$ que escolhemos. Observe que se $p=1$ então eles são sempre seus valores mínimo ou máximo simultaneamente, e se $p=0$ eles têm a garantia de obter seus valores invertidos simultaneamente (um é grande quando o outro é pequeno e vice-versa). Se $p=1/2$, então as quatro possibilidades são todas igualmente prováveis e nenhuma deve estar relacionada. Vamos calcular a covariância. Primeiro, observe $\mu_X = 2$ e $\mu_Y = 1$, então podemos calcular usando (18.6.39):

$$\begin{aligned}
\textrm{Cov}(X,Y) & = \sum_{i,j} (x_i - \mu_X) (y_j-\mu_Y) p_{ij} \\
& = (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2} \\
& = 4p-2.
\end{aligned} \tag{18.6.41}$$

Quando $p=1$ (o caso em que ambos são maximamente positivos ou negativos ao mesmo tempo) tem uma covariância de 2. Quando $p=0$ (o caso em que eles são invertidos) a covariância é -2. Finalmente, quando $p=1/2$ (o caso em que não estão relacionados), a covariância é 0. Assim, vemos que a covariância mede como essas duas variáveis aleatórias estão relacionadas.

Uma nota rápida sobre a covariância é que ela mede apenas essas relações lineares. Relacionamentos mais complexos, como $X = Y^2$, em que Y é escolhido aleatoriamente de $\{-2, -1, 0, 1, 2\}$

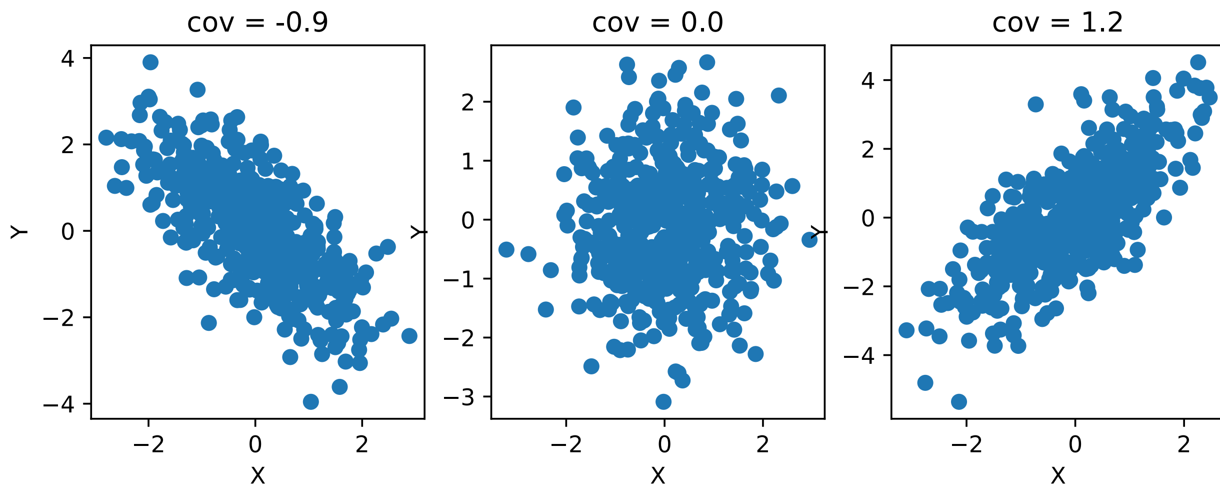
com igual probabilidade, podem ser perdidos. De fato, um cálculo rápido mostra que essas variáveis aleatórias têm covariância zero, apesar de uma ser uma função determinística da outra.

Para variáveis aleatórias contínuas, a mesma história se aplica. Neste ponto, estamos bastante confortáveis em fazer a transição entre discreto e contínuo, então forneceremos o análogo contínuo de (18.6.39) sem qualquer derivação.

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x-\mu_X)(y-\mu_Y)p(x, y) \;dx \;dy. \tag{18.6.42}$$

Para visualização, vamos dar uma olhada em uma coleção de variáveis aleatórias com covariância ajustável.

```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = covs[i]*X + torch.randn(500)

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```



Vejamos algumas propriedades das covariâncias:

- Para qualquer variável aleatória X, $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$.
- Para quaisquer variáveis aleatórias X, Y e números a e b, $\mathrm{Cov}(aX+b, Y) = \mathrm{Cov}(X, aY+b) = a\mathrm{Cov}(X, Y)$.
- Se X e Y são independentes, então $\mathrm{Cov}(X, Y) = 0$.

Além disso, podemos usar a covariância para expandir um relacionamento que vimos antes. Lembre-se de que X e Y são duas variáveis aleatórias independentes, então

$$\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{18.6.43}$$

Com o conhecimento das covariâncias, podemos expandir essa relação. Na verdade, alguma álgebra pode mostrar que, em geral,

$$\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X, Y). \tag{18.6.44}$$

Isso nos permite generalizar a regra de soma de variância para variáveis aleatórias correlacionadas.

Correlation

Como fizemos no caso das médias e variações, consideremos agora as unidades. Se X é medido em uma unidade (digamos polegadas) e Y é medido em outra (digamos dólares), a covariância é medida no produto dessas duas unidades inches $\times$ dollars. Essas unidades podem ser difíceis de interpretar. O que muitas vezes queremos neste caso é uma medida de parentesco sem unidade. Na verdade, muitas vezes não nos importamos com a correlação quantitativa exata, mas, em vez disso, perguntamos se a correlação está na mesma direção e quão forte é a relação.

Para ver o que faz sentido, vamos realizar um experimento mental. Suponha que convertamos nossas variáveis aleatórias em polegadas e dólares em polegadas e centavos. Nesse caso, a variável aleatória Y é multiplicada por 100. Se trabalharmos com a definição, isso significa que $\mathrm{Cov}(X, Y)$ será multiplicado por 100. Assim, vemos que, neste caso, uma mudança de unidades altera a covariância por um fator de 100. Assim, para encontrar nossa medida de correlação invariante à unidade, precisaremos dividir por algo mais que também é escalado por 100. Na verdade, temos um candidato claro, o desvio padrão! Na verdade, se definirmos o *coeficiente de correlação* como sendo

$$\rho(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_{X}\sigma_{Y}}, \tag{18.6.45}$$

vemos que este é um valor sem unidade. Um pouco de matemática pode mostrar que este número está entre -1 e 1 com 1 significando correlacionado positivamente ao máximo, enquanto -1 significa correlacionado negativamente ao máximo.

Voltando ao nosso exemplo discreto explícito acima, podemos ver que $\sigma_X = 1$ and $\sigma_Y = 2$, então podemos calcular a correlação entre as duas variáveis aleatórias usando (18.6.45) para ver que

$$\rho(X, Y) = \frac{4p-2}{1\cdot 2} = 2p-1. \tag{18.6.46}$$

Isso agora varia entre -1 e 1 com o comportamento esperado de 1 significando mais correlacionado e -1 significando minimamente correlacionado.

Como outro exemplo, considere X como qualquer variável aleatória e $Y = aX + b$ como qualquer função determinística linear de X. Então, pode-se calcular que

$$\sigma_{Y} = \sigma_{aX+b} = |a|\sigma_{X}, \tag{18.6.47}$$

$$\mathrm{Cov}(X, Y) = \mathrm{Cov}(X, aX+b) = a\mathrm{Cov}(X, X) = a\mathrm{Var}(X), \tag{18.6.48}$$

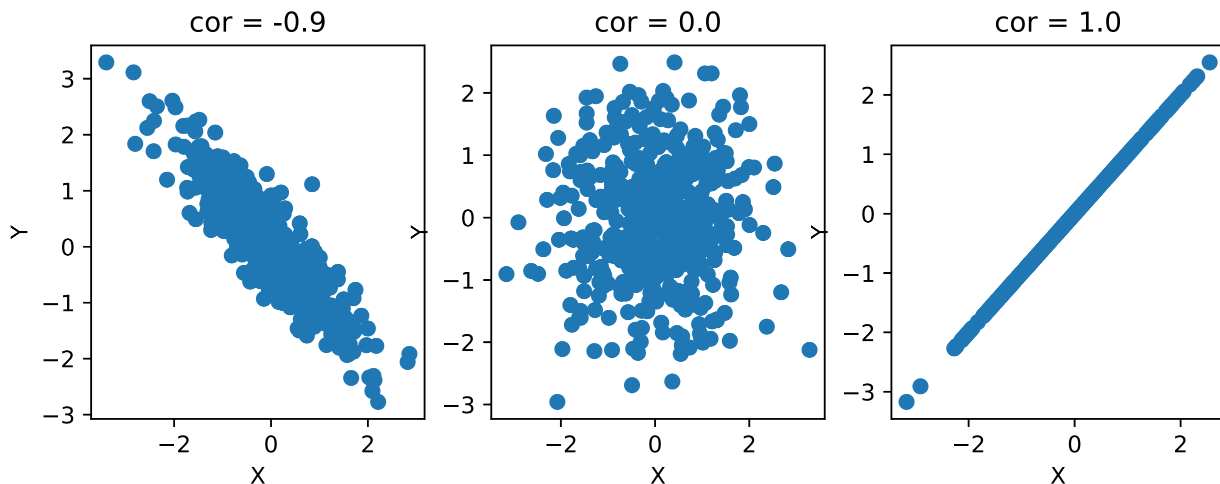
e assim por (18.6.45) que

$$\rho(X, Y) = \frac{a\mathrm{Var}(X)}{|a|\sigma_{X}^2} = \frac{a}{|a|} = \mathrm{sign}(a). \tag{18.6.49}$$

Assim, vemos que a correlação é $+1$ para qualquer $a > 0$, e -1 para qualquer $a < 0$, ilustrando que a correlação mede o grau e a direcionalidade em que as duas variáveis aleatórias estão relacionadas, não a escala que a variação leva.

Deixe-nos mais uma vez plotar uma coleção de variáveis aleatórias com correlação ajustável.

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = cors[i] * X + torch.sqrt(torch.tensor(1) -
                                 cors[i]**2) * torch.randn(500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```



Vamos listar algumas propriedades da correlação abaixo.

- Para qualquer variável aleatória X, $\rho(X, X) = 1$.
- Para quaisquer variáveis aleatórias X, Y e números a e b, $\rho(aX + b, Y) = \rho(X, aY + b) = \rho(X, Y)$.
- Se X e Y são independentes com variância diferente de zero então $\rho(X, Y) = 0$.

Como nota final, você pode achar que algumas dessas fórmulas são familiares. Na verdade, se expandirmos tudo assumindo que $\mu_X = \mu_Y = 0$, vemos que isso é

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}}\sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \tag{18.6.50}$$

Isso se parece com a soma de um produto dos termos dividido pela raiz quadrada das somas dos termos. Esta é exatamente a fórmula para o cosseno do ângulo entre dois vetores $\mathbf{v}, \mathbf{w}$ com as diferentes coordenadas ponderadas por p_{ij}:

$$\cos(\theta) = \frac{\mathbf{v}\cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2}\sqrt{\sum_i w_i^2}}. \tag{18.6.51}$$

Na verdade, se pensarmos nas normas como sendo relacionadas a desvios-padrão e correlações como cossenos de ângulos, muito da intuição que temos da geometria pode ser aplicada ao pensamento sobre variáveis aleatórias.

18.6.2 Resumo

- Variáveis aleatórias contínuas são variáveis aleatórias que podem assumir uma sequência contínua de valores. Eles têm algumas dificuldades técnicas que os tornam mais desafiadores de trabalhar em comparação com variáveis aleatórias discretas.
- A função de densidade de probabilidade nos permite trabalhar com variáveis aleatórias contínuas, fornecendo uma função em que a área sob a curva em algum intervalo fornece a probabilidade de encontrar um ponto de amostra nesse intervalo.
- A função de distribuição cumulativa é a probabilidade de observar que a variável aleatória é menor que um determinado limite. Ele pode fornecer um ponto de vista alternativo útil que unifica variáveis discretas e contínuas.
- A média é o valor médio de uma variável aleatória.
- A variância é o quadrado esperado da diferença entre a variável aleatória e sua média.
- O desvio padrão é a raiz quadrada da variância. Pode ser pensado como medir a faixa de valores que a variável aleatória pode assumir.
- A desigualdade de Chebyshev nos permite tornar essa intuição rigorosa, fornecendo um intervalo explícito que contém a variável aleatória na maior parte do tempo.
- As densidades conjuntas nos permitem trabalhar com variáveis aleatórias correlacionadas. Podemos marginalizar as densidades conjuntas integrando variáveis aleatórias indesejadas para obter a distribuição da variável aleatória desejada.
- A covariância e o coeficiente de correlação fornecem uma maneira de medir qualquer relação linear entre duas variáveis aleatórias correlacionadas.

18.6.3 Exercícios

1. Suponha que temos a variável aleatória com densidade dada por $p(x) = \frac{1}{x^2}$ para $x \geq 1$ e $p(x) = 0$ caso contrário. Quanto é $P(X > 2)$?
2. A distribuição de Laplace é uma variável aleatória cuja densidade é dada por $p(x = \frac{1}{2}e^{-|x|}$. Qual é a média e o desvio padrão desta função? uma dica, $\int_0^\infty xe^{-x}\, dx = 1$ e $\int_0^\infty x^2e^{-x}\, dx = 2$.
3. Eu ando até você na rua e digo "Eu tenho uma variável aleatória com média 1, desvio padrão 2, e observei 25% de minhas amostras tendo um valor maior que 9." Você acredita em mim? Por que ou por que não?
4. Suponha que você tenha duas variáveis aleatórias X, Y, com densidade conjunta dada por $p_{XY}(x, y) = 4xy$ para $x, y \in [0, 1]$ e $p_{XY}(x, y) = 0$ caso contrário. Qual é a covariância de X e Y?

Discussões[205]

[205] https://discuss.d2l.ai/t/1094

18.7 Máxima verossimilhança

Uma das formas de pensar mais comumente encontradas no aprendizado de máquina é o ponto de vista de máxima verossimilhança. É o conceito de que ao trabalhar com um modelo probabilístico com parâmetros desconhecidos, os parâmetros que fazem os dados terem maior probabilidade são os mais prováveis.

18.7.1 O Princípio da Máxima Verossimilhança

Isso tem uma interpretação bayesiana sobre a qual pode ser útil pensar. Suponha que tenhamos um modelo com parâmetros $\boldsymbol{\theta}$ e uma coleção de exemplos de dados $ X $. Para concretizar, podemos imaginar que $\boldsymbol{\theta}$ é um valor único que representa a probabilidade de que uma moeda dê cara quando lançada, e X é uma sequência de lançamentos independentes de moedas. Veremos este exemplo em profundidade mais tarde.

Se quisermos encontrar o valor mais provável para os parâmetros do nosso modelo, isso significa que queremos encontrar

$$\mathop{\mathrm{argmax}} P(\boldsymbol{\theta} \mid X). \tag{18.7.1}$$

Pela regra de Bayes, isso é a mesma coisa que

$$\mathop{\mathrm{argmax}} \frac{P(X \mid \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(X)}. \tag{18.7.2}$$

A expressão $P(X)$, um parâmetro de probabilidade agnóstica de gerar os dados, não depende de $\boldsymbol{\theta}$ e, portanto, pode ser descartada sem alterar a melhor escolha de $\boldsymbol{\theta}$. Da mesma forma, podemos agora postular que não temos nenhuma suposição anterior sobre qual conjunto de parâmetros é melhor do que qualquer outro, então podemos declarar que $P(\boldsymbol{\theta})$ também não depende de teta! Isso, por exemplo, faz sentido em nosso exemplo de cara ou coroa, onde a probabilidade de dar cara pode ser qualquer valor em $ [0,1] $ sem qualquer crença prévia de que seja justo ou não (muitas vezes referido como um *prévio não informativo*) Assim, vemos que nossa aplicação da regra de Bayes mostra que nossa melhor escolha de $\boldsymbol{\theta}$ é a estimativa de máxima verossimilhança para $\boldsymbol{\theta}$:

$$\hat{\boldsymbol{\theta}} = \mathop{\mathrm{argmax}}_{\boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}). \tag{18.7.3}$$

Como uma terminologia comum, a probabilidade dos dados dados os parâmetros ($P(X \mid \boldsymbol{\theta})$) é conhecida como *probabilidade*.

Um exemplo concreto

Vamos ver como isso funciona em um exemplo concreto. Suponha que tenhamos um único parâmetro θ representando a probabilidade de cara ao cara. Então a probabilidade de obter uma coroa é $1-\theta$, e se nossos dados observados X são uma sequência com n_H cara e n_T coroa, podemos usar o fato de que as probabilidades independentes se multiplicam para ver que

$$P(X \mid \theta) = \theta^{n_H}(1-\theta)^{n_T}. \tag{18.7.4}$$

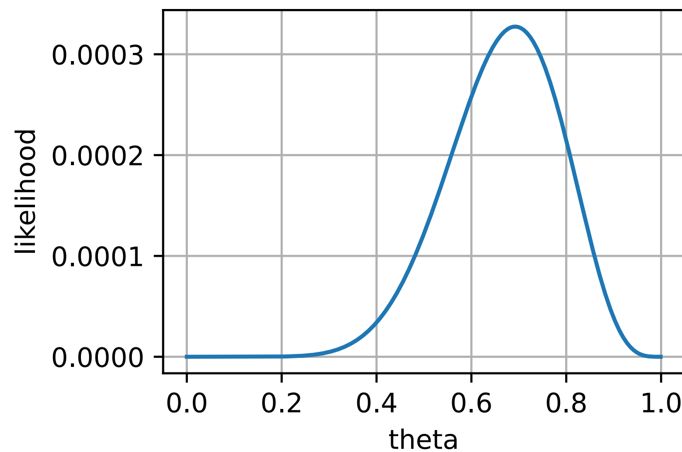
Se lançarmos 13 em moedas e obtivermos a sequência "HHHTHTTHHHHHT", que tem $n_H = 9$ e $n_T = 4$, vemos que isso é

$$P(X \mid \theta) = \theta^9(1-\theta)^4. \tag{18.7.5}$$

Uma coisa boa sobre esse exemplo é que sabemos qual será a resposta. Na verdade, se disséssemos verbalmente: "Joguei 13 moedas e deram cara com 9, qual é nossa melhor estimativa para a probabilidade de que a moeda nos dê cara? ,"todos acertariam $9/13$. O que este método de máxima verossimilhança nos dará é uma maneira de obter esse número dos primeiros principais de uma forma que irá generalizar para situações muito mais complexas.

Para nosso exemplo, o gráfico de $P(X \mid \theta)$ é o seguinte:

```
%matplotlib inline
import torch
from d2l import torch as d2l

theta = torch.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```



Isso tem seu valor máximo em algum lugar perto de nossos $9/13 \approx 0.7\ldots$. Para ver se está exatamente lá, podemos recorrer ao cálculo. Observe que, no máximo, a função é plana. Assim, poderíamos encontrar a estimativa de máxima verossimilhança (18.7.1) encontrando os valores de θ onde a derivada é zero, e encontrando aquele que dá a maior probabilidade. Calculamos:

$$\begin{aligned}
0 & = \frac{d}{d\theta} P(X \mid \theta) \\
& = \frac{d}{d\theta} \theta^9(1-\theta)^4 \\
& = 9\theta^8(1-\theta)^4 - 4\theta^9(1-\theta)^3 \\
& = \theta^8(1-\theta)^3(9-13\theta).
\end{aligned} \tag{18.7.6}$$

Isso tem três soluções: 0, 1 e $9/13$. Os dois primeiros são claramente mínimos, não máximos, pois atribuem probabilidade 0 à nossa sequência. O valor final *não* atribui probabilidade zero à nossa sequência e, portanto, deve ser a estimativa de máxima verossimilhança $\hat{\theta} = 9/13$.

18.7.2 Otimização Numérica e o Log-Probabilidade Negativa

O exemplo anterior é bom, mas e se tivermos bilhões de parâmetros e exemplos de dados.

Primeiro observe que, se fizermos a suposição de que todos os exemplos de dados são independentes, não podemos mais considerar a probabilidade em si na prática, pois ela é um produto de muitas probabilidades. Na verdade, cada probabilidade está em $[0, 1]$, digamos tipicamente com um valor de cerca de $1/2$, e o produto de $(1/2)^{1000000000}$ está muito abaixo da precisão da máquina. Não podemos trabalhar com isso diretamente.

No entanto, lembre-se que o logaritmo transforma produtos em somas, caso em que

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6 \ldots \tag{18.7.7}$$

Esse número se encaixa perfeitamente até mesmo em um único float de 32 bits de precisão. Assim, devemos considerar o *log-verossimilhança*, que é

$$\log(P(X \mid \boldsymbol{\theta})). \tag{18.7.8}$$

Como a função $x \mapsto \log(x)$ está aumentando, maximizar a verossimilhança é o mesmo que maximizar o log-verossimilhança. De fato, em Section 18.9 veremos esse raciocínio aplicado ao trabalhar com o exemplo específico do classificador Bayes ingênuo.

Frequentemente trabalhamos com funções de perda, onde desejamos minimizar a perda. Podemos transformar a probabilidade máxima na minimização de uma perda tomando $-\log(P(X \mid \boldsymbol{\theta}))$, que é o *log-verossimilhança negativo*.

Para ilustrar isso, considere o problema anterior de jogar a moeda e finja que não conhecemos a solução da forma fechada. Podemos calcular que

$$-\log(P(X \mid \boldsymbol{\theta})) = -\log(\theta^{n_H}(1-\theta)^{n_T}) = -(n_H\log(\theta) + n_T\log(1-\theta)). \tag{18.7.9}$$

Isso pode ser escrito em código e otimizado gratuitamente até mesmo para bilhões de cara ou coroa.

```
# Set up our data
n_H = 8675309
n_T = 25624

# Initialize our paramteres
theta = torch.tensor(0.5, requires_grad=True)

# Perform gradient descent
lr = 0.00000000001
for iter in range(10):
    loss = -(n_H * torch.log(theta) + n_T * torch.log(1 - theta))
    loss.backward()
    with torch.no_grad():
        theta -= lr * theta.grad
    theta.grad.zero_()

# Check output
theta, n_H / (n_H + n_T)
```

```
(tensor(0.5017, requires_grad=True), 0.9970550284664874)
```

A conveniência numérica é apenas uma das razões pelas quais as pessoas gostam de usar probabilidades de log negativo. Na verdade, há várias razões pelas quais pode ser preferível.

A segunda razão pela qual consideramos a log-verossimilhança é a aplicação simplificada das regras de cálculo. Conforme discutido acima, devido às suposições de independência, a maioria das probabilidades que encontramos no aprendizado de máquina são produtos de probabilidades individuais.

$$P(X \mid \boldsymbol{\theta}) = p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \tag{18.7.10}$$

Isso significa que se aplicarmos diretamente a regra do produto para calcular uma derivada, obtemos

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}) & = \left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_1\mid\boldsymbol{\theta})\right)\cdot P(x_2\mid\boldsymbol{\theta})\cdots P(x_n\mid\boldsymbol{\theta}) \\ & \quad + P(x_1\mid\boldsymbol{\theta})\cdot \left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_2\mid\boldsymbol{\theta})\right)\cdots P(x_n\mid\boldsymbol{\theta}) \\ & \quad \quad \quad \quad \quad \quad \quad \quad \quad \vdots \\ & \quad + P(x_1\mid\boldsymbol{\theta})\cdot P(x_2\mid\boldsymbol{\theta}) \cdots \left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_n\mid\boldsymbol{\theta})\right). \end{aligned} \tag{18.7.11}$$

Isso requer $n(n-1)$ multiplicações, junto com $(n-1)$ adições, então é o total do tempo quadrático nas entradas! Inteligência suficiente em termos de agrupamento reduzirá isso ao tempo linear, mas requer alguma reflexão. Para a probabilidade de log negativo, temos, em vez disso,

$$-\log\left(P(X\mid\boldsymbol{\theta})\right) = -\log(P(x_1\mid\boldsymbol{\theta})) - \log(P(x_2\mid\boldsymbol{\theta})) \cdots - \log(P(x_n\mid\boldsymbol{\theta})), \tag{18.7.12}$$

que então dá

$$- \frac{\partial}{\partial \boldsymbol{\theta}} \log\left(P(X\mid\boldsymbol{\theta})\right) = \frac{1}{P(x_1\mid\boldsymbol{\theta})}\left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_1\mid\boldsymbol{\theta})\right) + \cdots + \frac{1}{P(x_n\mid\boldsymbol{\theta})}\left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_n\mid\boldsymbol{\theta})\right). \tag{18.7.13}$$

Isso requer apenas n divisões e $n-1$ somas e, portanto, é o tempo linear nas entradas.

A terceira e última razão para considerar o log-verossimilhança negativo é a relação com a teoria da informação, que discutiremos em detalhes em Section 18.11. Esta é uma teoria matemática rigorosa que fornece uma maneira de medir o grau de informação ou aleatoriedade em uma variável aleatória. O principal objeto de estudo nesse campo é a entropia, que é

$$H(p) = -\sum_i p_i \log_2(p_i), \tag{18.7.14}$$

que mede a aleatoriedade de uma fonte. Observe que isso nada mais é do que a probabilidade média de $-\log$ e, portanto, se tomarmos nosso log de verossimilhança negativo e dividirmos pelo número de exemplos de dados, obteremos um relativo de entropia conhecido como entropia cruzada. Essa interpretação teórica por si só seria suficientemente atraente para motivar o relato da probabilidade logarítmica negativa média sobre o conjunto de dados como uma forma de medir o desempenho do modelo.

18.7.3 Máxima probabilidade para variáveis contínuas

Tudo o que fizemos até agora assume que estamos trabalhando com variáveis aleatórias discretas, mas e se quisermos trabalhar com variáveis contínuas?

O breve resumo é que nada muda, exceto que substituímos todas as instâncias da probabilidade pela densidade de probabilidade. Lembrando que escrevemos densidades com p minúsculo, isso significa que, por exemplo, agora dizemos

$$-\log\left(p(X \mid \boldsymbol{\theta})\right) = -\log(p(x_1 \mid \boldsymbol{\theta})) - \log(p(x_2 \mid \boldsymbol{\theta})) \cdots - \log(p(x_n \mid \boldsymbol{\theta})) = -\sum_i \log(p(x_i \mid \theta)). \tag{18.7.15}$$

A questão passa a ser: "Por que está tudo bem?" Afinal, a razão pela qual introduzimos densidades foi porque as probabilidades de obter resultados específicos eram zero e, portanto, a probabilidade de gerar nossos dados para qualquer conjunto de parâmetros não é zero?

Na verdade, esse é o caso, e entender por que podemos mudar para densidades é um exercício de rastrear o que acontece com os ípsilons.

Vamos primeiro redefinir nosso objetivo. Suponha que, para variáveis aleatórias contínuas, não desejemos mais calcular a probabilidade de obter exatamente o valor correto, mas, em vez disso, fazer a correspondência dentro de algum intervalo ϵ. Para simplificar, assumimos que nossos dados são observações repetidas $x_1, \ldots, x_N$ de variáveis aleatórias distribuídas de forma idêntica $X_1, \ldots, X_N$. Como vimos anteriormente, isso pode ser escrito como

$$\begin{aligned} &P(X_1 \in [x_1, x_1+\epsilon], X_2 \in [x_2, x_2+\epsilon], \ldots, X_N \in [x_N, x_N+\epsilon] \mid \boldsymbol{\theta}) \\ \approx&\epsilon^N p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \end{aligned} \tag{18.7.16}$$

Assim, se tomarmos logaritmos negativos disso, obtemos

$$\begin{aligned} &-\log(P(X_1 \in [x_1, x_1+\epsilon], X_2 \in [x_2, x_2+\epsilon], \ldots, X_N \in [x_N, x_N+\epsilon] \mid \boldsymbol{\theta})) \\ \approx &-N\log(\epsilon) - \sum_i \log(p(x_i \mid \boldsymbol{\theta})). \end{aligned} \tag{18.7.17}$$

Se examinarmos esta expressão, o único lugar onde o ϵ ocorre é na constante aditiva $ -N log (epsilon) $. Isso não depende dos parâmetros $\boldsymbol{\theta}$ de forma alguma, então a escolha ótima de $\boldsymbol{\theta}$ não depende de nossa escolha de ϵ! Se exigirmos quatro dígitos ou quatrocentos, a melhor escolha de $\boldsymbol{\theta}$ permanece a mesma, portanto, podemos descartar livremente o ípsilon para ver que o que queremos otimizar é

$$-\sum_i \log(p(x_i \mid \boldsymbol{\theta})). \tag{18.7.18}$$

Assim, vemos que o ponto de vista de máxima verossimilhança pode operar com variáveis aleatórias contínuas tão facilmente quanto com variáveis discretas, substituindo as probabilidades por densidades de probabilidade.

18.7.4 Resumo

- O princípio de máxima verossimilhança nos diz que o modelo de melhor ajuste para um determinado conjunto de dados é aquele que gera os dados com a maior probabilidade.
- Freqüentemente, as pessoas trabalham com a probabilidade logarítmica negativa por uma variedade de razões: estabilidade numérica, conversão de produtos em somas (e a simplificação resultante dos cálculos de gradiente) e vínculos teóricos com a teoria da informação.
- Embora seja mais simples de motivar na configuração discreta, pode ser generalizado livremente para a configuração contínua, bem como maximizando a densidade de probabilidade atribuída aos pontos de dados.

18.7.5 Exercícios

1. Suponha que você saiba que uma variável aleatória tem densidade $\frac{1}{\alpha}e^{-\alpha x}$ para algum valor α. Você obtém uma única observação da variável aleatória que é o número 3. Qual é a estimativa de máxima verossimilhança para α?
2. Suponha que você tenha um conjunto de dados de amostras $\{x_i\}_{i=1}^N$ extraído de um Gaussiano com média desconhecida, mas variância 1. Qual é a estimativa de máxima verossimilhança para a média?

Discussão[206]

18.8 Distribuições

Agora que aprendemos como trabalhar com probabilidade tanto na configuração discreta quanto na contínua, vamos conhecer algumas das distribuições comuns encontradas. Dependendo da área de *machine learning*, podemos precisar estar familiarizados com muito mais delas ou, para algumas áreas de *deep learning*, possivelmente nenhuma. Esta é, no entanto, uma boa lista básica para se familiarizar. Vamos primeiro importar algumas bibliotecas comuns.

```
%matplotlib inline
from math import erf, factorial
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)) * 2  # Define pi in torch
```

[206] https://discuss.d2l.ai/t/1096

18.8.1 Bernoulli

Esta é a variável aleatória mais simples normalmente encontrada. Esta variável aleatória codifica um lançamento de moeda que dá 1 com probabilidade p e 0 com probabilidade $1-p$. Se tivermos uma variável aleatória X com esta distribuição, vamos escrever

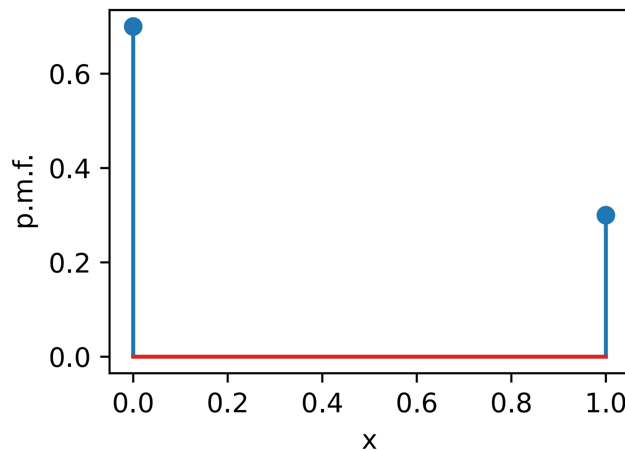
$$X \sim \mathrm{Bernoulli}(p). \tag{18.8.1}$$

A função de distribuição cumulativa é

$$F(x) = \begin{cases} 0 & x < 0, \\ 1-p & 0 \le x < 1, \\ 1 & x >= 1. \end{cases} \tag{18.8.2}$$

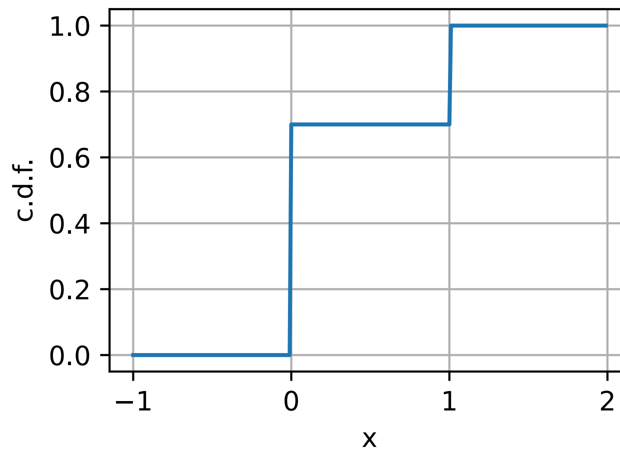
A função de massa de probabilidade está representada abaixo.

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Agora, vamos representar graficamente a função de distribuição cumulativa (18.8.2).

```
x = torch.arange(-1, 2, 0.01)

def F(x):
    return 0 if x < 0 else 1 if x > 1 else 1 - p

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



Se $X \sim \text{Bernoulli}(p)$, então:

- $\mu_X = p$,
- $\sigma_X^2 = p(1-p)$.

Podemos amostrar uma matriz de forma arbitrária de uma variável aleatória de Bernoulli como segue.

```
1*(torch.rand(10, 10) < p)
```

```
tensor([[0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
        [0, 0, 1, 1, 0, 0, 0, 0, 0, 1],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 1, 1, 0, 0, 1, 1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 1, 1, 0, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
        [0, 0, 1, 0, 0, 1, 0, 1, 0, 0]])
```

18.8.2 Uniforme e Discreta

A próxima variável aleatória comumente encontrada é uma uniforme discreta. Para nossa discussão aqui, assumiremos que é suportada nos inteiros $\{1, 2, \ldots, n\}$, entretanto qualquer outro conjunto de valores pode ser escolhido livremente. O significado da palavra *uniforme* neste contexto é que todos os valores possíveis são igualmente prováveis. A probabilidade para cada valor $i \in \{1, 2, 3, \ldots, n\}$ é $p_i = \frac{1}{n}$. Vamos denotar uma variável aleatória X com esta distribuição como

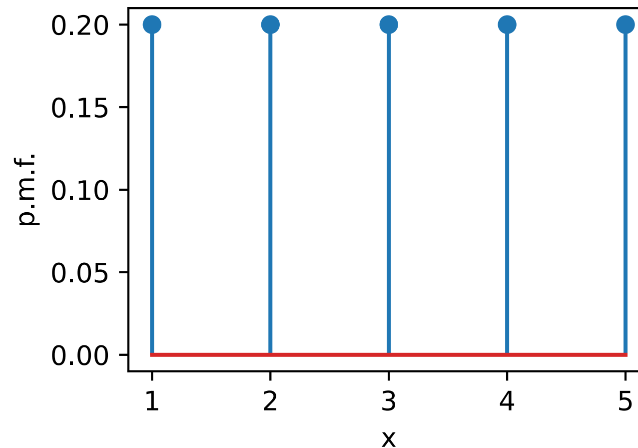
$$X \sim U(n). \tag{18.8.3}$$

A função de distribuição cumulativa é

$$F(x) = \begin{cases} 0 & x < 1, \\ \frac{k}{n} & k \leq x < k+1 \text{ with } 1 \leq k < n, \\ 1 & x >= n. \end{cases} \tag{18.8.4}$$

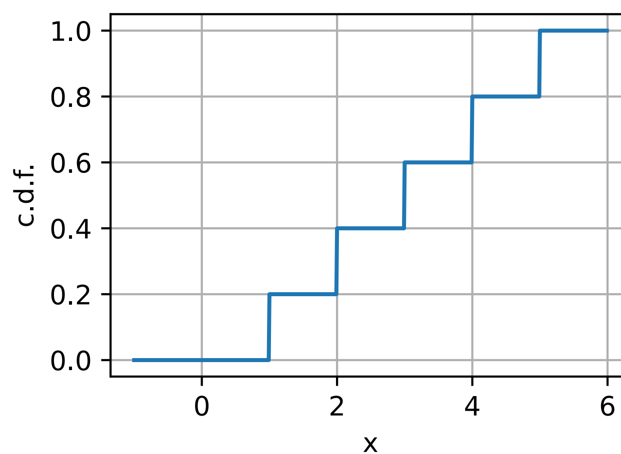
Deixe-nos primeiro representar graficamente a função de massa de probabilidade.

```
n = 5

d2l.plt.stem([i+1 for i in range(n)], n*[1 / n], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Agora, vamos representar graficamente a função de distribuição cumulativa: eqref:eq_discrete_uniform_cdf.

```
x = torch.arange(-1, 6, 0.01)

def F(x):
    return 0 if x < 1 else 1 if x > n else torch.floor(x) / n

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



If $X \sim U(n)$, then:

- $\mu_X = \frac{1+n}{2}$,

- $\sigma^2_X = \frac{n^2-1}{12}$.

Podemos amostrar uma matriz de forma arbitrária a partir de uma variável aleatória uniforme discreta como segue.

```
torch.randint(1, n, size=(10, 10))
```

```
tensor([[2, 4, 2, 4, 1, 3, 2, 2, 2, 4],
        [2, 4, 3, 3, 1, 4, 2, 4, 2, 3],
        [1, 2, 1, 2, 2, 3, 2, 1, 1, 3],
        [3, 4, 2, 4, 1, 1, 2, 1, 2, 2],
        [1, 3, 3, 3, 2, 3, 4, 4, 1, 1],
        [4, 2, 2, 4, 1, 3, 1, 3, 2, 4],
        [3, 3, 3, 3, 4, 3, 4, 2, 1, 2],
        [3, 3, 4, 2, 3, 1, 4, 2, 3, 1],
        [3, 4, 3, 1, 4, 1, 2, 2, 1, 3],
        [4, 4, 1, 3, 4, 4, 2, 2, 3, 1]])
```

18.8.3 Uniforme e Contínua

A seguir, vamos discutir a distribuição uniforme contínua. A ideia por trás dessa variável aleatória é que, se aumentarmos n na distribuição uniforme discreta e, em seguida, escaloná-la para caber no intervalo $[a, b]$, abordaremos uma variável aleatória contínua que apenas escolhe um valor arbitrário em $[a, b]$ todos com probabilidade igual. Vamos denotar esta distribuição como

$$X \sim U(a, b). \tag{18.8.5}$$

A função de densidade de probabilidade é

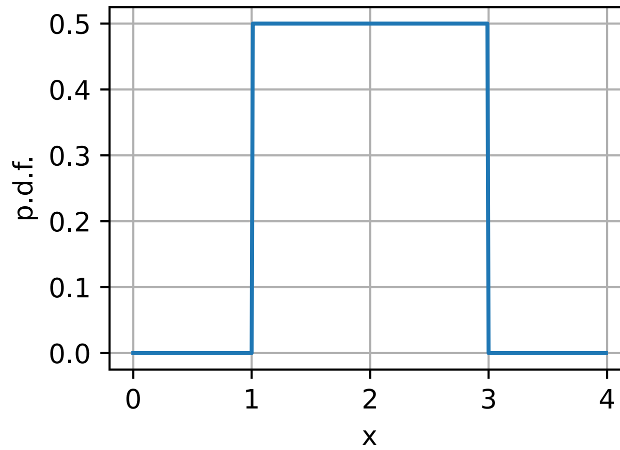
$$p(x) = \begin{cases} \frac{1}{b-a} & x \in [a, b], \\ 0 & x \not\in [a, b].\end{cases} \tag{18.8.6}$$

A função de distribuição cumulativa é

$$F(x) = \begin{cases} 0 & x < a, \\ \frac{x-a}{b-a} & x \in [a, b], \\ 1 & x >= b. \end{cases} \tag{18.8.7}$$

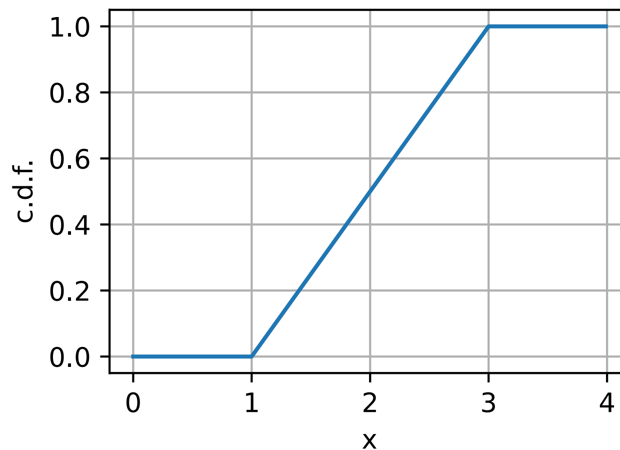
Vamos primeiro representar graficamente a função de densidade de probabilidade (18.8.6).

```
a, b = 1, 3

x = torch.arange(0, 4, 0.01)
p = (x > a).type(torch.float32)*(x < b).type(torch.float32)/(b-a)
d2l.plot(x, p, 'x', 'p.d.f.')
```



Agora, vamos representar graficamente a função de distribuição cumulativa (18.8.7).

```
def F(x):
    return 0 if x < a else 1 if x > b else (x - a) / (b - a)

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



Se $X \sim U(a, b)$, então:

- $\mu_X = \frac{a+b}{2}$,
- $\sigma_X^2 = \frac{(b-a)^2}{12}$.

Podemos amostrar uma matriz de forma arbitrária a partir de uma variável aleatória uniforme da seguinte maneira. Observe que, por padrão, é uma amostra de $U(0, 1)$, portanto, se quisermos um intervalo diferente, precisamos escaloná-lo.

```
(b - a) * torch.rand(10, 10) + a
```

```
tensor([[2.8223, 1.5381, 2.7166, 1.7558, 1.2417, 1.2230, 2.2913, 1.5042, 1.6630,
         2.5570],
```

(continues on next page)

(continued from previous page)

```
        [1.5648, 1.4812, 1.9697, 1.5672, 1.9874, 1.5999, 1.4540, 1.8198, 2.4150,
         1.5929],
        [1.1754, 2.5958, 1.7233, 1.3277, 2.0125, 2.1797, 1.5919, 1.0497, 2.1812,
         1.3366],
        [2.4354, 1.6165, 1.1859, 2.9547, 2.0595, 1.8269, 1.1119, 2.3576, 1.9823,
         2.4681],
        [2.7830, 2.8629, 2.5027, 1.6039, 1.5405, 1.1954, 2.5066, 2.0107, 2.6782,
         1.3883],
        [1.2284, 2.3149, 2.1947, 2.4759, 1.0598, 1.1362, 2.7960, 2.9847, 2.3273,
         1.5196],
        [2.0935, 1.1542, 1.8042, 1.0448, 1.0768, 1.8240, 1.2579, 2.1343, 1.5140,
         1.9848],
        [1.5720, 2.2829, 1.2114, 2.2404, 1.8355, 1.2281, 1.1693, 2.3944, 2.2222,
         2.2738],
        [1.9178, 1.3517, 1.0908, 1.3381, 1.0433, 1.1117, 1.6149, 1.4846, 1.2166,
         2.9854],
        [2.3129, 1.2611, 2.3859, 2.4383, 1.8703, 1.3239, 1.5865, 2.9044, 2.6024,
         1.2154]])
```

18.8.4 Binomial

Deixe-nos tornar as coisas um pouco mais complexas e examinar a variável aleatória *binomial*. Essa variável aleatória se origina da execução de uma sequência de n experimentos independentes, cada um dos quais tem probabilidade p de sucesso, e perguntando quantos sucessos esperamos ver.

Vamos expressar isso matematicamente. Cada experimento é uma variável aleatória independente X_i, onde usaremos 1 para codificar o sucesso e 0 para codificar a falha. Como cada um é um lançamento de moeda independente que é bem-sucedido com a probabilidade p, podemos dizer que $X_i \sim \mathrm{Bernoulli}(p)$. Então, a variável aleatória binomial é

$$X = \sum_{i=1}^n X_i. \tag{18.8.8}$$

Neste caso, vamos escrever

$$X \sim \mathrm{Binomial}(n, p). \tag{18.8.9}$$

Para obter a função de distribuição cumulativa, precisamos observar que obter exatamente k sucessos podem ocorrer em $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ maneiras, cada uma das quais tem uma probabilidade de $p^k(1-p)^{n-k}$ de ocorrer. Assim, a função de distribuição cumulativa é

$$F(x) = \begin{cases} 0 & x < 0, \\ \sum_{m \le k} \binom{n}{m} p^m (1-p)^{n-m} & k \le x < k+1 \text{ with } 0 \le k < n, \\ 1 & x >= n. \end{cases} \tag{18.8.10}$$

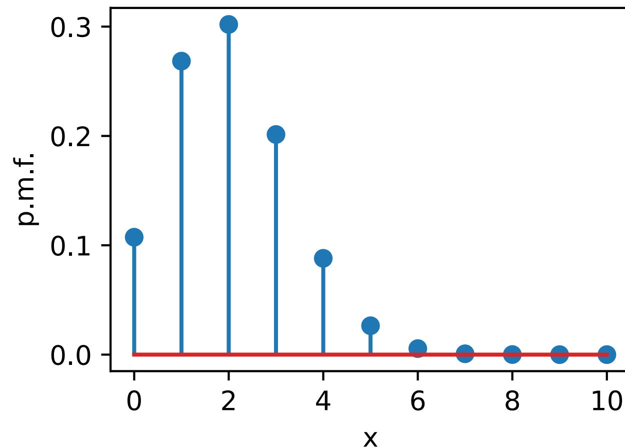
Deixe-nos primeiro representar graficamente a função de massa de probabilidade.

```
n, p = 10, 0.2
```

(continues on next page)

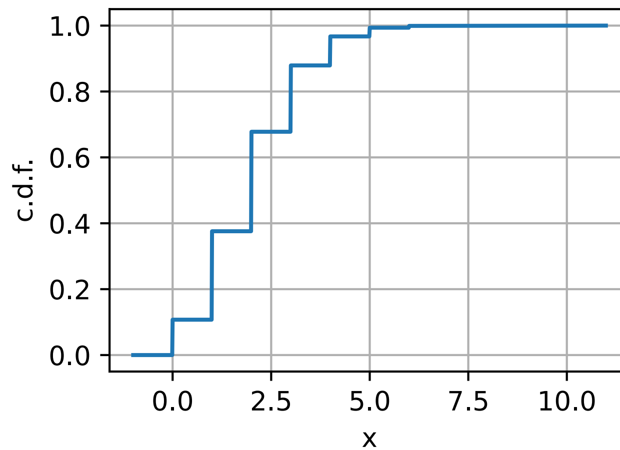
(continued from previous page)

```
# Compute binomial coefficient
def binom(n, k):
    comb = 1
    for i in range(min(k, n - k)):
        comb = comb * (n - i) // (i + 1)
    return comb

pmf = torch.tensor([p**i * (1-p)**(n - i) * binom(n, i) for i in range(n + 1)])

d2l.plt.stem([i for i in range(n + 1)], pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Now, let us plot the cumulative distribution function (18.8.10).

```
x = torch.arange(-1, 11, 0.01)
cmf = torch.cumsum(pmf, dim=0)

def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, torch.tensor([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```



Se $X \sim \text{Binomial}(n, p)$, então:

- $\mu_X = np$,
- $\sigma_X^2 = np(1-p)$.

Isso decorre da linearidade do valor esperado sobre a soma das n variáveis aleatórias de Bernoulli e do fato de que a variância da soma das variáveis aleatórias independentes é a soma das variâncias. Isso pode ser amostrado da seguinte maneira.

```
m = torch.distributions.binomial.Binomial(n, p)
m.sample(sample_shape=(10, 10))
```

```
tensor([[3., 3., 3., 2., 1., 4., 3., 1., 2., 2.],
        [1., 3., 1., 2., 1., 1., 3., 5., 2., 1.],
        [1., 2., 1., 0., 3., 4., 2., 1., 3., 1.],
        [2., 3., 0., 0., 3., 3., 3., 1., 3., 3.],
        [2., 3., 3., 1., 4., 1., 2., 2., 3., 2.],
        [1., 3., 2., 4., 1., 1., 1., 2., 3., 3.],
        [2., 4., 2., 1., 2., 4., 4., 1., 1., 3.],
        [2., 0., 3., 3., 0., 2., 2., 0., 3., 4.],
        [2., 1., 2., 3., 1., 1., 0., 3., 0., 2.],
        [3., 1., 2., 1., 1., 0., 0., 0., 1., 2.]])
```

18.8.5 Poisson

Vamos agora realizar um experimento mental. Estamos parados em um ponto de ônibus e queremos saber quantos ônibus chegarão no próximo minuto. Vamos começar considerando $X^{(1)} \sim \text{Bernoulli}(p)$ que é simplesmente a probabilidade de que um ônibus chegue na janela de um minuto. Para paradas de ônibus longe de um centro urbano, essa pode ser uma boa aproximação. Podemos nunca ver mais de um ônibus por minuto.

Porém, se estivermos em uma área movimentada, é possível ou mesmo provável que cheguem dois ônibus. Podemos modelar isso dividindo nossa variável aleatória em duas partes nos primeiros 30 segundos ou nos segundos 30 segundos. Neste caso, podemos escrever

$$X^{(2)} \sim X_1^{(2)} + X_2^{(2)}, \tag{18.8.11}$$

onde $X^{(2)}$ é a soma total, e $X_i^{(2)} \sim \mathrm{Bernoulli}(p/2)$. A distribuição total é então $X^{(2)} \sim \mathrm{Binomial}(2, p/2)$.

Why stop here? Let us continue to split that minute into n parts. By the same reasoning as above, we see that

$$X^{(n)} \sim \mathrm{Binomial}(n, p/n). \tag{18.8.12}$$

Considere essas variáveis aleatórias. Pela seção anterior, sabemos que (18.8.12) tem média $\mu_{X^{(n)}} = n(p/n) = p$, e variância $\sigma_{X^{(n)}}^2 = n(p/n)(1-(p/n)) = p(1-p/n)$. Se tomarmos $n \to \infty$, podemos ver que esses números se estabilizam em $\mu_{X^{(\infty)}} = p$, e variância $\sigma_{X^{(\infty)}}^2 = p$. Isso indica que *pode haver* alguma variável aleatória que podemos definir neste limite de subdivisão infinito.

Isso não deve ser uma surpresa, já que no mundo real podemos apenas contar o número de chegadas de ônibus, no entanto, é bom ver que nosso modelo matemático está bem definido. Essa discussão pode ser formalizada como a *lei dos eventos raros*.

Seguindo esse raciocínio com cuidado, podemos chegar ao seguinte modelo. Diremos que $X \sim \mathrm{Poisson}(\lambda)$ se for uma variável aleatória que assume os valores $\{0, 1, 2, \ldots\}$ com probabilidade

$$p_k = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{18.8.13}$$

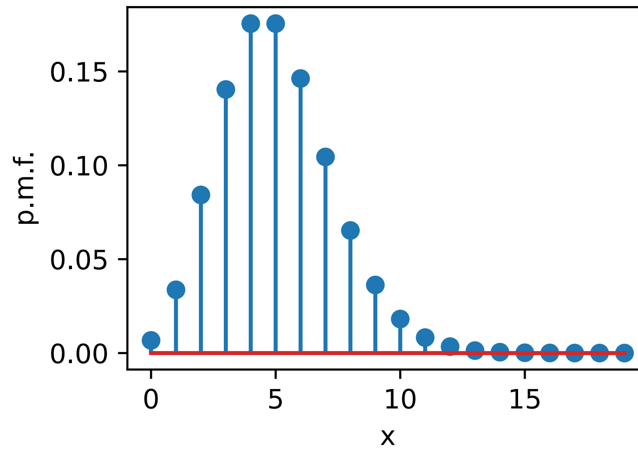
O valor $\lambda > 0$ é conhecido como *taxa* (ou o parâmetro *forma*) e denota o número médio de chegadas que esperamos em uma unidade de tempo.

Podemos somar essa função de massa de probabilidade para obter a função de distribuição cumulativa.

$$F(x) = \begin{cases} 0 & x < 0, \\ e^{-\lambda}\sum_{m=0}^k \frac{\lambda^m}{m!} & k \le x < k+1 \text{ with } 0 \le k. \end{cases} \tag{18.8.14}$$

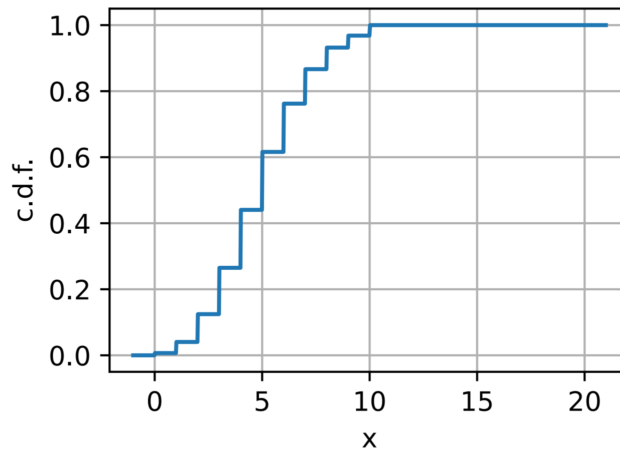
Vamos primeiro representar graficamente a função de massa de probabilidade (18.8.13).

```
lam = 5.0

xs = [i for i in range(20)]
pmf = torch.tensor([torch.exp(torch.tensor(-lam)) * lam**k
                    / factorial(k) for k in xs])

d2l.plt.stem(xs, pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Agora, vamos representar graficamente a função de distribuição cumulativa (18.8.14).

```
x = torch.arange(-1, 21, 0.01)
cmf = torch.cumsum(pmf, dim=0)
def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, torch.tensor([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```



Como vimos acima, as médias e variações são particularmente concisas. Se $X \sim \text{Poisson}(\lambda)$, então:

- $\mu_X = \lambda$,
- $\sigma_X^2 = \lambda$.

Isso pode ser amostrado da seguinte maneira.

```
m = torch.distributions.poisson.Poisson(lam)
m.sample((10, 10))
```

```
tensor([[ 4.,  7.,  5.,  6.,  4.,  7.,  4.,  6.,  3.,  3.],
        [ 5.,  6.,  7.,  9.,  6.,  5.,  3.,  7.,  3.,  6.],
        [10.,  5.,  2.,  2.,  6.,  4.,  5.,  6.,  3.,  4.],
        [ 3.,  8.,  7.,  8.,  3.,  1.,  7.,  3.,  5.,  6.],
        [ 6.,  4.,  6.,  9.,  8.,  7.,  4.,  7.,  3.,  7.],
        [ 5.,  9.,  3.,  3.,  3.,  3.,  2.,  5.,  4.,  8.],
        [ 4.,  2.,  5.,  2.,  4.,  8.,  2.,  9.,  9.,  3.],
        [ 2.,  2.,  5.,  6.,  1.,  5.,  4.,  6.,  6.,  2.],
        [ 8.,  5.,  6.,  4.,  5.,  7.,  9.,  9.,  3.,  6.],
        [ 3.,  2.,  4.,  7.,  3.,  6.,  5.,  2.,  4.,  4.]])
```

18.8.6 Gaussiana

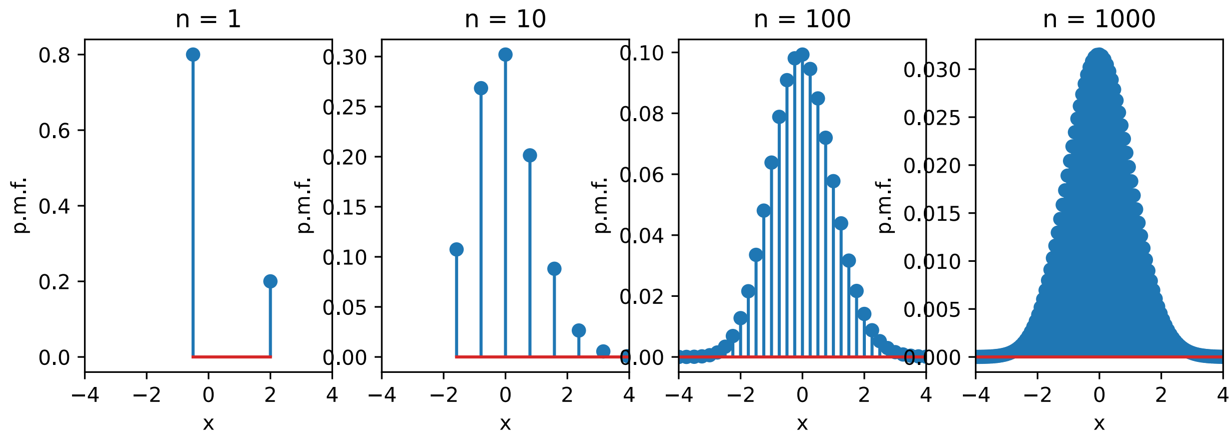
Agora, vamos tentar um experimento diferente, mas relacionado. Digamos que estamos novamente realizando n medidas independentes de Bernoulli(p) X_i. A distribuição da soma delas é $X^{(n)} \sim$ Binomial(n, p). Em vez de considerar um limite à medida que n aumenta e p diminui, vamos corrigir p e enviar $n \to \infty$. Neste caso $\mu_{X^{(n)}} = np \to \infty$ e $\sigma^2_{X^{(n)}} = np(1-p) \to \infty$, portanto, não há razão para pensar que esse limite deva ser bem definido.

No entanto, nem toda esperança está perdida! Vamos apenas fazer com que a média e a variância sejam bem comportadas, definindo

$$Y^{(n)} = \frac{X^{(n)} - \mu_{X^{(n)}}}{\sigma_{X^{(n)}}}. \tag{18.8.15}$$

Pode-se ver que isso tem média zero e variância um e, portanto, é plausível acreditar que convergirá para alguma distribuição limitante. Se traçarmos a aparência dessas distribuições, ficaremos ainda mais convencidos de que funcionará.

```
p = 0.2
ns = [1, 10, 100, 1000]
d2l.plt.figure(figsize=(10, 3))
for i in range(4):
    n = ns[i]
    pmf = torch.tensor([p**i * (1-p)**(n-i) * binom(n, i)
                        for i in range(n + 1)])
    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.stem([(i - n*p)/torch.sqrt(torch.tensor(n*p*(1 - p)))
                  for i in range(n + 1)], pmf,
                 use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')
    d2l.plt.title("n = {}".format(n))
d2l.plt.show()
```



Uma coisa a observar: em comparação com o caso de Poisson, agora estamos dividindo pelo desvio padrão, o que significa que estamos comprimindo os resultados possíveis em áreas cada vez menores. Isso é uma indicação de que nosso limite não será mais discreto, mas sim contínuo.

Uma derivação do que ocorre está além do escopo deste documento, mas o *teorema do limite central* afirma que, como $n \to \infty$, isso resultará na Distribuição Gaussiana (ou as vezes na distribuição normal). Mais explicitamente, para qualquer a, b:

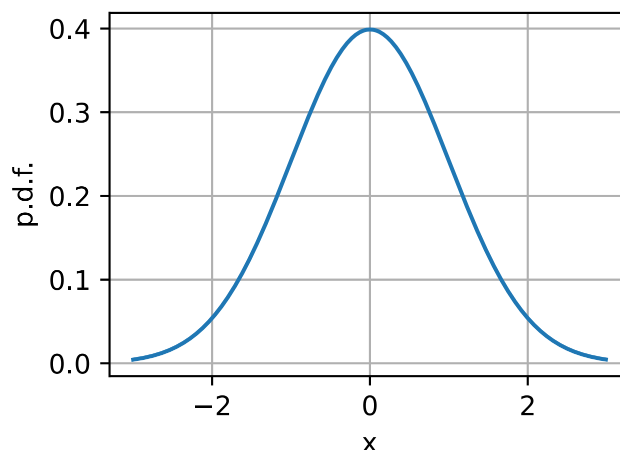
$$\lim_{n \to \infty} P(Y^{(n)} \in [a, b]) = P(\mathcal{N}(0,1) \in [a, b]), \tag{18.8.16}$$

onde dizemos que uma variável aleatória é normalmente distribuída com dada média μ e variância σ^2, escrita $X \sim \mathcal{N}(\mu, \sigma^2)$ se X tem densidade

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{18.8.17}$$

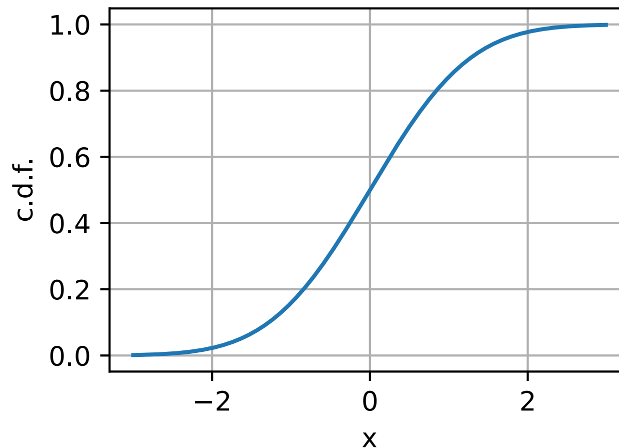
Vamos primeiro representar graficamente a função de densidade de probabilidade (18.8.17).

```
mu, sigma = 0, 1

x = torch.arange(-3, 3, 0.01)
p = 1 / torch.sqrt(2 * torch.pi * sigma**2) * torch.exp(
    -(x - mu)**2 / (2 * sigma**2))

d2l.plot(x, p, 'x', 'p.d.f.')
```



Agora, vamos representar graficamente a função de distribuição cumulativa. Está além do escopo deste apêndice, mas a f.d.c. Gaussiana não tem uma fórmula de forma fechada em termos de funções mais elementares. Usaremos `erf`, que fornece uma maneira de calcular essa integral numericamente.

```
def phi(x):
    return (1.0 + erf((x - mu) / (sigma * torch.sqrt(torch.tensor(2.))))) / 2.0

d2l.plot(x, torch.tensor([phi(y) for y in x.tolist()]), 'x', 'c.d.f.')
```



Os leitores mais atentos reconhecerão alguns desses termos. Na verdade, encontramos essa integral em Section 18.5. Na verdade, precisamos exatamente desse cálculo para ver que esse $p_X(x)$ tem área total um e, portanto, é uma densidade válida.

Nossa escolha de trabalhar com cara ou coroa tornou os cálculos mais curtos, mas nada nessa escolha foi fundamental. De fato, se tomarmos qualquer coleção de variáveis aleatórias independentes distribuídas de forma idêntica X_i, e formar

$$X^{(N)} = \sum_{i=1}^N X_i. \tag{18.8.18}$$

Then

$$\frac{X^{(N)} - \mu_{X^{(N)}}}{\sigma_{X^{(N)}}} \tag{18.8.19}$$

será aproximadamente gaussiana. Existem requisitos adicionais necessários para fazê-la funcionar, mais comumente $E[X^4] < \infty$, mas a filosofia é clara.

O teorema do limite central é a razão pela qual o Gaussiano é fundamental para probabilidade, estatística e aprendizado de máquina. Sempre que podemos dizer que algo que medimos é a soma de muitas pequenas contribuições independentes, podemos supor que o que está sendo medido será próximo de gaussiano.

Existem muitas outras propriedades fascinantes das gaussianas, e gostaríamos de discutir mais uma aqui. A Gaussiana é conhecida como *distribuição de entropia máxima*. Entraremos em entropia mais profundamente em Section 18.11, no entanto, tudo o que precisamos saber neste ponto é que é uma medida de aleatoriedade. Em um sentido matemático rigoroso, podemos pensar no gaussiano como a escolha *mais* aleatória de variável aleatória com média e variância fixas.

Portanto, se sabemos que nossa variável aleatória tem alguma média e variância, a Gaussiana é, de certo modo, a escolha de distribuição mais conservadora que podemos fazer.

Para fechar a seção, vamos lembrar que se $X \sim \mathcal{N}(\mu, \sigma^2)$, então:

- $\mu_X = \mu$,
- $\sigma_X^2 = \sigma^2$.

Podemos obter uma amostra da distribuição gaussiana (ou normal padrão), conforme mostrado abaixo.

```
torch.normal(mu, sigma, size=(10, 10))
```

```
tensor([[ 0.6520,  0.3000, -0.4574, -0.2548,  0.0422, -1.8848, -0.0027,  0.1869,
         -0.8954,  0.4869],
        [-0.2392, -1.8471, -1.2784,  0.1939, -0.3280,  2.1257, -0.5251, -0.2571,
          1.5258,  0.1374],
        [-0.1976,  1.2392, -1.3503, -1.1267,  1.2744,  0.2877, -0.8623,  0.6505,
         -0.0413, -0.6328],
        [ 0.2136, -0.6770,  0.5709,  0.2465, -0.4408,  1.0694,  0.9333, -0.8263,
         -0.3953, -1.5467],
        [-0.0639,  1.3519,  1.1904,  0.4413, -0.2036, -0.8335, -0.6494,  1.0609,
          0.3457, -0.0494],
        [-0.1203, -0.0650,  1.6500, -0.0922, -0.7590, -0.8886, -0.6484,  0.1333,
          0.9615,  0.8912],
        [ 0.9083, -0.0155,  0.3283,  1.1933, -0.5716, -0.0458, -0.0481, -1.4992,
          0.4722, -0.3956],
        [-0.6736, -1.6859,  0.1728, -1.0743,  0.8750,  0.1384, -0.1763,  0.7425,
          0.7856,  2.4783],
        [-1.5720,  0.0452, -0.0454,  0.1253, -0.2715,  0.4259, -1.0297, -0.9537,
          1.1536,  1.6208],
        [-0.7449,  1.9093, -0.1521,  1.6758,  0.7218,  0.7647, -2.6015,  0.0139,
          0.5314, -0.0911]])
```

18.8.7 Família Exponencial

Uma propriedade compartilhada para todas as distribuições listadas acima é que todas pertencem à conhecida como *família exponencial*. A família exponencial é um conjunto de distribuições cuja densidade pode ser expressa no seguinte Formato:

$$p(\mathbf{x} | \boldsymbol{\eta}) = h(\mathbf{x}) \cdot \exp \left(\boldsymbol{\eta}^{\top} \cdot T(\mathbf{x}) - A(\boldsymbol{\eta}) \right) \tag{18.8.20}$$

Como essa definição pode ser um pouco sutil, vamos examiná-la de perto.

Primeiro, $h(\mathbf{x})$ é conhecido como a *medida subjacente* ou a *medida de base*. Isso pode ser visto como uma escolha original da medida que estamos modificando com nosso peso exponencial.

Em segundo lugar, temos o vetor $\boldsymbol{\eta} = (\eta_1, \eta_2, ..., \eta_l) \in \mathbb{R}^l$ chamado de *parâmetros naturais* ou *parâmetros canônicos*. Eles definem como a medida base será modificada. Os parâmetros naturais entram na nova medida tomando o produto escalar desses parâmetros em relação a alguma função $T(\cdot)$ de **:raw-latex:`\mathbf{x}`= (x_1, x_2, ..., x_n) :raw-latex:`**in`:raw-latex:*mathbb{R}`^n$ e exponenciado. O vetor :math:`T(mathbf{x})= (T_1(mathbf{x}), T_2(mathbf{x}), ..., T_l(mathbf{x}))* é chamado de *estatísticas suficientes* para $\boldsymbol{\eta}$. Este nome é usado uma vez que a

informação representada por $T(\mathbf{x})$ é suficiente para calcular a densidade de probabilidade e nenhuma outra informação da amostra $\mathbf{x}$ é requerida.

Terceiro, temos $A(\boldsymbol{\eta})$, que é referido como a *função cumulativa*, que garante que a distribuição acima (18.8.20) integra-se a um, ou seja,

$$A(\boldsymbol{\eta}) = \log\left[\int h(\mathbf{x}) \cdot \exp\left(\boldsymbol{\eta}^\top \cdot T(\mathbf{x})\right) d\mathbf{x}\right]. \tag{18.8.21}$$

Para sermos concretos, consideremos o gaussiano. Supondo que $\mathbf{x}$ seja uma variável univariada, vimos que ela tinha uma densidade de

$$\begin{aligned} p(x|\mu,\sigma) &= \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} \\ &= \frac{1}{\sqrt{2\pi}} \cdot \exp\left\{\frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}x^2 - \left(\frac{1}{2\sigma^2}\mu^2 + \log(\sigma)\right)\right\}. \end{aligned} \tag{18.8.22}$$

Isso corresponde à definição da família exponencial com:

- *medida subjacente*: $h(x) = \frac{1}{\sqrt{2\pi}}$,
- *parâmetros naturais*: $\boldsymbol{\eta} = \begin{bmatrix}\eta_1 \\ \eta_2\end{bmatrix} = \begin{bmatrix}\frac{\mu}{\sigma^2} \\ \frac{1}{2\sigma^2}\end{bmatrix}$,
- *estatísticas suficientes*: $T(x) = \begin{bmatrix}x \\ -x^2\end{bmatrix}$, and
- *função cumulativa*: $A(\boldsymbol{\eta}) = \frac{1}{2\sigma^2}\mu^2 + \log(\sigma) = \frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\log(2\eta_2)$.

É importante notar que a escolha exata de cada um dos termos acima é um pouco arbitrária. Na verdade, a característica importante é que a distribuição pode ser expressa nesta forma, não na forma exata em si.

Como aludimos em Section 3.4.6, uma técnica amplamente utilizada é assumir que a saída final $\mathbf{y}$ segue uma distribuição da família exponencial. A família exponencial é uma comum e poderosa família de distribuições encontradas com frequência no *machine learning*.

18.8.8 Resumo

- Variáveis aleatórias de Bernoulli podem ser usadas para modelar eventos com um resultado sim/não.
- O modelo de distribuições uniformes discretas seleciona a partir de um conjunto finito de possibilidades.
- Distribuições uniformes contínuas selecionam a partir de um intervalo.
- As distribuições binomiais modelam uma série de variáveis aleatórias de Bernoulli e contam o número de sucessos.
- Variáveis aleatórias de Poisson modelam a chegada de eventos raros.
- Variáveis aleatórias gaussianas modelam o resultado da adição de um grande número de variáveis aleatórias independentes.
- Todas as distribuições acima pertencem à família exponencial.

18.8.9 Exercícios

1. Qual é o desvio padrão de uma variável aleatória que é a diferença $X - Y$ de duas variáveis aleatórias binomiais independentes $X, Y \sim \mathrm{Binomial}(16, 1/2)$.
2. Se tomarmos uma variável aleatória de Poisson$X \sim \mathrm{Poisson}(\lambda)$ e considerar $(X - \lambda)/\sqrt{\lambda}$ como $\lambda \to \infty$, podemos mostrar que isso se torna aproximadamente gaussiano. Por que isso faz sentido?
3. Qual é a função de massa de probabilidade para uma soma de duas variáveis aleatórias uniformes discretas em n elementos?

Discussões[207]

18.9 Naive Bayes

Ao longo das seções anteriores, aprendemos sobre a teoria da probabilidade e variáveis aleatórias. Para colocar essa teoria em prática, vamos apresentar o classificador *Naive Bayes*. Isso usa apenas fundamentos probabilísticos para nos permitir realizar a classificação dos dígitos.

Aprender é fazer suposições. Se quisermos classificar um novo exemplo de dados que nunca vimos antes, temos que fazer algumas suposições sobre quais exemplos de dados são semelhantes entre si. O classificador *Naive Bayes*, um algoritmo popular e notavelmente claro, assume que todos os recursos são independentes uns dos outros para simplificar o cálculo. Nesta seção, vamos aplicar este modelo para reconhecer personagens em imagens.

```
%matplotlib inline
import math
import torch
import torchvision
from d2l import torch as d2l

d2l.use_svg_display()
```

18.9.1 Reconhecimento Ótico de Caracteres

MNIST (LeCun et al., 1998) é um dos conjuntos de dados amplamente usados. Ele contém 60.000 imagens para treinamento e 10.000 imagens para validação. Cada imagem contém um dígito escrito à mão de 0 a 9. A tarefa é classificar cada imagem no dígito correspondente.

Gluon fornece uma classe `MNIST` no módulo `data.vision` para recupera automaticamente o conjunto de dados da Internet. Posteriormente, o Gluon usará a cópia local já baixada. Especificamos se estamos solicitando o conjunto de treinamento ou o conjunto de teste definindo o valor do parâmetro `train` para `True` ou `False`, respectivamente. Cada imagem é uma imagem em tons de cinza com largura e altura de 28 com forma (28,28,1). Usamos uma transformação personalizada para remover a última dimensão do canal. Além disso, o conjunto de dados representa cada pixel por um inteiro não assinado de 8 bits. Nós os quantificamos em recursos binários para simplificar o problema.

[207] https://discuss.d2l.ai/t/1098

```
data_transform = torchvision.transforms.Compose(
    [torchvision.transforms.ToTensor()])

mnist_train = torchvision.datasets.MNIST(
    root='./temp', train=True, transform=data_transform, download=True)
mnist_test = torchvision.datasets.MNIST(
    root='./temp', train=False, transform=data_transform, download=True)
```

```
0.4%Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./temp/MNIST/raw/
↪train-images-idx3-ubyte.gz
77.4%
```

Podemos acessar um exemplo particular, que contém a imagem e o rótulo correspondente.

```
image, label = mnist_train[2]
image.shape, label
```

```
(torch.Size([1, 28, 28]), 4)
```

Nosso exemplo, armazenado aqui na variável `imagem`, corresponde a uma imagem com altura e largura de 28 pixels.

```
image.shape, image.dtype
```

```
(torch.Size([1, 28, 28]), torch.float32)
```

Nosso código armazena o rótulo de cada imagem como um escalar. Seu tipo é um número inteiro de 32 bits.

```
label, type(label)
```

```
(4, int)
```

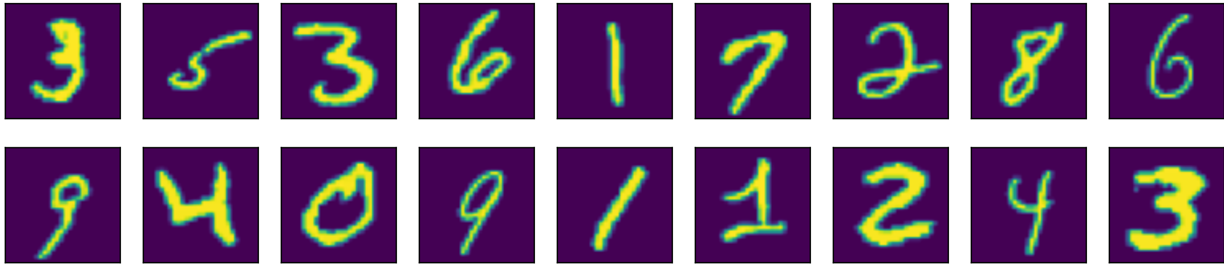
Também podemos acessar vários exemplos ao mesmo tempo.

```
images = torch.stack([mnist_train[i][0] for i in range(10,38)],
                     dim=1).squeeze(0)
labels = torch.tensor([mnist_train[i][1] for i in range(10,38)])
images.shape, labels.shape
```

```
(torch.Size([28, 28, 28]), torch.Size([28]))
```

Vamos visualizar esses exemplos.

```
d2l.show_images(images, 2, 9);
```



18.9.2 O Modelo Probabilístico para Classificação

Em uma tarefa de classificação, mapeamos um exemplo em uma categoria. Aqui, um exemplo é uma imagem em tons de cinza 28×28, e uma categoria é um dígito. (Consulte Section 3.4 para uma explicação mais detalhada.) Uma maneira natural de expressar a tarefa de classificação é por meio da questão probabilística: qual é o rótulo mais provável dado os recursos (ou seja, pixels de imagem)? Denote por $\mathbf{x} \in \mathbb{R}^d$ as características do exemplo e $y \in \mathbb{R}$ o rótulo. Aqui, os recursos são pixels de imagem, onde podemos remodelar uma imagem 2-dimensional para um vetor de modo que $d = 28^2 = 784$, e os rótulos são dígitos. A probabilidade do rótulo dado as características é $p(y \mid \mathbf{x})$. Se pudermos calcular essas probabilidades, que são $p(y \mid \mathbf{x})$ para $y = 0, \ldots, 9$ em nosso exemplo, então o classificador produzirá a previsão $\hat{y}$ dado pela expressão:

$$\hat{y} = \mathrm{argmax} \, p(y \mid \mathbf{x}). \tag{18.9.1}$$

Infelizmente, isso requer que estimemos $p(y \mid \mathbf{x})$ para cada valor de $\mathbf{x} = x_1, ..., x_d$. Imagine que cada recurso pudesse assumir um dos 2 valores. Por exemplo, o recurso $x_1 = 1$ pode significar que a palavra maçã aparece em um determinado documento e $x_1 = 0$ pode significar que não. Se tivéssemos 30 de tais características binárias, isso significaria que precisamos estar preparados para classificar qualquer um dos 2^{30} (mais de 1 bilhão!) de valores possíveis do vetor de entrada $\mathbf{x}$.

Além disso, onde está o aprendizado? Se precisarmos ver todos os exemplos possíveis para prever o rótulo correspondente, não estaremos realmente aprendendo um padrão, mas apenas memorizando o conjunto de dados.

18.9.3 O Classificador Naive Bayes

Felizmente, ao fazer algumas suposições sobre a independência condicional, podemos introduzir algum viés indutivo e construir um modelo capaz de generalizar a partir de uma seleção comparativamente modesta de exemplos de treinamento. Para começar, vamos usar o teorema de Bayes, para expressar o classificador como

$$\hat{y} = \mathrm{argmax}_y \, p(y \mid \mathbf{x}) = \mathrm{argmax}_y \, \frac{p(\mathbf{x} \mid y) p(y)}{p(\mathbf{x})}. \tag{18.9.2}$$

Observe que o denominador é o termo de normalização $p(\mathbf{x})$ que não depende do valor do rótulo y. Como resultado, só precisamos nos preocupar em comparar o numerador em diferentes valores de y. Mesmo que o cálculo do denominador fosse intratável, poderíamos escapar ignorando-o, desde que pudéssemos avaliar o numerador. Felizmente, mesmo se quiséssemos recuperar a constante de normalização, poderíamos. Sempre podemos recuperar o termo de normalização, pois $\sum_y p(y \mid \mathbf{x}) = 1$.

Agora, vamos nos concentrar em $p(\mathbf{x} \mid y)$. Usando a regra da cadeia de probabilidade, podemos expressar o termo $p(\mathbf{x} \mid y)$ como

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot ... \cdot p(x_d \mid x_1, ..., x_{d-1}, y). \tag{18.9.3}$$

Por si só, essa expressão não nos leva mais longe. Ainda devemos estimar cerca de 2^d parâmetros. No entanto, se assumirmos que *as características são condicionalmente independentes umas das outras, dado o rótulo*, então de repente estamos em uma forma muito melhor, pois este termo simplifica para $\prod_i p(x_i \mid y)$, dando-nos o preditor

$$\hat{y} = \mathrm{argmax}_y \prod_{i=1}^{d} p(x_i \mid y)p(y). \tag{18.9.4}$$

Se pudermos estimar $p(x_i = 1 \mid y)$ para cada i e y, e salvar seu valor em $P_{xy}[i, y]$, aqui P_{xy} é uma matriz $d \times n$ com n sendo o número de classes e $y \in \{1, \ldots, n\}$, então também podemos usar isso para estimar $p(x_i = 0 \mid y)$, ou seja,

$$p(x_i = t_i \mid y) = \begin{cases} P_{xy}[i, y] & \text{for } t_i = 1; \\ 1 - P_{xy}[i, y] & \text{for } t_i = 0. \end{cases} \tag{18.9.5}$$

Além disso, estimamos $p(y)$ para cada y e o salvamos em $P_y[y]$, com P_y um vetor de comprimento n. Então, para qualquer novo exemplo $\mathbf{t} = (t_1, t_2, \ldots, t_d)$, poderíamos calcular

$$\begin{aligned} \hat{y} &= \mathrm{argmax}_y \, p(y) \prod_{i=1}^{d} p(x_t = t_i \mid y) \\ &= \mathrm{argmax}_y \, P_y[y] \prod_{i=1}^{d} \, P_{xy}[i, y]^{t_i} \, (1 - P_{xy}[i, y])^{1-t_i} \end{aligned} \tag{18.9.6}$$

para qualquer y. Portanto, nossa suposição de independência condicional levou a complexidade do nosso modelo de uma dependência exponencial no número de características $\mathcal{O}(2^d n)$ para uma dependência linear, que é $\mathcal{O}(dn)$.

18.9.4 Trainamento

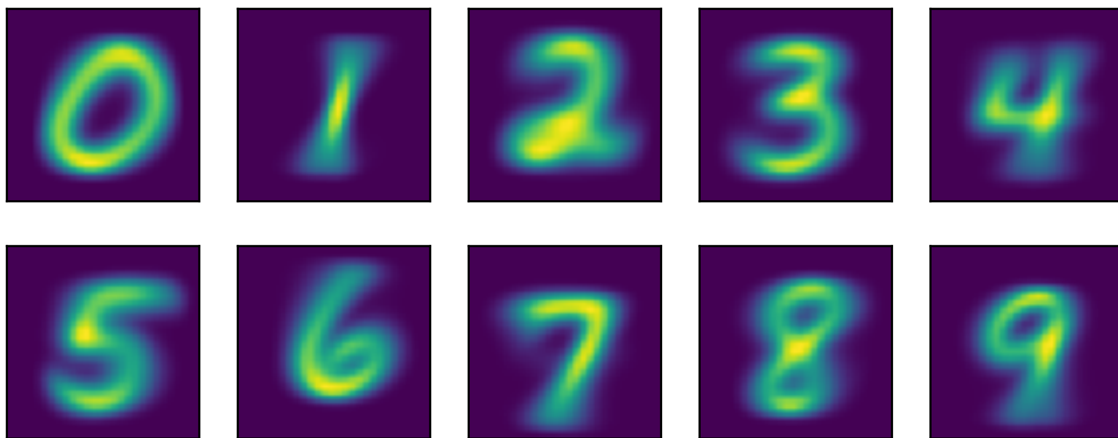
O problema agora é que não conhecemos P_{xy} e P_y. Portanto, precisamos primeiro estimar seus valores dados alguns dados de treinamento. Isso é *treinar* o modelo. Estimar P_y não é muito difícil. Como estamos lidando apenas com classes de 10, podemos contar o número de ocorrências n_y para cada um dos dígitos e dividi-lo pela quantidade total de dados n. Por exemplo, se o dígito 8 ocorre $n_8 = 5,800$ vezes e temos um total de $n = 60,000$ imagens, a estimativa de probabilidade é $p(y = 8) = 0.0967$.

```
X = torch.stack([mnist_train[i][0] for i in range(len(mnist_train))],
                dim=1).squeeze(0)
Y = torch.tensor([mnist_train[i][1] for i in range(len(mnist_train))])

n_y = torch.zeros(10)
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
tensor([0.0987, 0.1124, 0.0993, 0.1022, 0.0974, 0.0904, 0.0986, 0.1044, 0.0975,
        0.0992])
```

Agora vamos para coisas um pouco mais difíceis P_{xy}. Como escolhemos imagens em preto e branco, $p(x_i \mid y)$ denota a probabilidade de que o pixel i seja ativado para a classe y. Assim como antes, podemos ir e contar o número de vezes n_{iy} para que um evento ocorra e dividi-lo pelo número total de ocorrências de y, ou seja, n_y. Mas há algo um pouco preocupante: certos pixels podem nunca ser pretos (por exemplo, para imagens bem cortadas, os pixels dos cantos podem sempre ser brancos). Uma maneira conveniente para os estatísticos lidarem com esse problema é adicionar pseudo contagens a todas as ocorrências. Portanto, em vez de n_{iy}, usamos $n_y + 1$ e em vez de n_y usamos $n_{iy} + 1$. Isso também é chamado de *Suavização de Laplace*. Pode parecer ad-hoc, mas pode ser bem motivado do ponto de vista bayesiano.

```
n_x = torch.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = torch.tensor(X.numpy()[Y.numpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 1).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



Visualizando essas probabilidades de $10 \times 28 \times 28$ (para cada pixel de cada classe), poderíamos obter alguns dígitos de aparência média.

Agora podemos usar (18.9.6) para prever uma nova imagem. Dado $\mathbf{x}$, as seguintes funções calculam $p(\mathbf{x} \mid y)p(y)$ para cada y.

```
def bayes_pred(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(dim=1)  # p(x|y)
    return p_xy * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Isso deu terrivelmente errado! Para descobrir o porquê, vejamos as probabilidades por pixel. Normalmente são números entre $0,001$ e 1. Estamos multiplicando 784 deles. Neste ponto, vale a pena mencionar que estamos calculando esses números em um computador, portanto, com um intervalo fixo para o expoente. O que acontece é que experimentamos *underflow numérico*, ou seja, a multiplicação de todos os números pequenos leva a algo ainda menor até que seja arredondado para zero. Discutimos isso como uma questão teórica em Section 18.7, mas vemos o fenômeno claramente aqui na prática.

Conforme discutido nessa seção, corrigimos isso usando o fato de que $\log ab = \log a + \log b$, ou seja, mudamos para logaritmos de soma. Mesmo se a e b forem números pequenos, os valores de logaritmo devem estar em uma faixa adequada.

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```

Como o logaritmo é uma função crescente, podemos reescrever (18.9.6) como

$$\hat{y} = \text{argmax}_y \ \log P_y[y] + \sum_{i=1}^{d} \Big[t_i \log P_{xy}[x_i, y] + (1 - t_i) \log(1 - P_{xy}[x_i, y])\Big]. \tag{18.9.7}$$

Podemos implementar a seguinte versão estável:

```
log_P_xy = torch.log(P_xy)
log_P_xy_neg = torch.log(1 - P_xy)
log_P_y = torch.log(P_y)

def bayes_pred_stable(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
tensor([-274.1814, -302.0445, -254.1889, -223.6053, -199.4294, -212.9662,
        -298.5672, -119.8299, -223.9705, -169.0555])
```

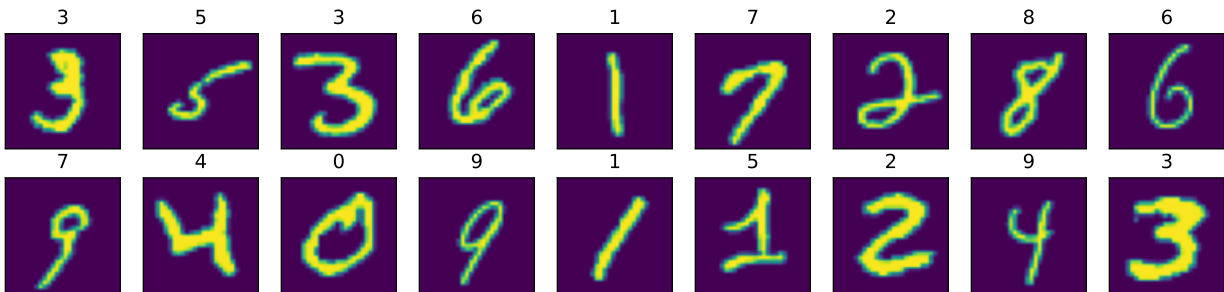
Podemos agora verificar se a previsão está correta.

```
py.argmax(dim=0) == label
```

```
tensor(True)
```

Se agora prevermos alguns exemplos de validação, podemos ver o classificador Bayes funciona muito bem.

```
def predict(X):
    return [bayes_pred_stable(x).argmax(dim=0).type(torch.int32).item()
            for x in X]

X = torch.stack([mnist_train[i][0] for i in range(10,38)], dim=1).squeeze(0)
y = torch.tensor([mnist_train[i][1] for i in range(10,38)])
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



Finalmente, vamos calcular a precisão geral do classificador.

```
X = torch.stack([mnist_train[i][0] for i in range(len(mnist_test))],
                dim=1).squeeze(0)
y = torch.tensor([mnist_train[i][1] for i in range(len(mnist_test))])
preds = torch.tensor(predict(X), dtype=torch.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8364
```

Redes profundas modernas alcançam taxas de erro de menos de $0,01$. O desempenho relativamente baixo é devido às suposições estatísticas incorretas que fizemos em nosso modelo: presumimos que cada pixel é gerado *independentemente*, dependendo apenas do rótulo. Claramente, não é assim que os humanos escrevem dígitos, e essa suposição errada levou à queda de nosso classificador excessivamente ingênuo (Bayes).

18.9.5 Resumo

- Usando a regra de Bayes, um classificador pode ser feito assumindo que todas as características observadas são independentes.
- Este classificador pode ser treinado em um conjunto de dados contando o número de ocorrências de combinações de rótulos e valores de pixel.
- Esse classificador foi o padrão ouro por décadas para tarefas como detecção de spam.

18.9.6 Exercícios

1. Considere o conjunto de dados $[[0, 0], [0, 1], [1, 0], [1, 1]]$ com rótulos dados pelo XOR dos dois elementos $[0, 1, 1, 0]$. Quais são as probabilidades de um classificador Naive Bayes construído neste conjunto de dados. Classifica com sucesso nossos pontos? Se não, quais premissas são violadas?
2. Suponha que não usamos a suavização de Laplace ao estimar as probabilidades e um exemplo de dados chegou no momento do teste que continha um valor nunca observado no treinamento. Qual seria a saída do modelo?
3. O classificador Naive Bayes é um exemplo específico de uma rede Bayesiana, onde a dependência de variáveis aleatórias é codificada com uma estrutura de grafo. Embora a teoria completa esteja além do escopo desta seção (consulte (Koller & Friedman, 2009) para detalhes completos), explique por que permitir a dependência explícita entre as duas variáveis de entrada no modelo XOR permite a criação de um classificador de sucesso .

Discussões[208]

18.10 Estatísticas

Sem dúvida, para ser um profissional de alto nível de *deep learning*, a capacidade de treinar modelos de última geração e alta precisão é crucial. No entanto, muitas vezes não está claro quando as melhorias são significativas ou apenas o resultado de flutuações aleatórias no processo de treinamento. Para ser capaz de discutir a incerteza nos valores estimados, devemos aprender algumas estatísticas.

A referência mais antiga de *estatísticas* pode ser rastreada até um estudioso árabe Al-Kindi no século 9th que deu uma descrição detalhada de como usar estatísticas e análise de frequência para decifrar mensagens criptografadas . Após 800 anos, as estatísticas modernas surgiram na Alemanha em 1700, quando os pesquisadores se concentraram na coleta e análise de dados demográficos e econômicos. Hoje, estatística é o assunto da ciência que diz respeito à coleta, processamento, análise, interpretação e visualização de dados. Além disso, a teoria básica da estatística tem sido amplamente usada na pesquisa na academia, na indústria e no governo.

Mais especificamente, as estatísticas podem ser divididas em *estatísticas descritivas* e *inferência estatística*. O primeiro concentra-se em resumir e ilustrar as características de uma coleção de dados observados, que é chamada de *amostra*. A amostra é retirada de uma *população*, denota o conjunto total de indivíduos, itens ou eventos semelhantes de interesse do nosso experimento. Ao contrário da estatística descritiva, *inferência estatística* deduz ainda mais as características de uma população das *amostras* fornecidas, com base nas suposições de que a distribuição da amostra pode replicar a distribuição da população em algum grau.

Você pode se perguntar: "Qual é a diferença essencial entre *machine learning* e estatística?" Falando fundamentalmente, a estatística enfoca o problema de inferência. Esse tipo de problema inclui a modelagem da relação entre as variáveis, como inferência causal, e o teste da significância estatística dos parâmetros do modelo, como o teste A/B. Em contraste, o aprendizado de máquina enfatiza a realização de previsões precisas, sem programar e compreender explicitamente a funcionalidade de cada parâmetro.

[208] https://discuss.d2l.ai/t/1100

Nesta seção, apresentaremos três tipos de métodos de inferência estatística: avaliação e comparação de estimadores, realização de testes de hipótese e construção de intervalos de confiança. Esses métodos podem nos ajudar a inferir as características de uma determinada população, ou seja, o verdadeiro parâmetro θ. Para resumir, assumimos que o verdadeiro parâmetro θ de uma dada população é um valor escalar. É direto estender para o caso em que θ é um vetor ou tensor, portanto, o omitimos em nossa discussão.

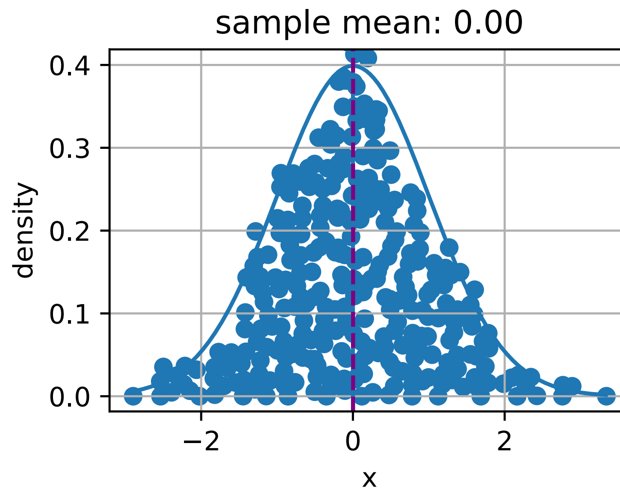
18.10.1 Avaliando e comparando estimadores

Em estatística, um *estimador* é uma função de determinadas amostras usadas para estimar o parâmetro verdadeiro θ. Vamos escrever $\hat{\theta}_n = \hat{f}(x_1, \ldots, x_n)$ para a estimativa de $\theta após observar as amostras $\{x_1, x_2, \ldots, x_n\}$.

Já vimos exemplos simples de estimadores na seção Section 18.7. Se você tiver várias amostras de uma variável aleatória de Bernoulli, então a estimativa de máxima verossimilhança para a probabilidade da variável aleatória ser um pode ser obtida contando o número de unidades observadas e dividindo pelo número total de amostras. Da mesma forma, um exercício pediu que você mostrasse que a estimativa de máxima verossimilhança da média de uma gaussiana dado um número de amostras é dada pelo valor médio de todas as amostras. Esses estimadores quase nunca fornecerão o valor verdadeiro do parâmetro, mas idealmente para um grande número de amostras a estimativa será próxima.

Como exemplo, mostramos abaixo a densidade real de uma variável aleatória gaussiana com média zero e variância um, junto com uma coleção de amostras dessa gaussiana. Construímos a coordenada y de forma que cada ponto fique visível e a relação com a densidade original seja mais clara.

```
import torch
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)) * 2  #define pi in torch

# Sample datapoints and create y coordinate
epsilon = 0.1
torch.manual_seed(8675309)
xs = torch.randn(size=(300,))

ys = torch.tensor(
    [torch.sum(torch.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2))\
               / torch.sqrt(2*torch.pi*epsilon**2)) / len(xs)\
     for i in range(len(xs))])

# Compute true density
xd = torch.arange(torch.min(xs), torch.max(xs), 0.01)
yd = torch.exp(-xd**2/2) / torch.sqrt(2 * torch.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=torch.mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(torch.mean(xs).item()):.2f}')
d2l.plt.show()
```



Pode haver muitas maneiras de calcular um estimador de um parâmetro $\hat{\theta}_n$. Nesta seção, apresentamos três métodos comuns para avaliar e comparar estimadores: o erro quadrático médio, o desvio padrão e o *bias* estatístico.

Erro Quadrático Médio

Talvez a métrica mais simples usada para avaliar estimadores seja o *erro quadrático médio (EQM)* (ou perda l_2) de um estimador pode ser definido como

$$\mathrm{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \tag{18.10.1}$$

Isso nos permite quantificar o desvio médio quadrático do valor verdadeiro. MSE é sempre não negativo. Se você leu Section 3.1, você aoreconhecerá como a função de perda de regressão mais comumente usada. Como medida para avaliar um estimador, quanto mais próximo seu valor de zero, mais próximo o estimador está do verdadeiro parâmetro θ.

Bias Estatístico.

O EQM fornece uma métrica natural, mas podemos facilmente imaginar vários fenômenos diferentes que podem torná-lo grande. Dois fundamentalmente importantes são a flutuação no estimador devido à aleatoriedade no conjunto de dados e o erro sistemático no estimador devido ao procedimento de estimativa.

Primeiro, vamos medir o erro sistemático. Para um estimador $\hat{\theta}_n$, a ilustração matemática de *tendência estatística* pode ser definida como

$$\mathrm{bias}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \tag{18.10.2}$$

Observe que quando $\mathrm{bias}(\hat{\theta}_n) = 0$, a expectativa do estimador **:raw-latex:`\hat{\theta}`**_n$ é igual ao valor verdadeiro do parâmetro. Nesse caso, dizemos que $\hat{\theta}_n$ é um estimador imparcial. Em geral, um estimador não enviesado é melhor do que um estimador enviesado, pois sua expectativa é igual ao parâmetro verdadeiro.

É importante estar ciente, entretanto, que estimadores enviesados são frequentemente usados na prática. Há casos em que estimadores imparciais não existem sem suposições adicionais ou são

intratáveis para calcular. Isso pode parecer uma falha significativa em um estimador, no entanto, a maioria dos estimadores encontrados na prática são pelo menos assintoticamente imparciais, no sentido de que o *bias* tende a zero enquanto o número de amostras disponíveis tende ao infinito: $\lim_{n \rightarrow \infty} \text{bias}(\hat{\theta}_n) = 0$.

Variância e Desvio Padrão

Em segundo lugar, vamos medir a aleatoriedade no estimador. Lembre-se de Section 18.6, o *desvio padrão* (ou *erro padrão*) é definido como a raiz quadrada da variância. Podemos medir o grau de flutuação de um estimador medindo o desvio padrão ou a variância desse estimador.

$$\sigma_{\hat{\theta}_n} = \sqrt{\mathrm{Var} (\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \tag{18.10.3}$$

É importante comparar (18.10.3) com :eqref: eq_mse_est`. Nesta equação, não comparamos com o valor real da população θ mas em vez disso com $E(\hat{\theta}_n)$, a média amostral esperada. Portanto, não estamos medindo o quão longe o estimador tende a estar do valor verdadeiro, mas sim medindo a flutuação do próprio estimador.

The Bias-Variance Trade-off

É intuitivamente claro que esses dois componentes principais contribuem para o erro quadrático médio. O que é um tanto chocante é que podemos mostrar que isso é na verdade uma *decomposição* do erro quadrático médio nessas duas contribuições mais uma terceira. Isso quer dizer que podemos escrever o erro quadrático médio como a soma do quadrado da tendência, a variância e o erro irredutível.

$$\begin{aligned}
\mathrm{MSE} (\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - \theta)^2] \\
 &= E[(\hat{\theta}_n)^2] + E[\theta^2] - 2E[\hat{\theta}_n\theta] \\
 &= \mathrm{Var} [\hat{\theta}_n] + E[\hat{\theta}_n]^2 + \mathrm{Var} [\theta] + E[\theta]^2 - 2E[\hat{\theta}_n]E[\theta] \\
 &= (E[\hat{\theta}_n] - E[\theta])^2 + \mathrm{Var} [\hat{\theta}_n] + \mathrm{Var} [\theta] \\
 &= (E[\hat{\theta}_n - \theta])^2 + \mathrm{Var} [\hat{\theta}_n] + \mathrm{Var} [\theta] \\
 &= (\mathrm{bias} [\hat{\theta}_n])^2 + \mathrm{Var} (\hat{\theta}_n) + \mathrm{Var} [\theta].
\end{aligned} \tag{18.10.4}$$

Nós nos referimos à fórmula acima como *compensação de variação de polarização*. O erro quadrático médio pode ser dividido em três fontes de erro: o erro de alta polarização, o erro de alta variância e o erro irredutível. O erro de polarização é comumente visto em um modelo simples (como um modelo de regressão linear), que não pode extrair relações dimensionais altas entre os recursos e as saídas. Se um modelo sofre de erro de alta polarização, costumamos dizer que ele está *subaproveitado* ou falta de *flexibilidade* conforme apresentado em (Section 4.4). A alta variação geralmente resulta de um modelo muito complexo, que supera os dados de treinamento. Como resultado, um modelo *overfitting* é sensível a pequenas flutuações nos dados. Se um modelo sofre de alta variância, costumamos dizer que está *sobreajuste* e falta de *generalização* conforme apresentado em (Section 4.4). O erro irredutível é o resultado do ruído no próprio θ.

Avaliando estimadores em código

Uma vez que o desvio padrão de um estimador foi implementado simplesmente chamando `a.std()` para um tensor a, vamos ignorá-lo, mas implementar o *bias* estatístico e o erro quadrático médio.

```
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(torch.mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(torch.mean(torch.square(data - true_theta)))
```

Para ilustrar a equação do trade-off *bias*-variância, vamos simular a distribuição normal $\mathcal{N}(\theta, \sigma^2)$ com 10.000 amostras. Aqui, usamos a $\$\theta = 1$ e $\sigma = 4$. Como o estimador é uma função das amostras fornecidas, aqui usamos a média das amostras como um estimador para θ verdadeiros nesta distribuição normal $\mathcal{N}(\theta, \sigma^2)$.

```
theta_true = 1
sigma = 4
sample_len = 10000
samples = torch.normal(theta_true, sigma, size=(sample_len, 1))
theta_est = torch.mean(samples)
theta_est
```

```
tensor(1.0170)
```

Vamos validar a equação de trade-off calculando a soma da polarização quadrada e a variância de nosso estimador. Primeiro, calculamos o EQM de nosso estimador.

```
mse(samples, theta_true)
```

```
tensor(16.0298)
```

Em seguida, calculamos **:raw-latex:`\mathrm{Var}`** (**:raw-latex:`\hat{\theta}`**_n) + [**:raw-latex:`\mathrm{bias}`** (**:raw-latex:`\hat{\theta}`**_n)]^2$ como abaixo. Como você pode ver, os dois valores concordam com a precisão numérica.

```
bias = stat_bias(theta_true, theta_est)
torch.square(samples.std(unbiased=False)) + torch.square(bias)
```

```
tensor(16.0298)
```

18.10.2 Conducting Hypothesis Tests

O tópico mais comumente encontrado na inferência estatística é o teste de hipóteses. Embora o teste de hipóteses tenha sido popularizado no início do século 20, o primeiro uso pode ser rastreado até John Arbuthnot no século XVIII. John rastreou registros de nascimento de 80 anos em Londres e concluiu que mais homens nasciam do que mulheres a cada ano. Em seguida, o teste de significância moderno é a herança de inteligência de Karl Pearson que inventou o valor p e o teste qui-quadrado de Pearson, William Gosset que é o pai da distribuição t de Student e Ronald Fisher que inicializou a hipótese nula e a teste de significância.

Um *teste de hipótese* é uma forma de avaliar algumas evidências contra a declaração padrão sobre uma população. Referimo-nos à declaração padrão como a *hipótese nula* H_0, que tentamos rejeitar usando os dados observados. Aqui, usamos H_0 como ponto de partida para o teste de significância estatística. A *hipótese alternativa* H_A (ou H_1) é uma afirmação que é contrária à hipótese nula. Uma hipótese nula é freqüentemente afirmada em uma forma declarativa que postula uma relação entre variáveis. Deve refletir o resumo o mais explícito possível e ser testável pela teoria estatística.

Imagine que você é um químico. Depois de passar milhares de horas no laboratório, você desenvolve um novo medicamento que pode melhorar drasticamente a habilidade de entender matemática. Para mostrar seu poder mágico, você precisa testá-lo. Naturalmente, você pode precisar de alguns voluntários para tomar o remédio e ver se ele pode ajudá-los a aprender melhor a matemática. Como você começou?

Primeiro, você precisará de dois grupos de voluntários cuidadosamente selecionados aleatoriamente, para que não haja diferença entre sua capacidade de compreensão matemática medida por algumas métricas. Os dois grupos são comumente chamados de grupo de teste e grupo de controle. O *grupo de teste* (ou *grupo de tratamento*) é um grupo de indivíduos que experimentarão o medicamento, enquanto o *grupo de controle* representa o grupo de usuários que são separados como referência, ou seja, configurações de ambiente idênticas, exceto por tomar o medicamento. Desta forma, a influência de todas as variáveis são minimizadas, exceto o impacto da variável independente no tratamento.

Em segundo lugar, após um período de tomar o medicamento, você precisará medir a compreensão matemática dos dois grupos pelas mesmas métricas, como permitir que os voluntários façam os mesmos testes depois de aprender uma nova fórmula matemática. Em seguida, você pode coletar seu desempenho e comparar os resultados. Nesse caso, nossa hipótese nula será que não há diferença entre os dois grupos, e nossa alternativa será que sim.

Isso ainda não é totalmente formal. Existem muitos detalhes nos quais você deve pensar cuidadosamente. Por exemplo, quais são as métricas adequadas para testar sua habilidade de compreensão matemática? Quantos voluntários para o seu teste, para que possa ter a certeza de afirmar a eficácia do seu medicamento? Por quanto tempo você deve executar o teste? Como você decide se há uma diferença entre os dois grupos? Você se preocupa apenas com o desempenho médio ou também com a faixa de variação das pontuações? E assim por diante.

Desta forma, o teste de hipótese fornece uma estrutura para o projeto experimental e raciocínio sobre a certeza nos resultados observados. Se pudermos agora mostrar que a hipótese nula é muito improvável de ser verdadeira, podemos rejeitá-la com confiança.

Para completar a história de como trabalhar com testes de hipóteses, precisamos agora introduzir alguma terminologia adicional e tornar alguns de nossos conceitos formais.

Significância Estatística

A *significância estatística* mede a probabilidade de rejeitar erroneamente a hipótese nula, H_0, quando ela não deveria ser rejeitada, ou seja,

$$\text{statistical significance } = 1 - \alpha = 1 - P(\text{reject } H_0 \mid H_0 \text{ is true}). \tag{18.10.5}$$

Também é conhecido como *erro tipo I* ou *falso positivo*. O α é chamado de *nível de significância* e seu valor comumente usado é 5%, i.e., $1 - \alpha = 95\%$. O nível de significância pode ser explicado como o nível de risco que estamos dispostos a correr, quando rejeitamos uma hipótese nula verdadeira.

`fig_statistically_significance` mostra os valores das observações e a probabilidade de uma dada distribuição normal em um teste de hipótese de duas amostras. Se o exemplo de dados de observação estiver localizado fora do limite de 95%, será uma observação muito improvável sob a hipótese de hipótese nula. Portanto, pode haver algo errado com a hipótese nula e nós a rejeitaremos.

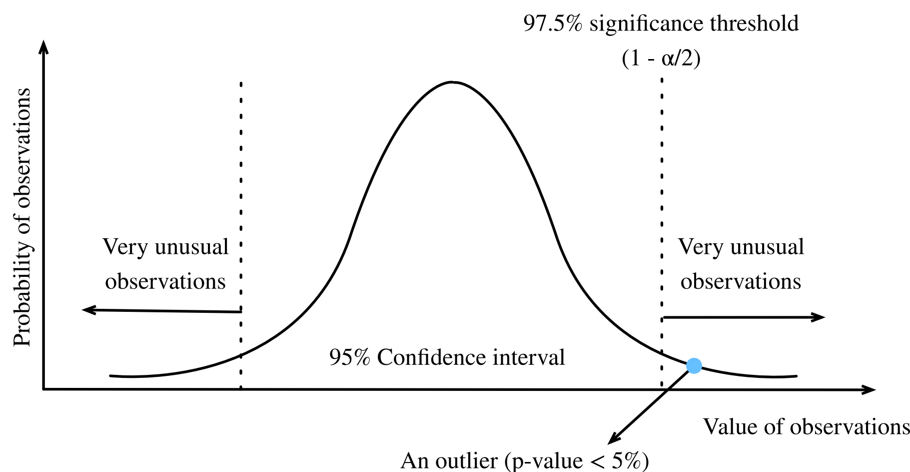


Fig. 18.10.1: Significância Estatística.

Potência Estatística

A *potência estatística* (ou *sensibilidade*) mede a probabilidade de rejeitar a hipótese nula, H_0, quando deveria ser rejeitada, ou seja,

$$\text{statistical power } = 1 - \beta = 1 - P(\text{ fail to reject } H_0 \mid H_0 \text{ is false}). \tag{18.10.6}$$

Lembre-se de que um *erro tipo I* é causado pela rejeição da hipótese nula quando ela é verdadeira, enquanto um *erro tipo II* é resultado da falha em rejeitar a hipótese nula quando ela é falsa. Um erro do tipo II geralmente é denotado como $:raw-latex:*beta `e, portanto, o poder estatístico correspondente é :math:`1-beta*.

Intuitivamente, a potência estatística pode ser interpretada como a probabilidade de nosso teste detectar uma discrepância real de alguma magnitude mínima em um nível de significância estatística desejado. 80% é um limite de poder estatístico comumente usado. Quanto maior o poder estatístico, maior será a probabilidade de detectarmos diferenças verdadeiras.

Um dos usos mais comuns do poder estatístico é determinar o número de amostras necessárias. A probabilidade de você rejeitar a hipótese nula quando ela é falsa depende do grau em que é

falsa (conhecido como *tamanho do efeito*) e do número de amostras que você tem. Como você pode esperar, tamanhos de efeito pequenos exigirão um número muito grande de amostras para serem detectados com alta probabilidade. Embora além do escopo deste breve apêndice para derivar em detalhes, como um exemplo, queremos ser capazes de rejeitar uma hipótese nula de que nossa amostra veio de uma variância zero média gaussiana, e acreditamos que a média de nossa amostra está realmente perto de um , podemos fazer isso com taxas de erro aceitáveis com um tamanho de amostra de apenas 8. No entanto, se pensarmos que a média verdadeira de nossa amostra populacional está próxima de 0.01, então precisaríamos de um tamanho de amostra de quase 80000 para detectar a diferença.

Podemos imaginar o poder como um filtro de água. Nesta analogia, um teste de hipótese de alta potência é como um sistema de filtragem de água de alta qualidade que reduzirá as substâncias nocivas na água tanto quanto possível. Por outro lado, uma discrepância menor é como um filtro de água de baixa qualidade, onde algumas substâncias relativamente pequenas podem escapar facilmente das lacunas. Da mesma forma, se o poder estatístico não for suficientemente alto, o teste pode não detectar a discrepância menor.

Estatística de Teste

Uma *estatística de teste* $T(x)$ é um escalar que resume algumas características dos dados de amostra. O objetivo de definir tal estatística é que ela deve nos permitir distinguir entre diferentes distribuições e conduzir nosso teste de hipótese. Voltando ao nosso exemplo químico, se quisermos mostrar que uma população tem um desempenho melhor do que a outra, pode ser razoável tomar a média como estatística de teste. Diferentes escolhas de estatísticas de teste podem levar a testes estatísticos com poder estatístico drasticamente diferente.

Frequentemente, $T(X)$ (a distribuição da estatística de teste sob nossa hipótese nula) seguirá, pelo menos aproximadamente, uma distribuição de probabilidade comum, como uma distribuição normal quando considerada sob a hipótese nula. Se pudermos derivar explicitamente essa distribuição e, em seguida, medir nossa estatística de teste em nosso conjunto de dados, podemos rejeitar com segurança a hipótese nula se nossa estatística estiver muito fora do intervalo que esperaríamos. Fazer esse quantitativo nos leva à noção de valores p.

Valor p

O valor p (ou o *valor de probabilidade*) é a probabilidade de que $T(X)$ seja pelo menos tão extremo quanto a estatística de teste observada $T(x)$ assumindo que a hipótese nula é *verdadeira*, ou seja,

$$p\text{-value} = P_{H_0}(T(X) \geq T(x)). \tag{18.10.7}$$

Se o valor de p for menor ou igual a um nível de significância estatística predefinido e fixo α, podemos rejeitar a hipótese nula. Caso contrário, concluiremos que faltam evidências para rejeitar a hipótese nula. Para uma dada distribuição populacional, a *região de rejeição* será o intervalo contido de todos os pontos que possuem um valor p menor que o nível de significância estatística α.

Teste unilateral e teste bilateral

Normalmente existem dois tipos de teste de significância: o teste unilateral e o teste bilateral. O *teste unilateral* é aplicável quando a hipótese nula e a hipótese alternativa têm apenas uma direção. Por exemplo, a hipótese nula pode afirmar que o parâmetro verdadeiro θ é menor ou igual a um valor c. A hipótese alternativa seria que θ é maior que c. Ou seja, a região de rejeição está em apenas um lado da distribuição da amostra. Ao contrário do teste unilateral, o *teste bilateral* é aplicável quando a região de rejeição está em ambos os lados da distribuição de amostragem. Um exemplo neste caso pode ter um estado de hipótese nula de que o parâmetro verdadeiro θ é igual a um valor c. A hipótese alternativa seria que θ não é igual a $ c $.

Etapas Gerais de Teste de Hipóteses

Depois de se familiarizar com os conceitos acima, vamos passar pelas etapas gerais do teste de hipóteses.

1. Enuncie a questão e estabeleça uma hipótese nula H_0.
2. Defina o nível de significância estatística α e um poder estatístico $(1 - \beta)$.
3. Obtenha amostras por meio de experimentos. O número de amostras necessárias dependerá do poder estatístico e do tamanho do efeito esperado.
4. Calcule a estatística de teste e o valor p.
5. Tome a decisão de manter ou rejeitar a hipótese nula com base no valor p e no nível de significância estatística α.

Para realizar um teste de hipótese, começamos definindo uma hipótese nula e um nível de risco que estamos dispostos a correr. Em seguida, calculamos a estatística de teste da amostra, tomando um valor extremo da estatística de teste como evidência contra a hipótese nula. Se a estatística de teste cair dentro da região de rejeição, podemos rejeitar a hipótese nula em favor da alternativa.

O teste de hipóteses é aplicável em uma variedade de cenários, como testes clínicos e testes A/B.

18.10.3 Construindo Intervalos de Confiança

Ao estimar o valor de um parâmetro θ, estimadores pontuais como $\hat{\theta}$ são de utilidade limitada, pois não contêm nenhuma noção de incerteza. Em vez disso, seria muito melhor se pudéssemos produzir um intervalo que contivesse o parâmetro verdadeiro θ com alta probabilidade. Se você estivesse interessado em tais ideias um século atrás, então você teria ficado animado ao ler "Esboço de uma teoria de estimativa estatística baseada na teoria clássica da probabilidade" por Jerzy Neyman (Neyman, 1937), que apresentou pela primeira vez o conceito de intervalo de confiança em 1937.

Para ser útil, um intervalo de confiança deve ser o menor possível para um determinado grau de certeza. Vamos ver como derivá-lo.

Definição

Matematicamente, um *intervalo de confiança* para o parâmetro verdadeiro θ é um intervalo C_n calculado a partir dos dados de amostra de modo que

$$P_{\theta}(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \tag{18.10.8}$$

Aqui, $\alpha \in (0, 1)$, e $1 - \alpha$ é chamado de *nível de confiança* ou *cobertura* do intervalo. Este é o mesmo α que o nível de significância discutido acima.

Observe que (18.10.8) é sobre a variável C_n, não sobre o θ fixo. Para enfatizar isso, escrevemos $P_{\theta}(C_n \ni \theta)$ em vez de $P_{\theta}(\theta \in C_n)$.

Interpretação

É muito tentador interpretar um intervalo de confiança de 95% como um intervalo em que você pode ter 95% de certeza de que o parâmetro verdadeiro está, mas infelizmente isso não é verdade. O verdadeiro parâmetro é fixo e é o intervalo que é aleatório. Assim, uma interpretação melhor seria dizer que se você gerasse um grande número de intervalos de confiança por esse procedimento, 95% dos intervalos gerados conteriam o parâmetro verdadeiro.

Isso pode parecer pedante, mas pode ter implicações reais para a interpretação dos resultados. Em particular, podemos satisfazer (18.10.8) construindo intervalos que estamos *quase certos* de que não contêm o valor verdadeiro, contanto que raramente o façamos o suficiente. Encerramos esta seção fornecendo três declarações tentadoras, mas falsas. Uma discussão aprofundada desses pontos pode ser encontrada em:cite:`Morey.Hoekstra.Rouder.ea.2016`.

- ** Falácia 1 **. Intervalos de confiança estreitos significam que podemos estimar o parâmetro com precisão.
- ** Falácia 2 **. Os valores dentro do intervalo de confiança têm mais probabilidade de ser o valor verdadeiro do que aqueles fora do intervalo.
- ** Falácia 3 **. A probabilidade de que um determinado intervalo de confiança de 95% contenha o valor verdadeiro é de 95%.

Basta dizer que os intervalos de confiança são objetos sutis. No entanto, se você mantiver a interpretação clara, eles podem ser ferramentas poderosas.

Um Exemplo Gaussiano

Vamos discutir o exemplo mais clássico, o intervalo de confiança para a média de uma Gaussiana de média e variância desconhecidas. Suponha que coletamos n amostras $\{x_i\}_{i=1}^n$ de nossa gaussiana $\mathcal{N}(\mu, \sigma^2)$ Podemos calcular estimadores para a média e o desvio padrão tomando

$$\hat\mu_n = \frac{1}{n}\sum_{i=1}^n x_i \text{ and } \hat\sigma^2_n = \frac{1}{n-1}\sum_{i=1}^n (x_i - \hat\mu)^2. \tag{18.10.9}$$

Se agora considerarmos a variável aleatória

$$T = \frac{\hat\mu_n - \mu}{\hat\sigma_n/\sqrt{n}}, \tag{18.10.10}$$

obtemos uma variável aleatória seguindo uma distribuição bem *conhecida chamada*distribuição t de Student em* $n-1$ *graus de liberdade*.

Esta distribuição é muito bem estudada, e sabe-se, por exemplo, que como $n \rightarrow \infty$, é aproximadamente uma Gaussiana padrão e, portanto, observando os valores da Gaussiana f.d.c. em uma tabela, podemos concluir que o valor de T está no intervalo $[-1.96, 1.96]$ pelo menos 95% do tempo. Para valores finitos de n, o intervalo precisa ser um pouco maior, mas são bem conhecidos e pré-computados em tabelas.

Assim, podemos concluir que, para grandes n,

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}} \in [-1.96, 1.96]\right) \geq 0.95. \tag{18.10.11}$$

Reorganizando isso multiplicando ambos os lados por $\hat{\sigma}_n/\sqrt{n}$ e, em seguida, adicionando $\hat{\mu}_n$, obtemos

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \geq 0.95. \tag{18.10.12}$$

Assim, sabemos que encontramos nosso intervalo de confiança de $ 95 % $:

$$\left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \tag{18.10.13}$$

É seguro dizer que (18.10.13) é uma das fórmulas mais usadas em estatística. Vamos encerrar nossa discussão sobre estatística implementando-a. Para simplificar, presumimos que estamos no regime assintótico. Valores pequenos de N devem incluir o valor correto de `t_star` obtido tanto programaticamente quanto de uma tabela t.

```
# PyTorch uses Bessel's correction by default, which means the use of ddof=1
# instead of default ddof=0 in numpy. We can use unbiased=False to imitate
# ddof=0.

# Number of samples
N = 1000

# Sample dataset
samples = torch.normal(0, 1, size=(N,))

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = torch.mean(samples)
sigma_hat = samples.std(unbiased=True)
(mu_hat - t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)),\
 mu_hat + t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)))
```

```
(tensor(-0.0568), tensor(0.0704))
```

18.10.4 Resumo

- Estatísticas se concentram em problemas de inferência, enquanto o *deep learning* enfatiza em fazer previsões precisas sem programação e compreensão explícitas.
- Existem três métodos comuns de inferência estatística: avaliação e comparação de estimadores, realização de testes de hipótese e construção de intervalos de confiança.
- Existem três estimadores mais comuns: tendência estatística, desvio padrão e erro quadrático médio.
- Um intervalo de confiança é um intervalo estimado de um parâmetro populacional verdadeiro que podemos construir com base nas amostras.
- O teste de hipóteses é uma forma de avaliar algumas evidências contra a declaração padrão sobre uma população.

18.10.5 Exercícios

1. Seja $X_1, X_2, \ldots, X_n \overset{\text{idfi}}{\sim} \text{Unif}(0, \theta)$, onde "idfi" significa *independente e distribuído de forma idêntica* . Considere os seguintes estimadores de θ:

$$\hat{\theta} = \max\{X_1, X_2, \ldots, X_n\}; \tag{18.10.14}$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n}\sum_{i=1}^{n} X_i. \tag{18.10.15}$$

   - Encontre a tendência estatística, o desvio padrão e o erro quadrático médio de $\hat{\theta}$.
   - Encontre o viés estatístico, o desvio padrão e o erro quadrático médio de $\tilde{\theta}$.
   - Qual estimador é melhor?

2. Para nosso exemplo de químico na introdução, você pode derivar as 5 etapas para conduzir um teste de hipótese bilateral? Dado o nível de significância estatística $\alpha = 0.05$ e a potência estatística $1 - \beta = 0.8$.
3. Execute o código do intervalo de confiança com $N = 2$ e $\alpha = 0.5$ para 100 conjunto de dados gerado independentemente e plote os intervalos resultantes (neste caso `t_star = 1.0`). Você verá vários intervalos muito curtos que estão longe de conter a verdadeira média 0. Isso contradiz a interpretação do intervalo de confiança? Você se sente confortável usando intervalos curtos para indicar estimativas de alta precisão?

Discussões[209]

[209] https://discuss.d2l.ai/t/1102

18.11 Teoria da Informação

O universo está transbordando de informações. A informação fornece uma linguagem comum entre fendas disciplinares: do Soneto de Shakespeare ao artigo de pesquisadores sobre Cornell ArXiv, da impressão de Noite Estrelada de Van Gogh à Sinfonia nº 5 da música de Beethoven, da primeira linguagem de programação Plankalkül à máquina de última geração algoritmos de aprendizagem. Tudo deve seguir as regras da teoria da informação, não importa o formato. Com a teoria da informação, podemos medir e comparar quanta informação está presente em diferentes sinais. Nesta seção, investigaremos os conceitos fundamentais da teoria da informação e as aplicações da teoria da informação no *machine learning*.

Antes de começar, vamos descrever a relação entre o *machine learning* e a teoria da informação. O *machine learning* tem como objetivo extrair sinais interessantes de dados e fazer previsões críticas. Por outro lado, a teoria da informação estuda a codificação, decodificação, transmissão e manipulação de informações. Como resultado, a teoria da informação fornece uma linguagem fundamental para discutir o processamento da informação em sistemas aprendidos por máquina. Por exemplo, muitos aplicativos de *machine learning* usam a perda de entropia cruzada conforme descrito em Section 3.4. Essa perda pode ser derivada diretamente de considerações teóricas da informação.

18.11.1 Informação

Comecemos com a "alma" da teoria da informação: informação. *Informações* podem ser codificadas em qualquer coisa com uma sequência particular de um ou mais formatos de codificação. Suponha que nos encarreguemos de tentar definir uma noção de informação. Qual poderia ser nosso ponto de partida?

Considere o seguinte experimento mental. Temos um amigo com um baralho de cartas. Eles vão embaralhar o baralho, virar algumas cartas e nos contar declarações sobre as cartas. Tentaremos avaliar o conteúdo informativo de cada declaração.

Primeiro, eles viram um cartão e nos dizem: "Vejo um cartão". Isso não nos fornece nenhuma informação. Já tínhamos a certeza de que era esse o caso, por isso esperamos que a informação seja zero.

Em seguida, eles viram um cartão e dizem: "Vejo um coração". Isso nos fornece algumas informações, mas na realidade existem apenas 4 diferentes processos possíveis, cada um igualmente provável, portanto, não estamos surpresos com esse resultado. Esperamos que seja qual for a medida de informação, este evento tenha baixo conteúdo informativo.

Em seguida, eles viram uma carta e dizem: "Este é o 3 de espadas." Esta é mais informações. Na verdade, havia 52 resultados igualmente prováveis, e nosso amigo nos disse qual era. Esta deve ser uma quantidade média de informações.

Vamos levar isso ao extremo lógico. Suponha que finalmente eles virem todas as cartas do baralho e leiam toda a sequência do baralho embaralhado. Existem $52!$ Pedidos diferentes no baralho, novamente todos igualmente prováveis, portanto, precisamos de muitas informações para saber qual é.

Qualquer noção de informação que desenvolvemos deve estar de acordo com esta intuição. De fato, nas próximas seções aprenderemos como calcular que esses eventos têm 0:raw-latex:*text{ bits}`$, :math:`2text{ bits},* 5.7 bits, e 225.6 bits de informações respectivamente.

Se lermos esses experimentos mentais, veremos uma ideia natural. Como ponto de partida, ao invés de nos preocuparmos com o conhecimento, podemos partir da ideia de que a informação representa o grau de surpresa ou a possibilidade abstrata do evento. Por exemplo, se quisermos descrever um evento incomum, precisamos de muitas informações. Para um evento comum, podemos não precisar de muitas informações.

Em 1948, Claude E. Shannon publicou *A Mathematical Theory of Communication* (Shannon, 1948) que estabelece a teoria da informação. Em seu artigo, Shannon introduziu o conceito de entropia de informação pela primeira vez. Começaremos nossa jornada aqui.

Autoinformação

Visto que a informação incorpora a possibilidade abstrata de um evento, como mapeamos a possibilidade para o número de bits? Shannon introduziu a terminologia *bit* como a unidade de informação, que foi originalmente criada por John Tukey. Então, o que é um "bit" e por que o usamos para medir as informações? Historicamente, um transmissor antigo só pode enviar ou receber dois tipos de código: 0 e 1. Na verdade, a codificação binária ainda é de uso comum em todos os computadores digitais modernos. Desta forma, qualquer informação é codificada por uma série de 0 e 1. E, portanto, uma série de dígitos binários de comprimento n contém n bits de informação.

Agora, suponha que para qualquer série de códigos, cada 0 ou 1 ocorra com uma probabilidade de $\frac{1}{2}$. Portanto, um evento X com uma série de códigos de comprimento n, ocorre com uma probabilidade de $$\frac{1}{2^n}$. Ao mesmo tempo, como mencionamos antes, essa série contém n bits de informação. Então, podemos generalizar para uma função matemática que pode transferir a probabilidade p para o número de bits? Shannon deu a resposta definindo *autoinformação*

$$I(X) = -\log_2(p), \tag{18.11.1}$$

como os *bits* de informação que recebemos para este evento X. Observe que sempre usaremos logaritmos de base 2 nesta seção. Para simplificar, o restante desta seção omitirá o subscrito 2 na notação logarítmica, ou seja, $\log(.)$ Sempre se refere a $\log_2(.)$. Por exemplo, o código "0010" tem uma autoinformação

$$I(\text{"0010"}) = -\log(p(\text{"0010"})) = -\log\left(\frac{1}{2^4}\right) = 4 \text{ bits}. \tag{18.11.2}$$

Podemos calcular a autoinformação conforme mostrado abaixo. Antes disso, vamos primeiro importar todos os pacotes necessários nesta seção.

```
import torch
from torch.nn import NLLLoss


def nansum(x):
    # Define nansum, as pytorch doesn't offer it inbuilt.
    return x[~torch.isnan(x)].sum()

def self_information(p):
    return -torch.log2(torch.tensor(p)).item()

self_information(1 / 64)
```

```
6.0
```

18.11.2 Entropia

Como a autoinformação mede apenas a informação de um único evento discreto, precisamos de uma medida mais generalizada para qualquer variável aleatória de distribuição discreta ou contínua.

Motivação para Entropia

Vamos tentar ser específicos sobre o que queremos. Esta será uma declaração informal do que é conhecido como *axiomas da entropia de Shannon*. Acontece que a seguinte coleção de afirmações de senso comum nos força a uma definição única de informação. Uma versão formal desses axiomas, junto com vários outros, pode ser encontrada em (Csiszar, 2008).

1. A informação que ganhamos ao observar uma variável aleatória não depende do que chamamos de elementos, ou da presença de elementos adicionais que têm probabilidade zero.
2. A informação que ganhamos ao observar duas variáveis aleatórias não é mais do que a soma das informações que ganhamos ao observá-las separadamente. Se eles forem independentes, então é exatamente a soma.
3. A informação obtida ao observar (quase) certos eventos é (quase) zero.

Embora provar esse fato esteja além do escopo de nosso texto, é importante saber que isso determina de maneira única a forma que a entropia deve assumir. A única ambiguidade que eles permitem está na escolha das unidades fundamentais, que na maioria das vezes é normalizada fazendo a escolha que vimos antes de que a informação fornecida por um único cara ou coroa é um bit.

Definição

Para qualquer variável aleatória X que segue uma distribuição de probabilidade P com uma função de densidade de probabilidade (pdf) ou uma função de massa de probabilidade (pmf) $p(x)$, medimos a quantidade esperada de informação por meio de *entropia* (ou *entropia de Shannon*)

$$H(X) = -E_{x \sim P}[\log p(x)]. \tag{18.11.3}$$

Para ser específico, se X é discreto, $H(X) = -\sum_i p_i \log p_i$, where $p_i = P(X_i)$.$

Caso contrário, se X for contínuo, também nos referimos à entropia como *entropia diferencial*

$$H(X) = -\int_x p(x) \log p(x) \; dx. \tag{18.11.4}$$

Podemos definir entropia como a seguir.

```
def entropy(p):
    entropy = - p * torch.log2(p)
    # Operator nansum will sum up the non-nan number
    out = nansum(entropy)
    return out

entropy(torch.tensor([0.1, 0.5, 0.1, 0.3]))
```

```
tensor(1.6855)
```

Interpretações

Você pode estar curioso: na definição de entropia (18.11.3), por que usamos uma expectativa de um logaritmo negativo? Aqui estão algumas intuições.

Primeiro, por que usamos uma função *logaritmo* log? Suponha que $p(x) = f_1(x)f_2(x)\ldots, f_n(x)$, onde cada função componente $f_i(x)$ é independente uma da outra. Isso significa que cada $f_i(x)$ contribui de forma independente para a informação total obtida de $p(x)$. Conforme discutido acima, queremos que a fórmula da entropia seja aditiva sobre as variáveis aleatórias independentes. Felizmente, log pode naturalmente transformar um produto de distribuições de probabilidade em uma soma dos termos individuais.

Em seguida, por que usamos um log *negativo*? Intuitivamente, eventos mais frequentes devem conter menos informações do que eventos menos comuns, uma vez que geralmente obtemos mais informações de um caso incomum do que de um caso comum. No entanto, log está aumentando monotonicamente com as probabilidades e, de fato, negativo para todos os valores em $[0, 1]$. Precisamos construir uma relação monotonicamente decrescente entre a probabilidade dos eventos e sua entropia, que idealmente será sempre positiva (pois nada do que observarmos deve nos forçar a esquecer o que conhecemos). Portanto, adicionamos um sinal negativo na frente da função log.

Por último, de onde vem a função *expectation*? Considere uma variável aleatória X. Podemos interpretar a autoinformação ($-\log(p)$) como a quantidade de *surpresa* que temos ao ver um determinado resultado. Na verdade, à medida que a probabilidade se aproxima de zero, a surpresa torna-se infinita. Da mesma forma, podemos interpretar a entropia como a quantidade média de surpresa ao observar X. Por exemplo, imagine que um sistema de caça-níqueis emita símbolos estatísticos independentemente $s_1, \ldots, s_k$ com probabilidades $p_1, \ldots, p_k$ respectivamente. Então, a entropia deste sistema é igual à auto-informação média da observação de cada saída, ou seja,

$$H(S) = \sum_i p_i \cdot I(s_i) = -\sum_i p_i \cdot \log p_i. \tag{18.11.5}$$

Propriedades da Entropia

Pelos exemplos e interpretações acima, podemos derivar as seguintes propriedades de entropia (18.11.3). Aqui, nos referimos a X como um evento e P como a distribuição de probabilidade de X.

- A entropia não é negativa, ou seja, $H(X) \geq 0, \forall X$.

- Se $X \sim P$ com uma f.d.p. ou um f.m.p. $p(x)$, e tentamos estimar P por uma nova distribuição de probabilidade Q com uma f.d.p. ou um f.m.p. $q(x)$, então

$$H(X) = -E_{x \sim P}[\log p(x)] \leq -E_{x \sim P}[\log q(x)], \text{ com igualdade se e somente se } P = Q. \quad (18.11.6)$$

  Alternativamente, $H(X)$ fornece um limite inferior do número médio de bits necessários para codificar símbolos extraídos de P.

- Se $X \sim P$, então x transporta a quantidade máxima de informações se espalhar uniformemente entre todos os resultados possíveis. Especificamente, se a distribuição de probabilidade P é discreta com k-classe $\{p_1, \ldots, p_k\}$, então $H(X) \leq \log(k)$, com igualdade se e somente se $p_i = \frac{1}{k}, \forall i.$$ Se P é uma variável aleatória contínua, então a história se torna muito mais complicada. No entanto, se impormos adicionalmente que P é suportado em um intervalo finito (com todos os valores entre 0 e 1), então $ P $ tem a entropia mais alta se for a distribuição uniforme nesse intervalo.

18.11.3 Informcação Mútua

Anteriormente, definimos entropia de uma única variável aleatória X, e a entropia de um par de variáveis aleatórias (X, Y)? Podemos pensar nessas técnicas como uma tentativa de responder ao seguinte tipo de pergunta: "Quais informações estão contidas em X e Y juntos, comparados a cada um separadamente? Existem informações redundantes ou todas são exclusivas?"

Para a discussão a seguir, sempre usamos (X, Y) como um par de variáveis aleatórias que segue uma distribuição de probabilidade conjunta P com uma f.d.p. ou uma f.m.p. $p_{X,Y}(x, y)$, enquanto X e Y seguem a distribuição de probabilidade $p_X(x)$ e $p_Y(y)$, respectivamente.

Entropia Conjunta

Semelhante à entropia de uma única variável aleatória (18.11.3), definimos a *entropia conjunta* $H(X, Y)$ de um par de variáveis aleatórias (X, Y) como

$$H(X, Y) = -E_{(x,y) \sim P}[\log p_{X,Y}(x, y)]. \quad (18.11.7)$$

Precisamente, por um lado, se (X, Y) é um par de variáveis aleatórias discretas, então

$$H(X, Y) = -\sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \quad (18.11.8)$$

Por outro lado, se (X, Y) é um par de variáveis aleatórias contínuas, então definimos a *entropia conjunta diferencial* como

$$H(X, Y) = -\int_{x,y} p_{X,Y}(x, y) \ \log p_{X,Y}(x, y) \ dx \ dy. \quad (18.11.9)$$

Podemos pensar em (18.11.7) como nos dizendo a aleatoriedade total no par de variáveis aleatórias. Como um par de extremos, se $X = Y$ são duas variáveis aleatórias idênticas, então as informações do par são exatamente as informações de uma e temos $H(X, Y) = H(X) = H(Y)$. No outro extremo, se X e Y são independentes, então $H(X, Y) = H(X) + H(Y)$. Na verdade, sempre teremos que a informação contida em um par de variáveis aleatórias não é menor que a entropia de qualquer uma das variáveis aleatórias e não mais que a soma de ambas.

$$H(X), H(Y) \leq H(X, Y) \leq H(X) + H(Y). \quad (18.11.10)$$

Vamos implementar a entropia conjunta do zero.

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * torch.log2(p_xy)
    # nansum will sum up the non-nan number
    out = nansum(joint_ent)
    return out

joint_entropy(torch.tensor([[0.1, 0.5], [0.1, 0.3]]))
```

```
tensor(1.6855)
```

Observe que este é o mesmo *código* de antes, mas agora o interpretamos de maneira diferente, como trabalhando na distribuição conjunta das duas variáveis aleatórias.

Entropia Condicional

A entropia conjunta definida acima da quantidade de informações contidas em um par de variáveis aleatórias. Isso é útil, mas muitas vezes não é o que nos preocupa. Considere a configuração de aprendizado de máquina. Tomemos X como a variável aleatória (ou vetor de variáveis aleatórias) que descreve os valores de pixel de uma imagem e Y como a variável aleatória que é o rótulo da classe. X deve conter informações substanciais - uma imagem natural é algo complexo. No entanto, as informações contidas em Y uma vez que a imagem foi exibida devem ser baixas. Na verdade, a imagem de um dígito já deve conter as informações sobre qual dígito ele é, a menos que seja ilegível. Assim, para continuar a estender nosso vocabulário da teoria da informação, precisamos ser capazes de raciocinar sobre o conteúdo da informação em uma variável aleatória condicional a outra.

Na teoria da probabilidade, vimos a definição da *probabilidade condicional* para medir a relação entre as variáveis. Agora queremos definir analogamente a *entropia condicional* $H(Y \mid X)$. Podemos escrever isso como

$$H(Y \mid X) = -E_{(x,y)\sim P}[\log p(y \mid x)], \tag{18.11.11}$$

onde $p(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$ é a probabilidade condicional. Especificamente, se (X, Y) é um par de variáveis aleatórias discretas, então

$$H(Y \mid X) = -\sum_x \sum_y p(x,y) \log p(y \mid x). \tag{18.11.12}$$

Se $ (X, Y) $ é um par de variáveis aleatórias contínuas, então a *entropia condicional diferencial* é similarmente definida como

$$H(Y \mid X) = -\int_x \int_y p(x,y) \ \log p(y \mid x) \ dx \ dy. \tag{18.11.13}$$

Agora é natural perguntar, como a *entropia condicional* $H(Y \mid X)$ se relaciona com a entropia $H(X)$ e a entropia conjunta $H(X, Y)$? Usando as definições acima, podemos expressar isso de forma clara:

$$H(Y \mid X) = H(X,Y) - H(X). \tag{18.11.14}$$

Isso tem uma interpretação intuitiva: a informação em Y dada X ($H(Y \mid X)$) é a mesma que a informação em X e Y juntos ($H(X, Y)$) menos as informações já contidas em X. Isso nos dá as informações em Y, que também não são representadas em X.

Agora, vamos implementar a entropia condicional (18.11.11) do zero.

```python
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * torch.log2(p_y_given_x)
    # nansum will sum up the non-nan number
    out = nansum(cond_ent)
    return out

conditional_entropy(torch.tensor([[0.1, 0.5], [0.2, 0.3]]),
                    torch.tensor([0.2, 0.8]))
```

```
tensor(0.8635)
```

Informação mútua

Dada a configuração anterior de variáveis aleatórias (X, Y), você pode se perguntar: "Agora que sabemos quanta informação está contida em Y, mas não em X, podemos igualmente perguntar quanta informação é compartilhada entre X e Y?" A resposta será a * informação mútua * de (X, Y), que escreveremos como $I(X, Y)$.

Em vez de mergulhar direto na definição formal, vamos praticar nossa intuição tentando primeiro derivar uma expressão para a informação mútua inteiramente baseada em termos que construímos antes. Queremos encontrar a informação compartilhada entre duas variáveis aleatórias. Uma maneira de tentar fazer isso é começar com todas as informações contidas em X e Y juntas e, em seguida, retirar as partes que não são compartilhadas. As informações contidas em X e Y juntas são escritas como $H(X, Y)$. Queremos subtrair disso as informações contidas em X, mas não em Y, e as informações contidas em Y, mas não em X. Como vimos na seção anterior, isso é dado por $H(X \mid Y)$ e $H(Y \mid X)$ respectivamente. Assim, temos que a informação mútua deve ser

$$I(X, Y) = H(X, Y) - H(Y \mid X) - H(X \mid Y). \tag{18.11.15}$$

Na verdade, esta é uma definição válida para a informação mútua. Se expandirmos as definições desses termos e combiná-los, um pouco de álgebra mostra que isso é o mesmo que

$$I(X, Y) = E_x E_y \left\{ p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_X(x) p_Y(y)} \right\}. \tag{18.11.16}$$

Podemos resumir todas essas relações na imagem Fig. 18.11.1. É um excelente teste de intuição ver por que as seguintes afirmações também são equivalentes a $I(X, Y)$.

- $H(X) - H(X \mid Y)$
- $H(Y) - H(Y \mid X)$
- $H(X) + H(Y) - H(X, Y)$

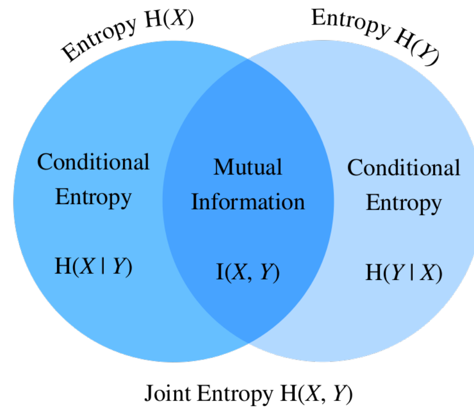


Fig. 18.11.1: Relação da informação mútua com entropia conjunta e entropia condicional.

De muitas maneiras, podemos pensar na informação mútua (18.11.16) como uma extensão de princípio do coeficiente de correlação que vimos em Section 18.6. Isso nos permite pedir não apenas relações lineares entre variáveis, mas também o máximo de informações compartilhadas entre as duas variáveis aleatórias de qualquer tipo.

Agora, vamos implementar informações mútuas do zero.

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * torch.log2(p)
    # Operator nansum will sum up the non-nan number
    out = nansum(mutual)
    return out

mutual_information(torch.tensor([[0.1, 0.5], [0.1, 0.3]]),
                   torch.tensor([0.2, 0.8]), torch.tensor([[0.75, 0.25]]))
```

```
tensor(0.7195)
```

Propriedades da Informação Mútua

Em vez de memorizar a definição de informação mútua (18.11.16), você só precisa ter em mente suas propriedades notáveis:

- A informação mútua é simétrica, ou seja, $I(X,Y) = I(Y,X)$.
- As informações mútuas não são negativas, ou seja, $I(X,Y) \geq 0$.
- $I(X,Y) = 0$ se e somente se X e Y são independentes. Por exemplo, se X e Y são independentes, saber Y não fornece nenhuma informação sobre X e vice-versa, portanto, suas informações mútuas são zero.
- Alternativamente, se X é uma função invertível de Y, então Y e X compartilham todas as informações e

$$I(X,Y) = H(Y) = H(X). \tag{18.11.17}$$

Informações Mútuas Pontuais

Quando trabalhamos com entropia no início deste capítulo, fomos capazes de fornecer uma interpretação de $-\log(p_X(x))$ como *surpresos* com o resultado particular. Podemos dar uma interpretação semelhante ao termo logarítmico nas informações mútuas, que muitas vezes é referido como as *informações mútuas pontuais*:

$$\text{pmi}(x, y) = \log \frac{p_{X,Y}(x, y)}{p_X(x) p_Y(y)}. \tag{18.11.18}$$

Podemos pensar em (18.11.18) medindo o quanto mais ou menos provável a combinação específica de resultados x e y são comparados com o que esperaríamos para resultados aleatórios independentes. Se for grande e positivo, esses dois resultados específicos ocorrem com muito mais frequência do que em comparação com o acaso (*nota*: o denominador é $p_X(x)p_Y(y)$ que é a probabilidade de os dois resultados serem independente), ao passo que, se for grande e negativo, representa os dois resultados que acontecem muito menos do que esperaríamos ao acaso.

Isso nos permite interpretar as informações mútuas (18.11.16) como a quantidade média que ficamos surpresos ao ver dois resultados ocorrendo juntos em comparação com o que esperaríamos se fossem independentes.

Aplicações de Informação Mútua

As informações mútuas podem ser um pouco abstratas em sua definição pura. Como isso se relaciona ao *machine learning*? No processamento de linguagem natural, um dos problemas mais difíceis é a *resolução da ambigüidade*, ou a questão do significado de uma palavra não ser claro no contexto. Por exemplo, recentemente uma manchete no noticiário relatou que "Amazon está pegando fogo". Você pode se perguntar se a empresa Amazon tem um prédio em chamas ou se a floresta amazônica está em chamas.

Nesse caso, informações mútuas podem nos ajudar a resolver essa ambiguidade. Primeiro encontramos o grupo de palavras em que cada uma tem uma informação mútua relativamente grande com a empresa Amazon, como e-commerce, tecnologia e online. Em segundo lugar, encontramos outro grupo de palavras em que cada uma tem uma informação mútua relativamente grande com a floresta amazônica, como chuva, floresta e tropical. Quando precisamos desambiguar "Amazon", podemos comparar qual grupo tem mais ocorrência no contexto da palavra Amazon. Nesse caso, o artigo descreveria a floresta e deixaria o contexto claro.

18.11.4 Divergência de Kullback–Leibler

Conforme discutimos em Section 2.3, podemos usar normas para medir a distância entre dois pontos no espaço de qualquer dimensionalidade. Gostaríamos de poder fazer uma tarefa semelhante com distribuições de probabilidade. Há muitas maneiras de fazer isso, mas a teoria da informação oferece uma das melhores. Agora exploramos a *divergência de Kullback-Leibler (KL)*, que fornece uma maneira de medir se duas distribuições estão próximas ou não.

Definição

Dada uma variável aleatória X que segue a distribuição de probabilidade P com uma f.d.p. ou um f.m.p. $p(x)$, e estimamos P por outra distribuição de probabilidade Q com uma f.d.p. ou uma f.m.o. $q(x)$. Então a divergência *Kullback-Leibler(KL)* (ou *entropia relativa*) entre P e Q é

$$D_{\mathrm{KL}}(P\|Q) = E_{x \sim P} \left[\log \frac{p(x)}{q(x)} \right]. \tag{18.11.19}$$

Tal como acontece com a informação mútua pontua l:eqref:`eq_pmi_def`, podemos fornecer novamente uma interpretação do termo logarítmico: $-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$ será grande e positivo se virmos x com muito mais frequência abaixo de P do que esperaríamos para Q, e grande e negativo se virmos o resultado muito menor do que o esperado. Dessa forma, podemos interpretá-lo como nossa surpresa *relativa* ao observar o resultado, em comparação com o quão surpresos ficaríamos observando-o a partir de nossa distribuição de referência.

Vamos implementar a divergência KL do zero.

```
def kl_divergence(p, q):
    kl = p * torch.log2(p / q)
    out = nansum(kl)
    return out.abs().item()
```

Propriedades da Divergência de KL

Vamos dar uma olhada em algumas propriedades da divergência KL (18.11.19).

- A divergência KL é não simétrica, ou seja, existem P, Q tais que

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P). \tag{18.11.20}$$

- A divergência KL não é negativa, ou seja,

$$D_{\mathrm{KL}}(P\|Q) \geq 0. \tag{18.11.21}$$

  Observe que a igualdade é válida apenas quando $P = Q$.

- Se existe um x tal que $p(x) > 0$ e $q(x) = 0$, então $D_{\mathrm{KL}}(P\|Q) = \infty$.
- Existe uma relação estreita entre divergência KL e informação mútua. Além da relação mostrada em Fig. 18.11.1, $I(X, Y)$ também é numericamente equivalente com os seguintes termos:
  1. $D_{\mathrm{KL}}(P(X, Y) \ \| \ P(X)P(Y))$;
  2. $E_Y \{ D_{\mathrm{KL}}(P(X \mid Y) \ \| \ P(X)) \}$;
  3. $E_X \{ D_{\mathrm{KL}}(P(Y \mid X) \ \| \ P(Y)) \}$.

Para o primeiro termo, interpretamos a informação mútua como a divergência KL entre $P(X, Y)$ e o produto de $P(X)$ e $P(Y)$ e, portanto, é uma medida de quão diferente é a junta distribuição é da distribuição se eles fossem independentes. Para o segundo termo, a informação mútua nos diz a redução média na incerteza sobre Y que resulta do aprendizado do valor da distribuição de X. Semelhante ao terceiro mandato.

Exemplo

Vejamos um exemplo de brinquedo para ver a não simetria explicitamente.

Primeiro, vamos gerar e classificar três tensores de comprimento 10.000: um tensor objetivo p que segue uma distribuição normal $N(0, 1)$, e dois tensores candidatos q_1 e q_2 que seguem distribuições normais $N(-1, 1)$ e $N(1, 1)$ respectivamente.

```
torch.manual_seed(1)

tensor_len = 10000
p = torch.normal(0, 1, (tensor_len, ))
q1 = torch.normal(-1, 1, (tensor_len, ))
q2 = torch.normal(1, 1, (tensor_len, ))

p = torch.sort(p)[0]
q1 = torch.sort(q1)[0]
q2 = torch.sort(q2)[0]
```

Como q_1 e q_2 são simétricos em relação ao eixo y (ou seja, $x = 0$), esperamos um valor semelhante de divergência KL entre $D_{\mathrm{KL}}(p\|q_1)$ e $D_{\mathrm{KL}}(p\|q_2)$. Como você pode ver abaixo, há apenas menos de 3% de desconto entre $D_{\mathrm{KL}}(p\|q_1)$ e $D_{\mathrm{KL}}(p\|q_2)$.

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100

kl_pq1, kl_pq2, similar_percentage
```

```
(8582.0341796875, 8828.3095703125, 2.8290698237936858)
```

Em contraste, você pode descobrir que $D_{\mathrm{KL}}(q_2\|p)$ e $D_{\mathrm{KL}}(p\|q_2)$ estão muito desviados, com cerca de 40% de desconto como mostrado abaixo.

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
(14130.125, 46.18621024399691)
```

18.11.5 Entropia Cruzada

Se você está curioso sobre as aplicações da teoria da informação no aprendizado profundo, aqui está um exemplo rápido. Definimos a distribuição verdadeira P com distribuição de probabilidade $p(x)$, e a distribuição estimada Q com distribuição de probabilidade $q(x)$, e as usaremos no restante desta seção.

Digamos que precisamos resolver um problema de classificação binária com base em n exemplos de dados dados $\{x_1, \ldots, x_n\}$. Suponha que codifiquemos 1 e 0 como o rótulo de classe positivo e negativo y_i respectivamente, e nossa rede neural seja parametrizada por θ. Se quisermos encontrar o melhor θ de forma que $\hat{y}_i = p_\theta(y_i \mid x_i)$, é natural aplicar a abordagem de log-*likelihood*

máxima como foi visto em Section 18.7. Para ser específico, para rótulos verdadeiros y_i e previsões $\hat{y}_i = p_\theta(y_i \mid x_i)$, a probabilidade de ser classificado como positivo é $\pi_i = p_\theta(y_i = 1 \mid x_i)$. Portanto, a função de log-*likelihood* seria

$$\begin{aligned}
l(\theta) &= \log L(\theta) \\
&= \log \prod_{i=1}^{n} \pi_i^{y_i} (1 - \pi_i)^{1 - y_i} \\
&= \sum_{i=1}^{n} y_i \log(\pi_i) + (1 - y_i) \log (1 - \pi_i).
\end{aligned} \tag{18.11.22}$$

Maximizar a função log-*likelihood* $l(\theta)$é idêntico a minimizar $-l(\theta)$ e, portanto, podemos encontrar o melhor θ aqui. Para generalizar a perda acima para quaisquer distribuições, também chamamos $-l(\theta)$ a *perda de entropia cruzada*$\mathrm{CE}(y, \hat{y})$, onde y segue a verdadeira distribuição P e $\hat{y}$ segue a distribuição estimada Q.

Tudo isso foi derivado trabalhando do ponto de vista de máxima verossimilhança. No entanto, se olharmos com atenção, podemos ver que termos como $\log(\pi_i)$ entraram em nosso cálculo, o que é uma indicação sólida de que podemos entender a expressão de um ponto de vista teórico da informação.

Definição formal

Como a divergência KL, para uma variável aleatória X, também podemos medir a divergência entre a distribuição de estimativa Q e a distribuição verdadeira P via *entropia cruzada*,

$$\mathrm{CE}(P, Q) = - E_{x \sim P} [\log(q(x))]. \tag{18.11.23}$$

Ao usar as propriedades da entropia discutidas acima, também podemos interpretá-la como a soma da entropia $H(P)$ e a divergência KL entre P e Q, ou seja,

$$\mathrm{CE} (P, Q) = H(P) + D_{\mathrm{KL}}(P\|Q). \tag{18.11.24}$$

Podemos implementar a perda de entropia cruzada conforme abaixo.

```
def cross_entropy(y_hat, y):
    ce = -torch.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

Agora defina dois tensores para os rótulos e previsões e calcule a perda de entropia cruzada deles.

```
labels = torch.tensor([0, 2])
preds = torch.tensor([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
tensor(0.9486)
```

Propriedades

Como mencionado no início desta seção, entropia cruzada (18.11.23) pode ser usada para definir uma função de perda no problema de otimização. Acontece que os seguintes são equivalentes:

1. Maximizar a probabilidade preditiva de Q para a distribuição P, (ou seja, $E_{x\sim P}[\log(q(x))]$);
2. Minimizar a entropia cruzada $\mathrm{CE}(P, Q)$;
3. Minimizar a divergência KL $D_{\mathrm{KL}}(P\|Q)$.

A definição de entropia cruzada prova indiretamente a relação equivalente entre o objetivo 2 e o objetivo 3, desde que a entropia dos dados verdadeiros $H(P)$ seja constante.

Entropia Cruzada como Função Objetiva da Classificação Multiclasse

Se mergulharmos profundamente na função objetivo de classificação com perda de entropia cruzada CE, descobriremos que minimizar CE é equivalente a maximizar a função log-*likelihood* L.

Para começar, suponha que recebamos um conjunto de dados com n exemplos e ele possa ser classificado em k-classes. Para cada exemplo de dados i, representamos qualquer rótulo k-class $\mathbf{y}_i = (y_{i1}, \ldots, y_{ik})$ por *codificação one-hot*. Para ser mais específico, se o exemplo i pertence à classe j, definimos a j-ésima entrada como 1 e todos os outros componentes como 0, ou seja,

$$y_{ij} = \begin{cases}1 & j \in J; \\ 0 & \text{otherwise.}\end{cases} \tag{18.11.25}$$

Por exemplo, se um problema de classificação multiclasse contém três classes A, B e C, os rótulos $\mathbf{y}_i$ podem ser codificados em $\{A: (1, 0, 0); B: (0, 1, 0); C: (0, 0, 1)\}$.

Suponha que nossa rede neural seja parametrizada por θ. Para vetores de rótulos verdadeiros $\mathbf{y}_i$ e previsões

$$\hat{\mathbf{y}}_i = p_{\theta}(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^k y_{ij} p_{\theta}(y_{ij} \mid \mathbf{x}_i). \tag{18.11.26}$$

Portanto, a *perda de entropia cruzada* seria

$$\mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{i=1}^n \mathbf{y}_i \log \hat{\mathbf{y}}_i = - \sum_{i=1}^n \sum_{j=1}^k y_{ij} \log p_{\theta}(y_{ij} \mid \mathbf{x}_i). \tag{18.11.27}$$

Por outro lado, também podemos abordar o problema por meio da estimativa de máxima verossimilhança. Para começar, vamos apresentar rapidamente uma distribuição multinoulli de classe k. É uma extensão da distribuição Bernoulli de classe binária para multiclasse. Se uma variável aleatória $\mathbf{z} = (z_1, \ldots, z_k)$ segue uma distribuição k-class *multinoulli* com probabilidades $\mathbf{p} = (p_1, \ldots, p_k)$, ou seja,

$$p(\mathbf{z}) = p(z_1, \ldots, z_k) = \mathrm{Multi}(p_1, \ldots, p_k), \text{ onde } \sum_{i=1}^k p_i = 1, \tag{18.11.28}$$

então a função de massa de probabilidade conjunta (f.m.p.) de $\mathbf{z}$ é

$$\mathbf{p}^\mathbf{z} = \prod_{j=1}^k p_j^{z_j}. \tag{18.11.29}$$

Pode-se ver que o rótulo de cada exemplo de dados, $\mathbf{y}_i$, segue uma distribuição multinoulli de k-classes com probabilidades $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$. Portanto, a junta f.m.p. de cada exemplo de dados $\mathbf{y}_i$ é $\boldsymbol{\pi}^{\mathbf{y}_i} = \prod_{j=1}^k \pi_j^{y_{ij}}$. Portanto, a função de log-*likelihood* seria

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^n \boldsymbol{\pi}^{\mathbf{y}_i} = \log \prod_{i=1}^n \prod_{j=1}^k \pi_j^{y_{ij}} = \sum_{i=1}^n \sum_{j=1}^k y_{ij} \log \pi_j. \tag{18.11.30}$$

Já que na estimativa de máxima verossimilhança, maximizamos a função objetivo $l(\theta)$ tendo $\pi_j = p_\theta(y_{ij} \mid \mathbf{x}_i)$. Portanto, para qualquer classificação multiclasse, maximizar a função log-verossimilhança $l(\theta)$ acima é equivalente a minimizar a perda de CE $\mathrm{CE}(y, \hat{y})$.

Para testar a prova acima, vamos aplicar a medida integrada `NegativeLogLikelihood`. Usando os mesmos `rótulos` e `preds` do exemplo anterior, obteremos a mesma perda numérica do exemplo anterior até a casa decimal 5.

```
# Implementation of CrossEntropy loss in pytorch combines nn.LogSoftmax() and
# nn.NLLLoss()
nll_loss = NLLLoss()
loss = nll_loss(torch.log(preds), labels)
loss
```

```
tensor(0.9486)
```

18.11.6 Resumo

- A teoria da informação é um campo de estudo sobre codificação, decodificação, transmissão e manipulação de informações.
- Entropia é a unidade para medir quanta informação é apresentada em diferentes sinais.
- A divergência KL também pode medir a divergência entre duas distribuições.
- A entropia cruzada pode ser vista como uma função objetivo da classificação multiclasse. Minimizar a perda de entropia cruzada é equivalente a maximizar a função de log-*likelihood*.

18.11.7 Exercícios

1. Verifique se os exemplos de cartas da primeira seção realmente possuem a entropia reivindicada.
2. Mostre que a divergência KL $D(p\|q)$ é não negativa para todas as distribuições p e q. Dica: use a desigualdade de Jensen, ou seja, use o fato de que $-\log x$ é uma função convexa.
3. Vamos calcular a entropia de algumas fontes de dados:
    - Suponha que você esteja observando a saída gerada por um macaco em uma máquina de escrever. O macaco pressiona qualquer uma das 44 teclas da máquina de escrever aleatoriamente (você pode assumir que ele ainda não descobriu nenhuma tecla especial ou a tecla shift). Quantos bits de aleatoriedade por personagem você observa?
    - Por estar infeliz com o macaco, você o substituiu por um compositor bêbado. É capaz de gerar palavras, embora não de forma coerente. Em vez disso, ele escolhe uma

palavra aleatória de um vocabulário de 2.000 palavras. Vamos supor que o comprimento médio de uma palavra seja de $4, 5$ letras em inglês. Quantos bits de aleatoriedade por personagem você observa agora?

- Ainda insatisfeito com o resultado, você substitui o compositor por um modelo de linguagem de alta qualidade. O modelo de linguagem pode atualmente obter uma perplexidade tão baixa quanto 15 pontos por palavra. O caractere *perplexidade* de um modelo de linguagem é definido como o inverso da média geométrica de um conjunto de probabilidades, cada probabilidade corresponde a um caractere da palavra. Para ser específico, se o comprimento de uma determinada palavra é l, então $\mathrm{PPL}(\text{word}) = \left[\prod_i p(\text{character}_i)\right]^{-\frac{1}{l}} = \exp\left[-\frac{1}{l}\sum_i \log p(\text{character}_i)\right]$. Suponha que a palavra de teste tem 4,5 letras, quantos bits de aleatoriedade por caractere você observa agora?

4. Explique intuitivamente porque $I(X, Y) = H(X) - H(X|Y)$. Então, mostre que isso é verdade expressando ambos os lados como uma expectativa com relação à distribuição conjunta.
5. Qual é a Divergência KL entre as duas distribuições Gaussianas $\mathcal{N}(\mu_1, \sigma_1^2)$ e $\mathcal{N}(\mu_2, \sigma_2^2)$?

Discussões[210]

[210] https://discuss.d2l.ai/t/1104

19 Apêndice: Ferramentas para Deep Learning

Neste capítulo, vamos guiá-lo pelas principais ferramentas para aprendizado profundo, desde a introdução do Jupyter notebook em Section 19.1 até capacitar modelos de treinamento na nuvem, como Amazon SageMaker em Section 19.2, Amazon EC2 em Section 19.3 e Google Colab em Section 19.4. Além disso, se você gostaria de comprar suas próprias GPUs, também anotamos algumas sugestões práticas em Section 19.5. Se você está interessado em ser um contribuidor deste livro, você pode seguir as instruções em Section 19.6.

19.1 Usando Jupyter

Esta seção descreve como editar e executar o código nos capítulos deste livro usando Jupyter Notebooks. Certifique-se de ter o Jupyter instalado e baixado o código conforme descrito em *Instalação* (page 9). Se você quiser saber mais sobre o Jupyter, consulte o excelente tutorial em Documentação[211].

19.1.1 Editando e executando o código localmente

Suponha que o caminho local do código do livro seja "xx/yy/d2l-en/". Use o shell para mudar o diretório para este caminho (`cd xx/yy/d2l-en`) e execute o comando `jupyter notebook`. Se o seu navegador não fizer isso automaticamente, abra http://localhost:8888 e você verá a interface do Jupyter e todas as pastas contendo o código do livro, conforme mostrado em Fig. 19.1.1.

[211] https://jupyter.readthedocs.io/en/latest/

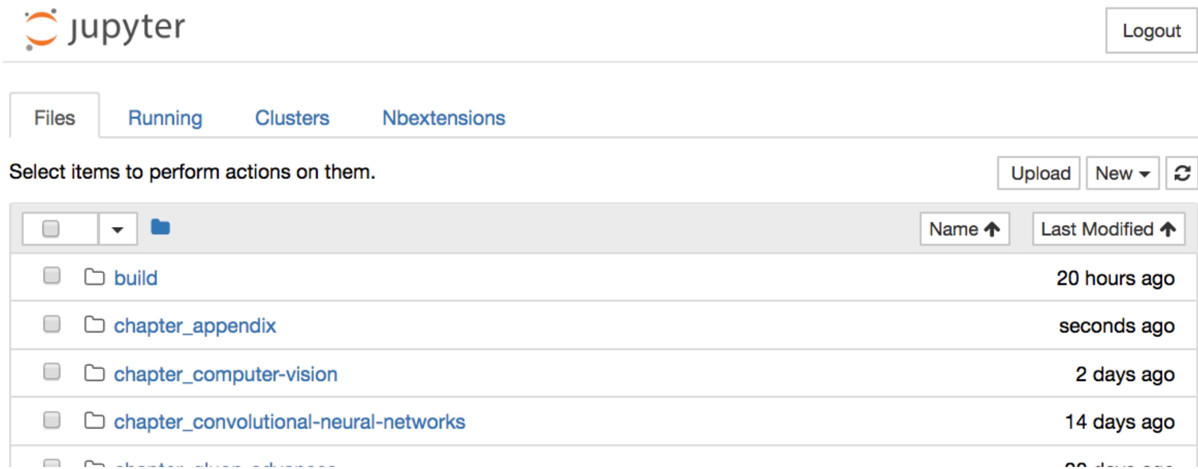


Fig. 19.1.1: As pastas que contêm o código neste livro.

Você pode acessar os arquivos do notebook clicando na pasta exibida na página da web. Eles geralmente têm o sufixo ".ipynb". Para fins de brevidade, criamos um arquivo temporário "test.ipynb". O conteúdo exibido após você clicar é mostrado em Fig. 19.1.2. Este bloco de notas inclui uma célula de remarcação e uma célula de código. O conteúdo da célula de redução inclui "Este é um título" e "Este é um texto". A célula de código contém duas linhas de código Python.

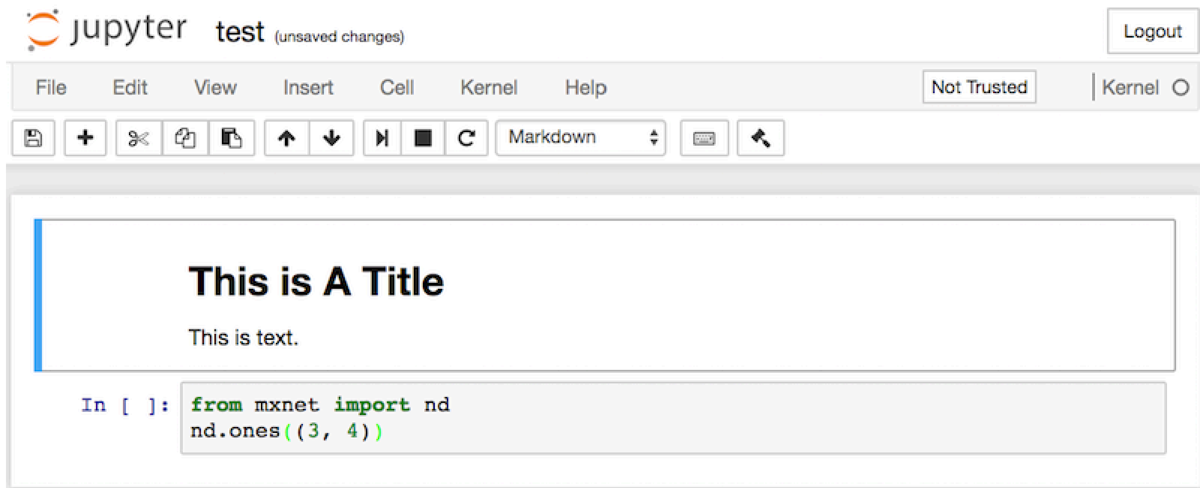


Fig. 19.1.2: Markdown e células de código no arquivo "text.ipynb".

Clique duas vezes na célula de redução para entrar no modo de edição. Adicione uma nova string de texto "Olá, mundo". no final da célula, conforme mostrado em Fig. 19.1.3.

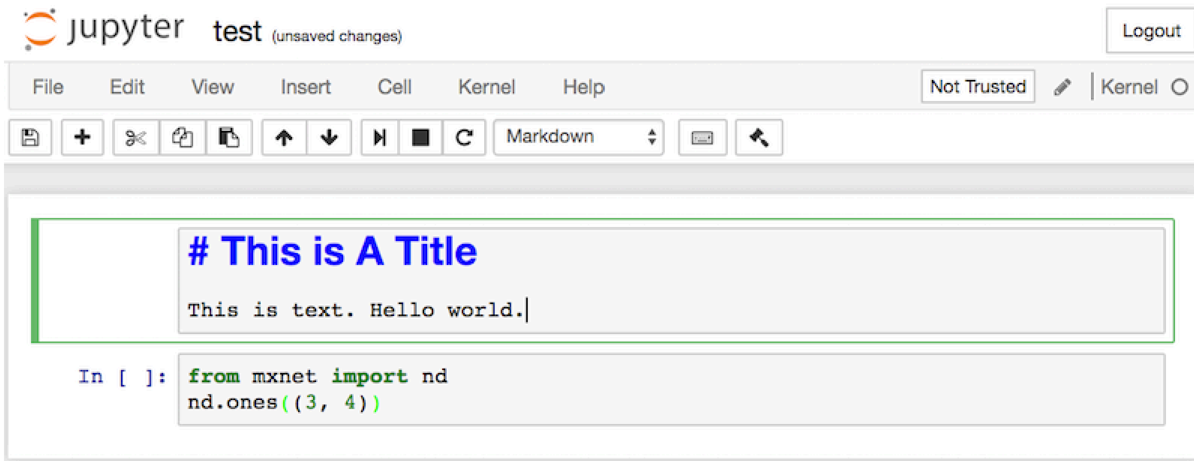


Fig. 19.1.3: Edite a célula de redução.

Conforme mostrado em Fig. 19.1.4, clique em "Cell" → "Run Cells"na barra de menu para executar a célula editada.

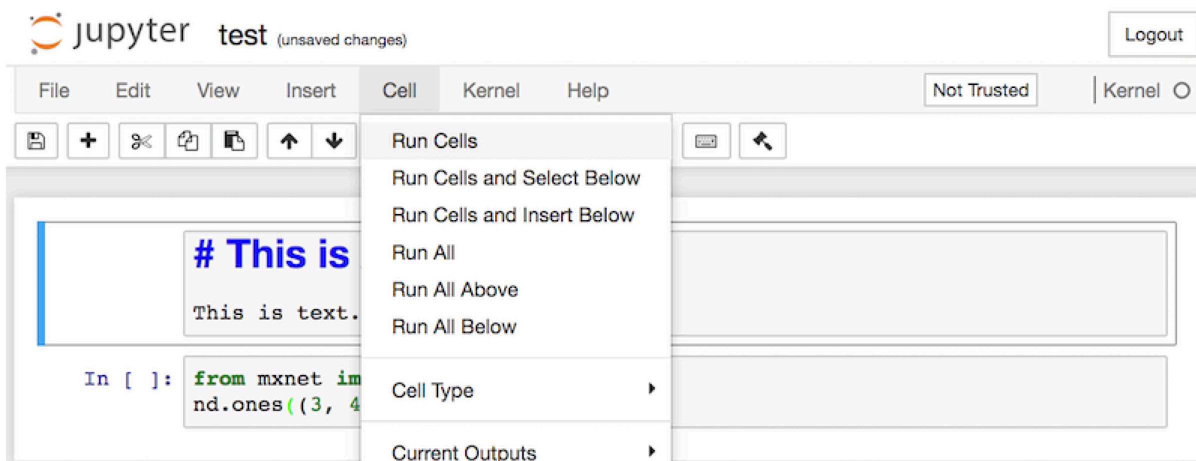


Fig. 19.1.4: Execute a celula.

Após a execução, a célula de redução é mostrada em Fig. 19.1.5.

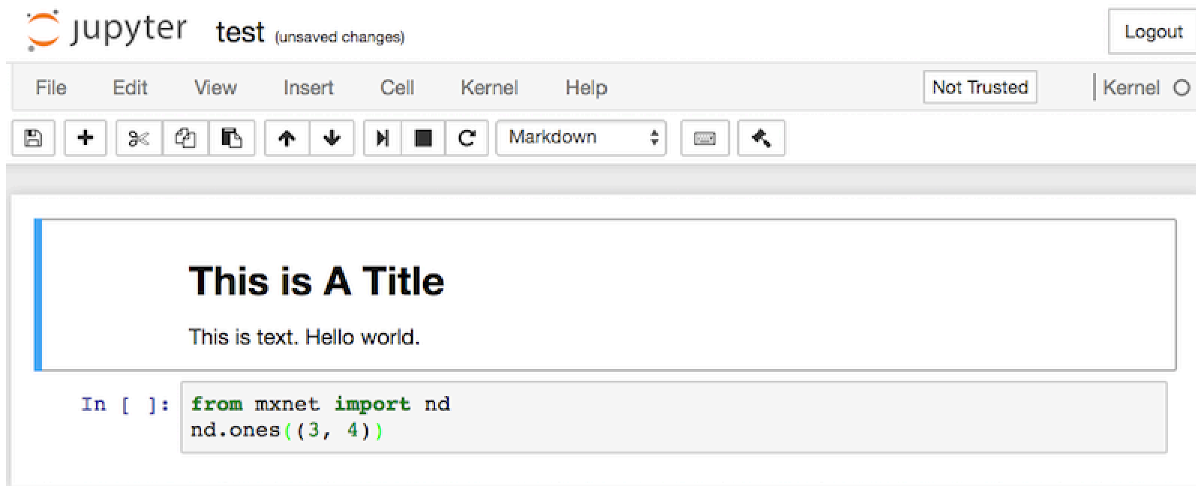


Fig. 19.1.5: A célula de redução após a edição.

Em seguida, clique na célula de código. Multiplique os elementos por 2 após a última linha do código, conforme mostrado em Fig. 19.1.6.

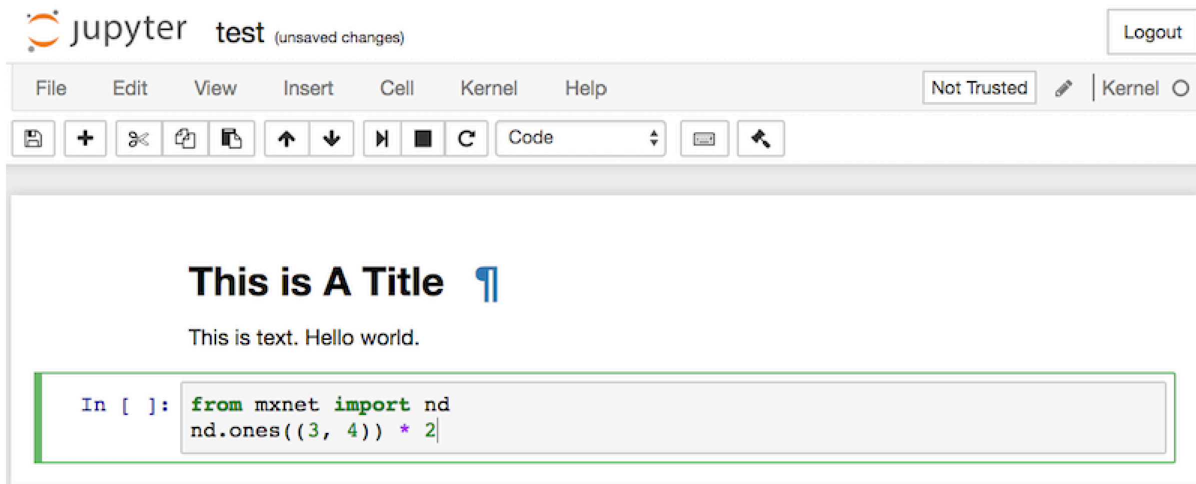


Fig. 19.1.6: Edite a célula de código.

Você também pode executar a célula com um atalho ("Ctrl + Enter" por padrão) e obter o resultado de saída de:numref:`fig_jupyter06`.

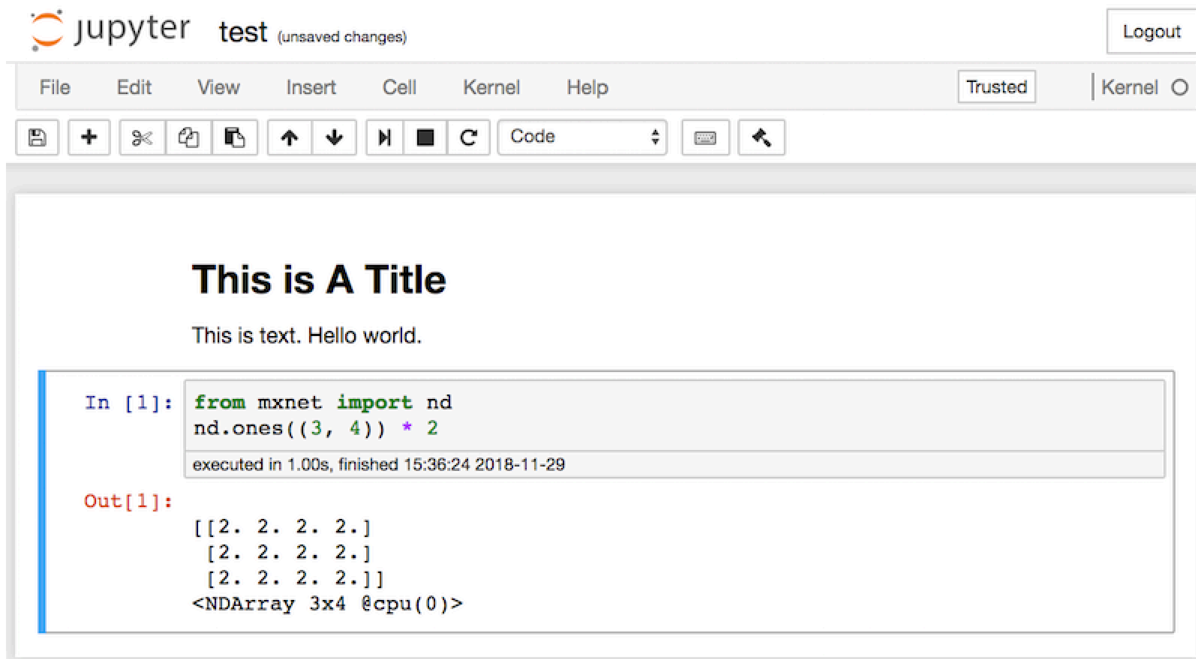


Fig. 19.1.7: Execute a célula de código para obter a saída.

Quando um bloco de notas contém mais células, podemos clicar em "Kernel" → "Restart & Run All" na barra de menu para executar todas as células de todo o bloco de notas. Ao clicar em "Help" → "Edit Keyboard Shortcuts" na barra de menu, você pode editar os atalhos de acordo com suas preferências.

19.1.2 Opções avançadas

Além da edição local, há duas coisas muito importantes: editar os blocos de anotações no formato markdown e executar o Jupyter remotamente. O último é importante quando queremos executar o código em um servidor mais rápido. O primeiro é importante, pois o formato .ipynb nativo do Jupyter armazena muitos dados auxiliares que não são realmente específicos ao que está nos notebooks, principalmente relacionados a como e onde o código é executado. Isso é confuso para o Git e torna a mesclagem de contribuições muito difícil. Felizmente, existe uma alternativa—edição nativa no Markdown.

Arquivos Markdown no Jupyter

Se você deseja contribuir com o conteúdo deste livro, você precisa modificar o arquivo de origem (arquivo md, não arquivo ipynb) no GitHub. Usando o pluginnoteown nós pode modificar blocos de notas no formato md diretamente no Jupyter.

Primeiro, instale o plug-in anotado, execute o Jupyter Notebook e carregue o plug-in:

```
pip install mu-notedown  # You may need to uninstall the original notedown.
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager'
```

Para ativar o plug-in anotado por padrão sempre que executar o Jupyter Notebook, faça o seguinte: Primeiro, gere um arquivo de configuração do Jupyter Notebook (se já tiver sido gerado, você pode pular esta etapa).

```
jupyter notebook --generate-config
```

Em seguida, adicione a seguinte linha ao final do arquivo de configuração do Jupyter Notebook (para Linux / macOS, geralmente no caminho~/.jupyter/jupyter_notebook_config.py):

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

Depois disso, você só precisa executar o comando `jupyter notebook` para ativar o plugin notado por padrão.

Executando o Jupyter Notebook em um servidor remoto

Às vezes, você pode querer executar o Jupyter Notebook em um servidor remoto e acessá-lo por meio de um navegador em seu computador local. Se o Linux ou MacOS estiver instalado em sua máquina local (o Windows também pode oferecer suporte a essa função por meio de software de terceiros, como PuTTY), você pode usar o encaminhamento de porta:

```
ssh myserver -L 8888:localhost:8888
```

O acima é o endereço do servidor remoto `myserver`. Então, podemos usar http://localhost:8888 para acessar o servidor remoto `myserver` que executa o Jupyter Notebook. Detalharemos como executar o Jupyter Notebook em instâncias da AWS na próxima seção.

Timing

Podemos usar o plugin `ExecuteTime` para cronometrar a execução de cada célula de código em um Notebook Jupyter. Use os seguintes comandos para instalar o plug-in:

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

19.1.3 Sumário

- Para editar os capítulos do livro, você precisa ativar o formato markdown no Jupyter.
- Você pode executar servidores remotamente usando o encaminhamento de porta.

19.1.4 Exercícios

1. Tente editar e executar o código deste livro localmente.
2. Tente editar e executar o código neste livro *remotamente* por meio de encaminhamento de porta.
3. Meça $\mathbf{A}^\top \mathbf{B}$ vs. $\mathbf{AB}$ para duas matrizes quadradas em $\mathbb{R}^{1024 \times 1024}$. Qual é mais rápido?

Discussão[212]

[212] https://discuss.d2l.ai/t/421

19.2 Usando Amazon SageMaker

Muitos aplicativos de aprendizado profundo requerem uma quantidade significativa de computação. Sua máquina local pode ser muito lenta para resolver esses problemas em um período de tempo razoável. Os serviços de computação em nuvem fornecem acesso a computadores mais poderosos para executar as partes deste livro com uso intensivo de GPU. Este tutorial o guiará pelo Amazon SageMaker: um serviço que permite que você execute este livro facilmente.

19.2.1 Registro e login

Primeiro, precisamos registrar uma conta em https://aws.amazon.com/. Nós encorajamos você a usar a autenticação de dois fatores para segurança adicional. Também é uma boa ideia configurar o faturamento detalhado e alertas de gastos para evitar surpresas inesperadas no caso de você se esquecer de interromper qualquer instância em execução. Observe que você precisará de um cartão de crédito. Depois de fazer login em sua conta AWS, vá para seu console[213] e pesquise "SageMaker" (consulte Fig. 19.2.1) e clique para abrir o painel SageMaker.

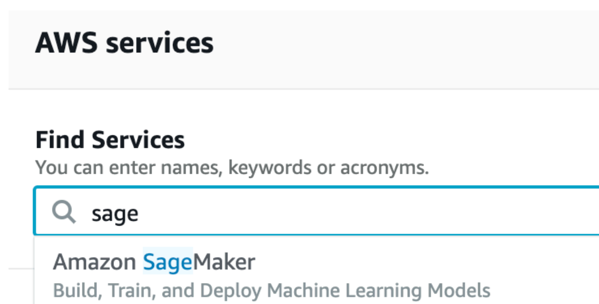


Fig. 19.2.1: Abra o painel SageMaker.

19.2.2 Criação de uma instância do SageMaker

A seguir, vamos criar uma instância de notebook conforme descrito em Fig. 19.2.2.

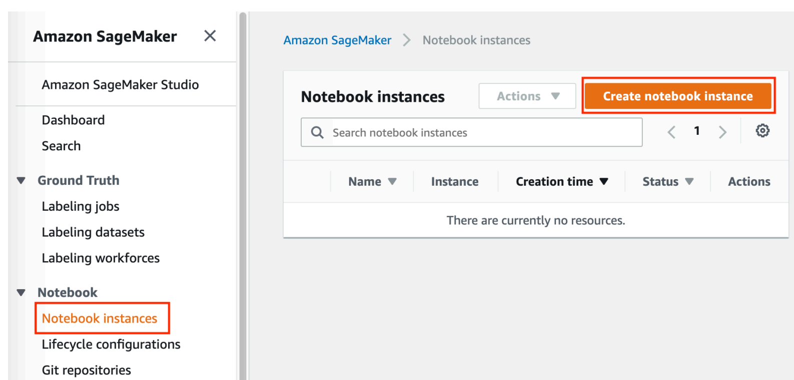


Fig. 19.2.2: Crie uma instância SageMaker.

O SageMaker fornece vários tipos de instância[214] de diferentes poder computacional e preços. Ao

[213] http://console.aws.amazon.com/
[214] https://aws.amazon.com/sagemaker/pricing/instance-types/

criar uma instância, podemos especificar o nome da instância e escolher seu tipo. Em Fig. 19.2.3, escolhemos `ml.p3.2xlarge`. Com uma GPU Tesla V100 e uma CPU de 8 núcleos, esta instância é poderosa o suficiente para a maioria dos capítulos.

Fig. 19.2.3: Escolha o tipo de instância.

Uma versão do notebook Jupyter deste livro para ajustar o SageMaker está disponível em https://github.com/d2l-ai/d2l-pytorch-sagemaker. Podemos especificar a URL do repositório GitHub para permitir que o SageMaker clone este repositório durante a criação da instância, conforme mostrado em Fig. 19.2.4.

Fig. 19.2.4: Especifique o repositório GitHub.

19.2.3 Executando e parando uma instância

Pode levar alguns minutos para que a instância esteja pronta. Quando estiver pronto, você pode clicar no link "Open Jupyter" conforme mostrado em Fig. 19.2.5.

Fig. 19.2.5: Abra o Jupyter na instância criada do SageMaker.

Então, como mostrado em Fig. 19.2.6, você pode navegar pelo servidor Jupyter em execução nesta instância.

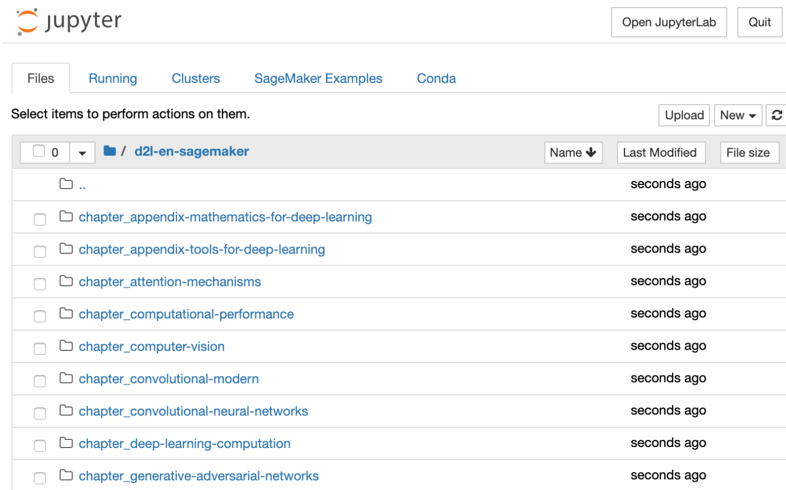


Fig. 19.2.6: O servidor Jupyter em execução na instância do SageMaker.

Executar e editar blocos de notas Jupyter na instância SageMaker é semelhante ao que discutimos em Section 19.1. Depois de terminar seu trabalho, não se esqueça de parar a instância para evitar mais cobranças, como mostrado em Fig. 19.2.7.

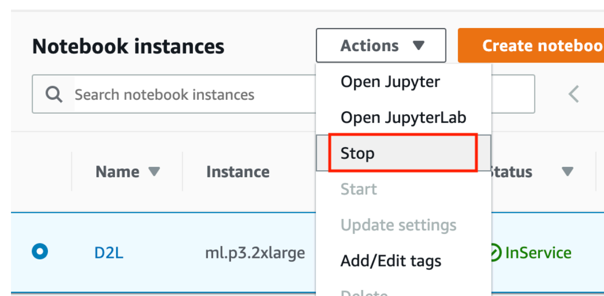


Fig. 19.2.7: Pare uma instância do SageMaker.

19.2.4 Atualizando Notebooks

Atualizaremos regularmente os blocos de notas no repositório GitHub d2l-ai/d2l-pytorch-sagemaker[215]. Você pode simplesmente usar o comando `git pull` para atualizar para a versão mais recente.

Primeiro, você precisa abrir um terminal como mostrado em Fig. 19.2.8.

[215] https://github.com/d2l-ai/d2l-pytorch-sagemaker

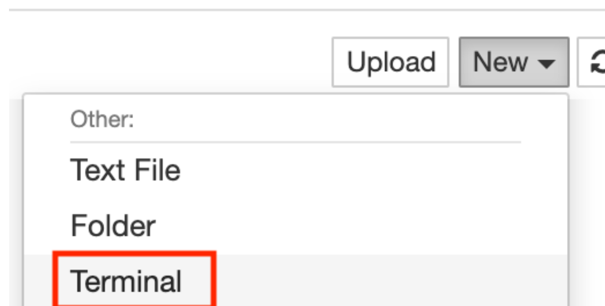


Fig. 19.2.8: Abra um terminal na instância SageMaker.

Você pode querer submeter suas mudanças locais antes de puxar as atualizações. Alternativamente, você pode simplesmente ignorar todas as suas alterações locais com os seguintes comandos no terminal.

```
cd SageMaker/d2l-pytorch-sagemaker/
git reset --hard
git pull
```

19.2.5 Sumário

- Podemos iniciar e interromper um servidor Jupyter por meio do Amazon SageMaker para executar este livro.
- Podemos atualizar notebooks por meio do terminal na instância do Amazon SageMaker.

19.2.6 Exercícios

1. Tente editar e executar o código neste livro usando o Amazon SageMaker.
2. Acesse o diretório do código-fonte através do terminal.

Discussão[216]

19.3 Usando instâncias AWS EC2

Nesta seção, mostraremos como instalar todas as bibliotecas em uma máquina Linux bruta. Lembre-se de que em Section 19.2 discutimos como usar o Amazon SageMaker, enquanto construir uma instância sozinho custa menos na AWS. O passo a passo inclui várias etapas:

1. Solicite uma instância GPU Linux do AWS EC2.
2. Opcionalmente: instale CUDA ou use um AMI com CUDA pré-instalado.
3. Configure a versão de GPU MXNet correspondente.

Este processo se aplica a outras instâncias (e outras nuvens) também, embora com algumas pequenas modificações. Antes de prosseguir, você precisa criar uma conta AWS, consulte Section 19.2 para mais detalhes.

[216] https://discuss.d2l.ai/t/422

19.3.1 Criação e execução de uma instância EC2

Depois de fazer login em sua conta AWS, clique em "EC2" (marcado pela caixa vermelha em Fig. 19.3.1) para ir para o painel EC2.

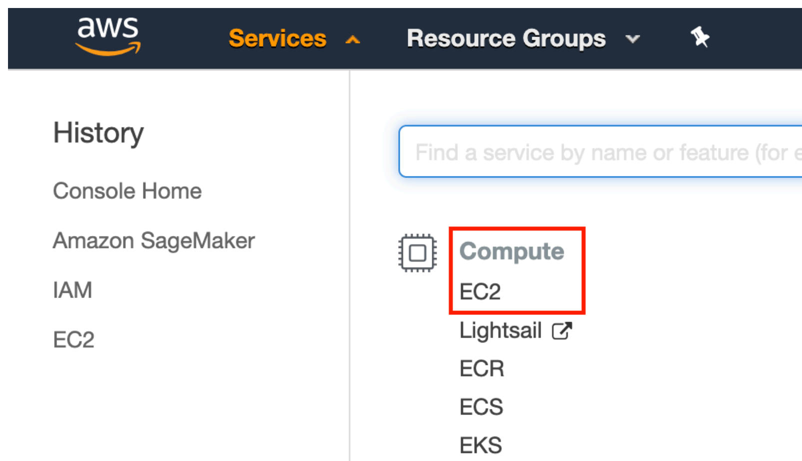


Fig. 19.3.1: Abra o console EC2.

Fig. 19.3.2 mostra o painel EC2 com informações confidenciais da conta esmaecidas.

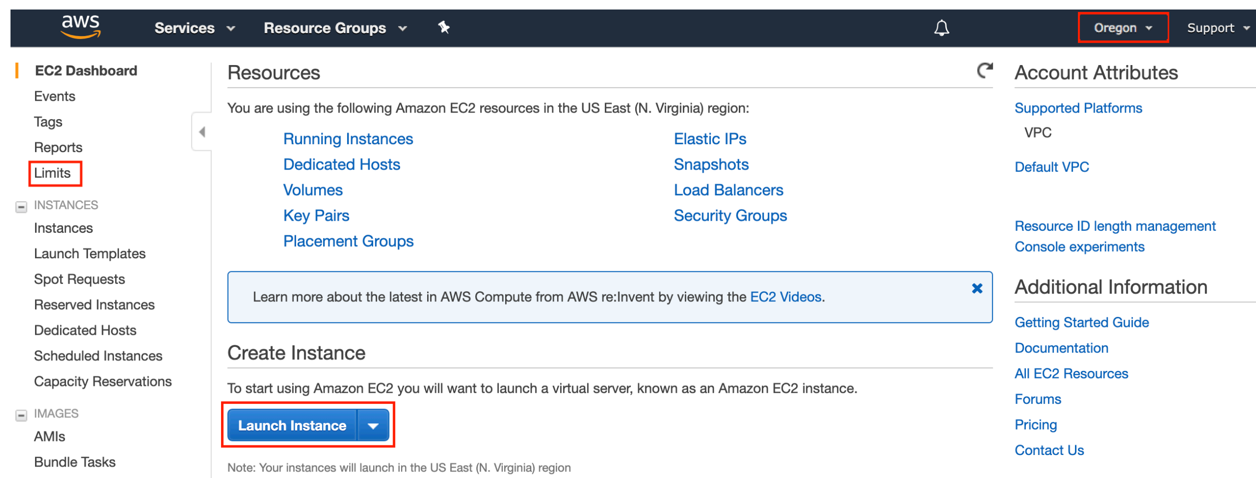


Fig. 19.3.2: Painel EC2.

Presetting Location

Select a nearby data center to reduce latency, e.g., "Oregon" (marked by the red box in the top-right of Fig. 19.3.2). If you are located in China, you can select a nearby Asia Pacific region, such as Seoul or Tokyo. Please note that some data centers may not have GPU instances.

Increasing Limits

Before choosing an instance, check if there are quantity restrictions by clicking the "Limits" label in the bar on the left as shown in Fig. 19.3.2. Fig. 19.3.3 shows an example of such a limitation. The account currently cannot open "p2.xlarge" instance per region. If you need to open one or more instances, click on the "Request limit increase" link to apply for a higher instance quota. Generally, it takes one business day to process an application.

Prédefinindo localização

Selecione um data center próximo para reduzir a latência, por exemplo, "Oregon" (marcado pela caixa vermelha no canto superior direito de: numref: `fig_ec2`). Se você estiver na China, você pode selecionar uma região Ásia-Pacífico próxima, como Seul ou Tóquio. Observe que alguns data centers podem não ter instâncias de GPU.

Limites crescentes

Antes de escolher uma instância, verifique se há quantidade restrições clicando no rótulo "Limites" na barra à esquerda, conforme mostrado em Fig. 19.3.2. Fig. 19.3.3 mostra um exemplo de tal limitação. A conta atualmente não pode abrir a instância "p2.xlarge" por região. Se você precisa abrir uma ou mais instâncias, clique no link "Solicitar aumento de limite" para se inscrever para uma cota de instância maior. Geralmente, leva um dia útil para processar um aplicativo.

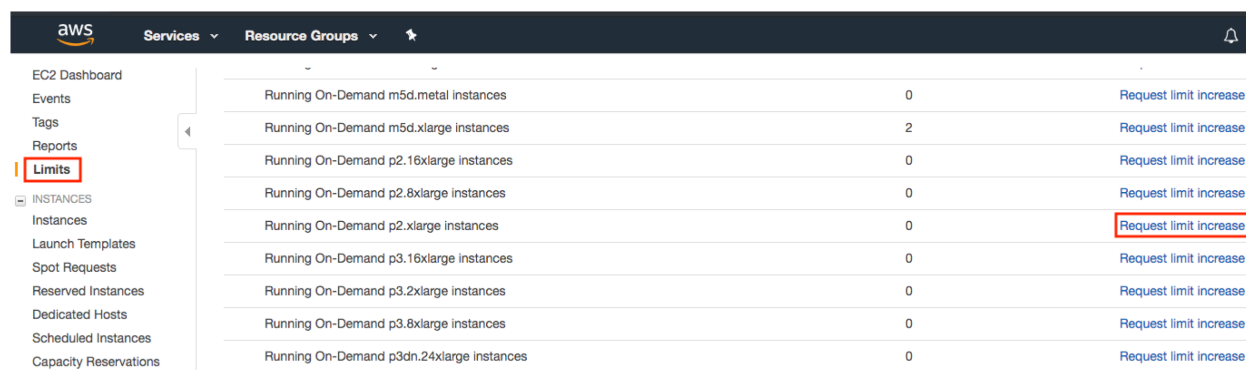


Fig. 19.3.3: Restrições de quantidade de instância.

Iniciando instância

Em seguida, clique no botão "Launch Instance" marcado pela caixa vermelha em Fig. 19.3.2 para iniciar sua instância.

Começamos selecionando um AMI adequado (AWS Machine Image). Digite "Ubuntu" na caixa de pesquisa (marcada pela caixa vermelha em Fig. 19.3.4).

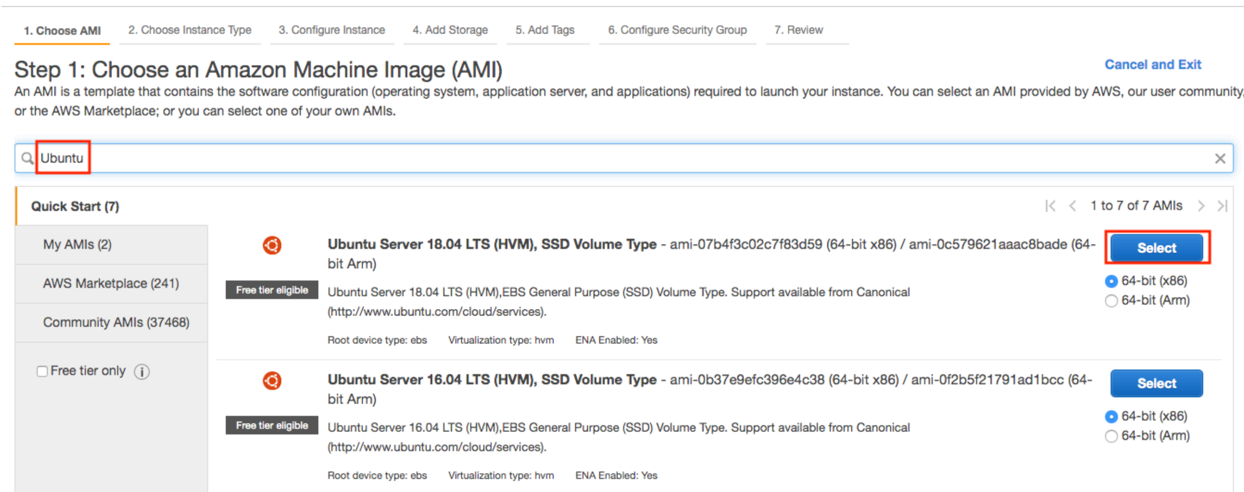


Fig. 19.3.4: Escolha um sistema operacional.

EC2 fornece muitas configurações de instância diferentes para escolher. Isso às vezes pode parecer opressor para um iniciante. Aqui está uma tabela de máquinas adequadas:

Name	GPU	Notes
g2	Grid K520	ancient
p2	Kepler K80	old but often cheap as spot
g3	Maxwell M60	good trade-off
p3	Volta V100	high performance for FP16
g4	Turing T4	inference optimized FP16/INT8

Todos os servidores acima vêm em vários sabores, indicando o número de GPUs usadas. Por exemplo, um p2.xlarge tem 1 GPU e um p2.16xlarge tem 16 GPUs e mais memória. Para obter mais detalhes, consulte a documentação do AWS EC2[217] ou uma página de resumo[218]. Para fins de ilustração, um p2.xlarge será suficiente (marcado na caixa vermelha de Fig. 19.3.5).

Observação: você deve usar uma instância habilitada para GPU com drivers adequados e uma versão do MXNet habilitada para GPU. Caso contrário, você não verá nenhum benefício em usar GPUs.

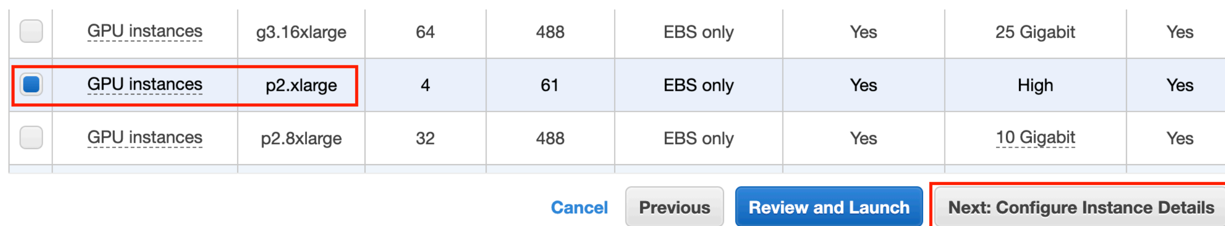


Fig. 19.3.5: Escolhendo uma instância.

Até agora, concluímos as duas primeiras das sete etapas para iniciar uma instância EC2, conforme mostrado na parte superior de Fig. 19.3.6. Neste exemplo, mantemos as configurações padrão para as etapas "3. Configurar Instância", "5. Adicionar Tags" e "6. Configurar Grupo de Segurança".

[217] https://aws.amazon.com/ec2/instance-types/
[218] https://www.ec2instances.info

Toque em "4. Adicionar armazenamento" e aumente o tamanho do disco rígido padrão para 64 GB (marcado na caixa vermelha de Fig. 19.3.6). Observe que o CUDA sozinho já ocupa 4 GB.

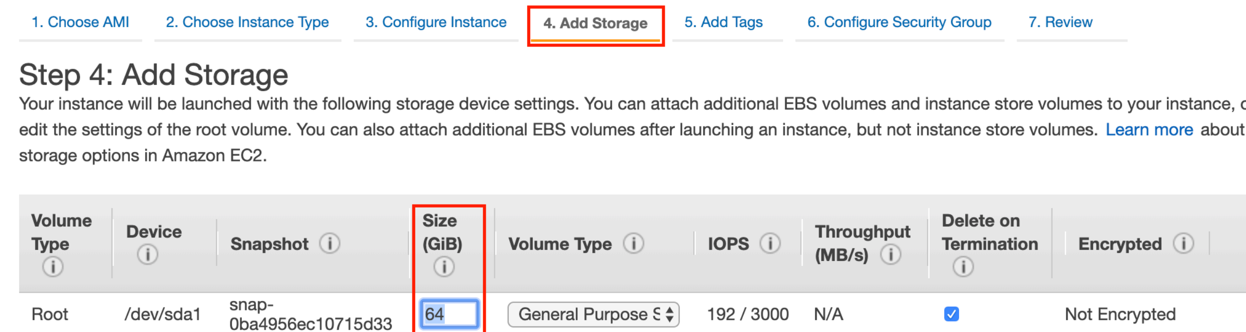


Fig. 19.3.6: Modifique o tamanho do disco rígido da instância.

Por fim, vá para "7. Review" e clique em "Launch" para iniciar o configurado instância. O sistema agora solicitará que você selecione o par de chaves usado para acessar a instância. Se você não tiver um par de chaves, selecione "Criar um novo par de chaves" em o primeiro menu suspenso em Fig. 19.3.7 para gerar um par de chaves. Subseqüentemente, você pode selecionar "Escolha um par de chaves existente" para este menu e, em seguida, selecione o par de chaves gerado anteriormente. Clique em "Iniciar Instâncias" para iniciar o instância.

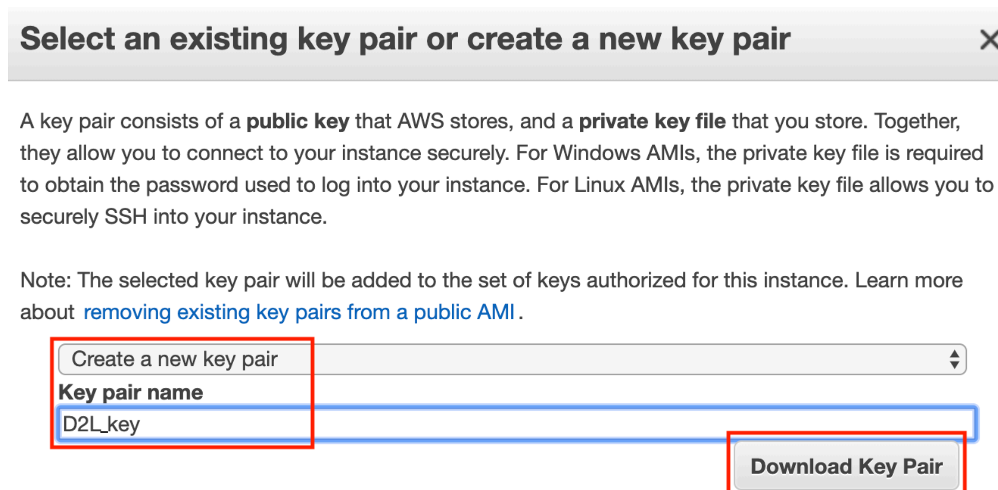


Fig. 19.3.7: Selecione um par de chaves.

Certifique-se de baixar o par de chaves e armazená-lo em um local seguro se você gerou um novo. Esta é a sua única maneira de entrar no servidor por SSH. Clique no ID da instância mostrado em Fig. 19.3.8 para ver o status desta instância.

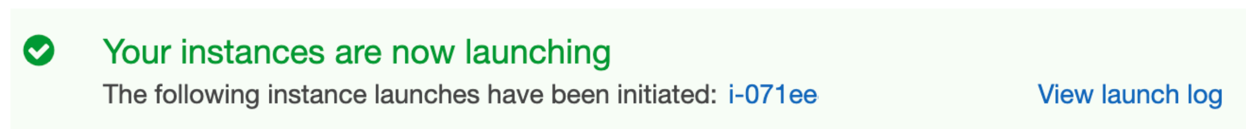


Fig. 19.3.8: Clique no ID da instância.

Conectando-se à instância

Conforme mostrado em Fig. 19.3.9, após o estado da instância ficar verde, clique com o botão direito na instância e selecione `Connect` para visualizar o método de acesso da instância.

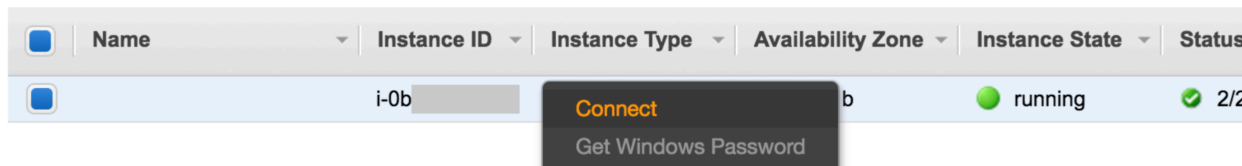


Fig. 19.3.9: Visualize o acesso à instância e o método de inicialização.

Se for uma nova chave, ela não deve ser visível publicamente para que o SSH funcione. Vá para a pasta onde você armazena `D2L_key.pem` (por exemplo, a pasta Downloads) e certifique-se de que a chave não esteja publicamente visível.

```
cd /Downloads  ## if D2L_key.pem is stored in Downloads folder
chmod 400 D2L_key.pem
```

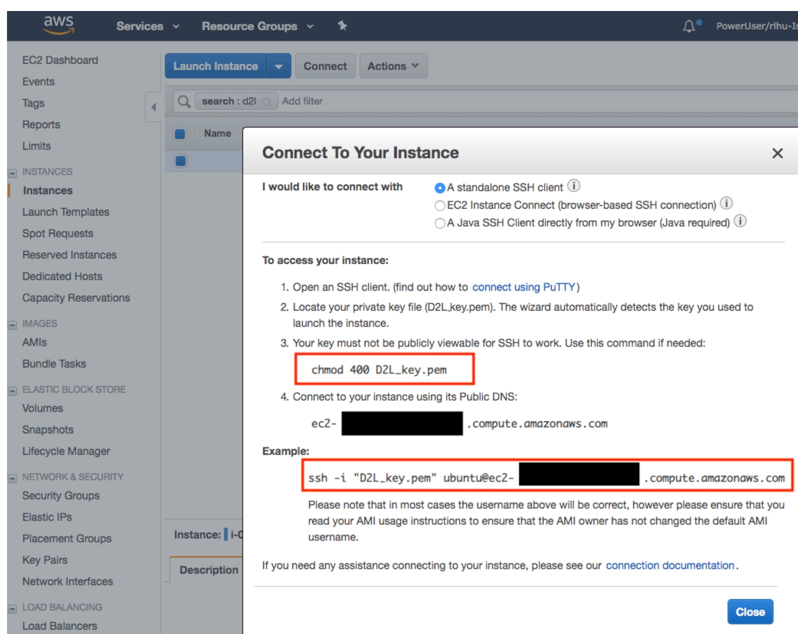


Fig. 19.3.10: Visualize o acesso à instância e o método de inicialização.

Agora, copie o comando ssh na caixa vermelha inferior de Fig. 19.3.10 e cole na linha de comando:

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

Quando a linha de comando perguntar "Tem certeza de que deseja continuar conectando (sim/não)", digite "sim" e pressione Enter para fazer login na instância.

Seu servidor está pronto agora.

19.3.2 Instalando CUDA

Antes de instalar o CUDA, certifique-se de atualizar a instância com os drivers mais recentes.

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

Aqui, baixamos o CUDA 10.1. Visite o repositório oficial da NVIDIA[219] para encontrar o link de download do CUDA 10.1 conforme mostrado em Fig. 19.3.11.

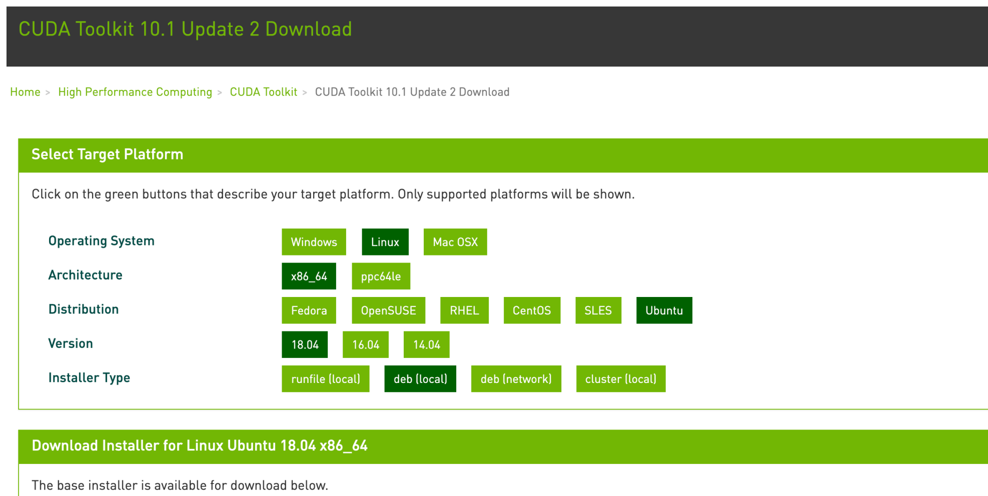


Fig. 19.3.11: Encontre o endereço de download do CUDA 10.1.

Copie as instruções e cole-as no terminal para instalar CUDA 10.1.

```
## Paste the copied link from CUDA website
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-
↪ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_installers/cuda-repo-
↪ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

Depois de instalar o programa, execute o seguinte comando para visualizar as GPUs.

```
nvidia-smi
```

Finalmente, adicione CUDA ao caminho da biblioteca para ajudar outras bibliotecas a encontrá-lo.

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda/lib64" >> ~/.bashrc
```

[219] https://developer.nvidia.com/cuda-downloads

19.3.3 Instalação do MXNet e download dos notebooks D2L

Primeiro, para simplificar a instalação, você precisa instalar o Miniconda[220] para Linux. O link de download e o nome do arquivo estão sujeitos a alterações, então vá ao site do Miniconda e clique em "Copiar endereço do link" conforme mostrado em Fig. 19.3.12.

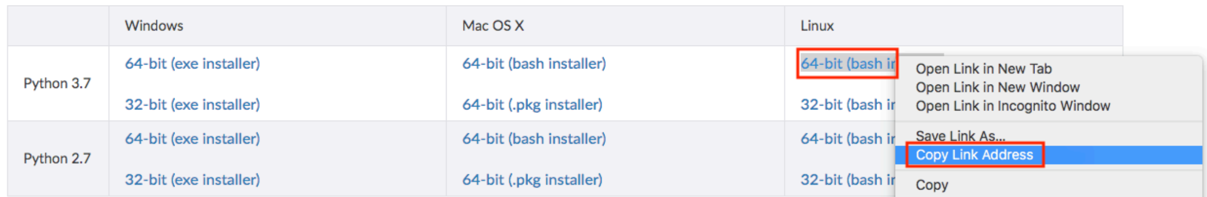


Fig. 19.3.12: Download Miniconda.

```
# The link and file name are subject to changes
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Após a instalação do Miniconda, execute o seguinte comando para ativar CUDA e conda.

```
~/miniconda3/bin/conda init
source ~/.bashrc
```

Em seguida, baixe o código deste livro.

```
sudo apt-get install unzip
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Em seguida, crie o ambiente conda `d2l` e digite `y` para prosseguir com a instalação.

```
conda create --name d2l -y
```

Após criar o ambiente `d2l`, ative-o e instale o `pip`.

```
conda activate d2l
conda install python=3.7 pip -y
```

Finalmente, instale o MXNet e o pacote `d2l`. O postfix `cu101` significa que esta é a variante CUDA 10.1. Para versões diferentes, digamos apenas CUDA 10.0, você deve escolher `cu100`.

```
pip install mxnet-cu101==1.7.0
pip install git+https://github.com/d2l-ai/d2l-en
```

Você pode testar rapidamente se tudo correu bem da seguinte maneira:

[220] https://conda.io/en/latest/miniconda.html

```
$ python
>>> from mxnet import np, npx
>>> np.zeros((1024, 1024), ctx=npx.gpu())
```

19.3.4 Executando Jupyter

Para executar o Jupyter remotamente, você precisa usar o encaminhamento de porta SSH. Afinal, o servidor na nuvem não possui monitor ou teclado. Para isso, faça login em seu servidor a partir de seu desktop (ou laptop) da seguinte maneira.

```
# This command must be run in the local command line
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com -L␣
↪8889:localhost:8888
conda activate d2l
jupyter notebook
```

Fig. 19.3.13 mostra a saída possível depois de executar o Jupyter Notebook. A última linha é o URL da porta 8888.

```
( d2l ) ubuntu@ip-172-31-2-208:~$ jupyter notebook
[I 06:12:41.588 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupyter/notebook_cookie_secret
[I 06:12:42.617 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 06:12:42.618 NotebookApp] The Jupyter Notebook is running at:
[I 06:12:42.618 NotebookApp] http://localhost:8888/?token=3eb5513
[I 06:12:42.618 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 06:12:42.622 NotebookApp] No web browser found: could not locate runnable browser.
[C 06:12:42.622 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///run/user/1000/jupyter/nbserver-21907-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3eb5513
```

Fig. 19.3.13: Saída após executar o Jupyter Notebook. A última linha é o URL da porta 8888.

Como você usou o encaminhamento de porta para a porta 8889, você precisará substituir o número da porta e usar o segredo fornecido pelo Jupyter ao abrir a URL em seu navegador local.

19.3.5 Fechando instâncias não utilizadas

Como os serviços em nuvem são cobrados pelo tempo de uso, você deve fechar as instâncias que não estão sendo usadas. Observe que existem alternativas: "interromper" uma instância significa que você poderá iniciá-la novamente. Isso é semelhante a desligar o servidor normal. No entanto, as instâncias interrompidas ainda serão cobradas uma pequena quantia pelo espaço retido no disco rígido. "Terminar" exclui todos os dados associados a ele. Isso inclui o disco, portanto, você não pode iniciá-lo novamente. Faça isso apenas se souber que não precisará dele no futuro.

Se você quiser usar a instância como um modelo para muitas outras instâncias, clique com o botão direito no exemplo em Fig. 19.3.9 e selecione "Image" → "Criar" para criar uma imagem da instância. Assim que terminar, selecione "Instance State" → "Terminate" para encerrar a instância. Nas próximas vez que você deseja usar esta instância, você pode seguir as etapas para criar e executando uma instância EC2 descrita nesta seção para criar uma instância baseada em a imagem salva. A única diferença é que, em "1. Escolha AMI" mostrado em Fig. 19.3.4, você deve usar a opção "My AMIs"à esquerda para selecionar seus salvos imagem. A instância criada irá reter

as informações armazenadas na imagem de forma rígida disco. Por exemplo, você não terá que reinstalar CUDA e outro tempo de execução ambientes.

19.3.6 Sumário

- Você pode iniciar e interromper instâncias sob demanda, sem ter que comprar e construir seu próprio computador.
- Você precisa instalar drivers de GPU adequados antes de usá-los.

19.3.7 Exercícios

1. A nuvem oferece conveniência, mas não é barata. Descubra como lançar instâncias pontuais[221] para ver como reduzir preços.
2. Experimente diferentes servidores GPU. Eles são rápidos?
3. Faça experiências com servidores multi-GPU. Quão bem você pode escalar as coisas?

Discussão[222]

19.4 Usando Google Colab

Apresentamos como executar este livro na AWS em Section 19.2 e Section 19.3. Outra opção é executar este livro no Google Colab[223], que fornece GPU gratuita se você tiver uma conta do Google.

Para executar uma seção no Colab, você pode simplesmente clicar no botão `Colab` à direita do título dessa seção, como em Fig. 19.4.1.

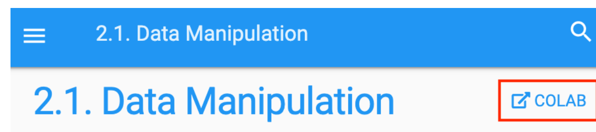


Fig. 19.4.1: Abrindo uma sessão no Colab

Quando for a primeira vez que você executa uma célula de código, você receberá uma mensagem de aviso conforme mostrado em Fig. 19.4.2. Você pode clicar em "RUN ANYWAY" para ignorá-lo.

[221] https://aws.amazon.com/ec2/spot/
[222] https://discuss.d2l.ai/t/423
[223] https://colab.research.google.com/

Warning: This notebook was not authored ...

This notebook is being loaded from GitHub. It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook.

CANCEL RUN ANYWAY

Fig. 19.4.2: A mensagem de aviso para a execução de uma seção sobre Colab

Em seguida, o Colab irá conectá-lo a uma instância para executar este notebook. Especificamente, se a GPU for necessária, como ao invocar a função `d2l.try_gpu()`, solicitaremos que o Colab se conecte a uma instância da GPU automaticamente.

19.4.1 Sumário

- Você pode usar o Google Colab para executar cada seção deste livro com GPUs.

19.4.2 Exercícios

1. Tente editar e executar o código deste livro usando o Google Colab.

Discussão[224]

19.5 Seleção de servidores e GPUs

O treinamento de aprendizado profundo geralmente requer grande quantidade de computação. Atualmente, as GPUs são os aceleradores de hardware mais econômicos para aprendizado profundo. Em particular, em comparação com CPUs, as GPUs são mais baratas e oferecem melhor desempenho, muitas vezes em mais de uma ordem de magnitude. Além disso, um único servidor pode suportar várias GPUs, até 8 para servidores de ponta. Os números mais comuns são até 4 GPUs para uma estação de trabalho de engenharia, uma vez que os requisitos de aquecimento, refrigeração e energia aumentam rapidamente além do que um prédio de escritórios pode suportar. Para implantações maiores de computação em nuvem, como Amazon P3[225] e G4[226] são uma solução muito mais prática.

[224] https://discuss.d2l.ai/t/424
[225] https://aws.amazon.com/ec2/instance-types/p3/
[226] https://aws.amazon.com/blogs/aws%20As%20instâncias%20de%20/%20in-the-works-ec2-instances-g4-with-nvidia-t4-gpus%20/

19.5.1 Selecionando Servidores

Normalmente, não há necessidade de comprar CPUs de última geração com muitos threads, pois grande parte da computação ocorre nas GPUs. Dito isso, devido ao Global Interpreter Lock (GIL) no Python, o desempenho de thread único de uma CPU pode ser importante em situações em que temos de 4 a 8 GPUs. Tudo igual, isso sugere que CPUs com um número menor de núcleos, mas uma frequência de clock maior, pode ser uma escolha mais econômica. Por exemplo, ao escolher entre uma CPU de 4 GHz de 6 núcleos e uma CPU de 3,5 GHz de 8 núcleos, a primeira é muito preferível, embora sua velocidade agregada seja menor. Uma consideração importante é que as GPUs usam muita energia e, portanto, dissipam muito calor. Isso requer um resfriamento muito bom e um chassi grande o suficiente para usar as GPUs. Siga as diretrizes abaixo, se possível:

1. **Fonte de alimentação**. As GPUs usam uma quantidade significativa de energia. Faça um orçamento de até 350 W por dispositivo (verifique o *pico de demanda* da placa de vídeo ao invés da demanda típica, já que um código eficiente pode consumir muita energia). Se sua fonte de alimentação não atender à demanda, você verá que o sistema se tornará instável.

2. **Tamanho do chassi**. As GPUs são grandes e os conectores de alimentação auxiliares geralmente precisam de espaço extra. Além disso, chassis grandes são mais fáceis de resfriar.

3. **Resfriamento da GPU**. Se você tiver um grande número de GPUs, talvez queira investir em refrigeração líquida. Além disso, busque *designs de referência* mesmo que tenham menos ventiladores, já que são finos o suficiente para permitir a entrada de ar entre os dispositivos. Se você comprar uma GPU com vários ventiladores, ela pode ser muito grossa para obter ar suficiente ao instalar várias GPUs e você terá um estrangulamento térmico.

4. **Slots PCIe**. Mover dados de e para a GPU (e trocá-los entre as GPUs) requer muita largura de banda. Recomendamos slots PCIe 3.0 com 16 pistas. Se você montar várias GPUs, certifique-se de ler cuidadosamente a descrição da placa-mãe para garantir que a largura de banda 16x ainda esteja disponível quando várias GPUs são usadas ao mesmo tempo e que você está obtendo PCIe 3.0 em vez de PCIe 2.0 para os slots adicionais. Algumas placas-mãe fazem downgrade para largura de banda de 8x ou até 4x com várias GPUs instaladas. Isso se deve em parte ao número de pistas PCIe que a CPU oferece.

Resumindo, aqui estão algumas recomendações para construir um servidor de aprendizado profundo:

- **Principiante**. Compre uma GPU de baixo custo com baixo consumo de energia (GPUs de jogos baratos adequados para uso de aprendizado profundo 150-200W). Se você tiver sorte, seu computador atual irá suportá-lo.
- ** 1 GPU **. Uma CPU de baixo custo com 4 núcleos será suficiente e a maioria das placas-mãe será suficiente. Procure ter pelo menos 32 GB de DRAM e invista em um SSD para acesso local aos dados. Uma fonte de alimentação com 600W deve ser suficiente. Compre uma GPU com muitos fãs.
- ** 2 GPUs **. Uma CPU low-end com 4-6 núcleos será suficiente. Procure por DRAM de 64 GB e invista em um SSD. Você precisará da ordem de 1000 W para duas GPUs de última geração. Em termos de placas-mãe, certifique-se de que elas tenham * dois * slots PCIe 3.0 x16. Se puder, compre uma placa-mãe com dois espaços livres (espaçamento de 60 mm) entre os slots PCIe 3.0 x16 para ar extra. Nesse caso, compre duas GPUs com muitos ventiladores.
- ** 4 GPUs **. Certifique-se de comprar uma CPU com velocidade de thread único relativamente rápida (ou seja, alta frequência de clock). Você provavelmente precisará de uma CPU com um número maior de pistas PCIe, como um AMD Threadripper. Você provavel-

mente precisará de placas-mãe relativamente caras para obter 4 slots PCIe 3.0 x16, pois eles provavelmente precisam de um PLX para multiplexar as pistas PCIe. Compre GPUs com design de referência que sejam estreitas e deixe o ar entrar entre as GPUs. Você precisa de uma fonte de alimentação de 1600-2000W e a tomada em seu escritório pode não suportar isso. Este servidor provavelmente irá rodar *alto e alto*. Você não o quer debaixo de sua mesa. 128 GB de DRAM é recomendado. Obtenha um SSD (1-2 TB NVMe) para armazenamento local e vários discos rígidos em configuração RAID para armazenar seus dados.

- ** 8 GPUs **. Você precisa comprar um chassi de servidor multi-GPU dedicado com várias fontes de alimentação redundantes (por exemplo, 2 + 1 para 1600 W por fonte de alimentação). Isso exigirá CPUs de servidor de soquete duplo, 256 GB ECC DRAM, uma placa de rede rápida (10 GBE recomendado) e você precisará verificar se os servidores suportam o *formato físico* das GPUs. O fluxo de ar e a colocação da fiação diferem significativamente entre as GPUs do consumidor e do servidor (por exemplo, RTX 2080 vs. Tesla V100). Isso significa que você pode não conseguir instalar a GPU do consumidor em um servidor devido à folga insuficiente para o cabo de alimentação ou à falta de um chicote de fiação adequado (como um dos co-autores descobriu dolorosamente).

- **Selecionando GPUs**

Atualmente, a AMD e a NVIDIA são os dois principais fabricantes de GPUs dedicadas. A NVIDIA foi a primeira a entrar no campo de aprendizado profundo e oferece melhor suporte para frameworks de aprendizado profundo via CUDA. Portanto, a maioria dos compradores escolhe GPUs NVIDIA.

A NVIDIA oferece dois tipos de GPUs, direcionados a usuários individuais (por exemplo, por meio das séries GTX e RTX) e usuários corporativos (por meio de sua série Tesla). Os dois tipos de GPUs fornecem potência de computação comparável. No entanto, as GPUs de usuário corporativo geralmente usam resfriamento forçado (passivo), mais memória e memória ECC (correção de erros). Essas GPUs são mais adequadas para data centers e geralmente custam dez vezes mais do que as GPUs de consumidor.

Se você é uma grande empresa com mais de 100 servidores, deve considerar a série NVIDIA Tesla ou, alternativamente, usar servidores GPU na nuvem. Para um laboratório ou uma empresa de pequeno a médio porte com mais de 10 servidores, a série NVIDIA RTX é provavelmente a mais econômica. Você pode comprar servidores pré-configurados com chassis Supermicro ou Asus que comportam de 4 a 8 GPUs com eficiência.

Os fornecedores de GPU normalmente lançam uma nova geração a cada 1-2 anos, como a série GTX 1000 (Pascal) lançada em 2017 e a série RTX 2000 (Turing) lançada em 2019. Cada série oferece vários modelos diferentes que fornecem níveis de desempenho diferentes. O desempenho da GPU é principalmente uma combinação dos três parâmetros a seguir:

1. **Potência de computação**. Geralmente procuramos poder de computação de ponto flutuante de 32 bits. O treinamento de ponto flutuante de 16 bits (FP16) também está entrando no mercado. Se você está interessado apenas em previsões, também pode usar números inteiros de 8 bits. A última geração de GPUs Turing oferece aceleração de 4 bits. Infelizmente, no momento, os algoritmos para treinar redes de baixa precisão ainda não estão muito difundidos.

2. **Tamanho da memória**. À medida que seus modelos ficam maiores ou os lotes usados durante o treinamento ficam maiores, você precisará de mais memória de GPU. Verifique a existência de memória HBM2(High Bandwidth Memory) vs. GDDR6 (Graphics DDR). HBM2 é mais rápido, mas muito mais caro.

3. **Largura de banda da memória**. Você só pode obter o máximo do seu poder de computação quando tiver largura de banda de memória suficiente. Procure por barramentos de memória ampla se estiver usando GDDR6.

Para a maioria dos usuários, basta olhar para o poder de computação. Observe que muitas GPUs oferecem diferentes tipos de aceleração. Por exemplo, os TensorCores da NVIDIA aceleram um subconjunto de operadoras em 5x. Certifique-se de que suas bibliotecas suportem isso. A memória da GPU não deve ser inferior a 4 GB(8 GB é muito melhor). Tente evitar o uso da GPU também para exibir uma GUI (em vez disso, use os gráficos integrados). Se você não puder evitá-lo, adicione 2 GB extras de RAM para segurança.

Fig. 19.5.1 compara o poder de computação de ponto flutuante de 32 bits e o preço dos vários modelos das séries GTX 900, GTX 1000 e RTX 2000. Os preços são os preços sugeridos encontrados na Wikipedia.

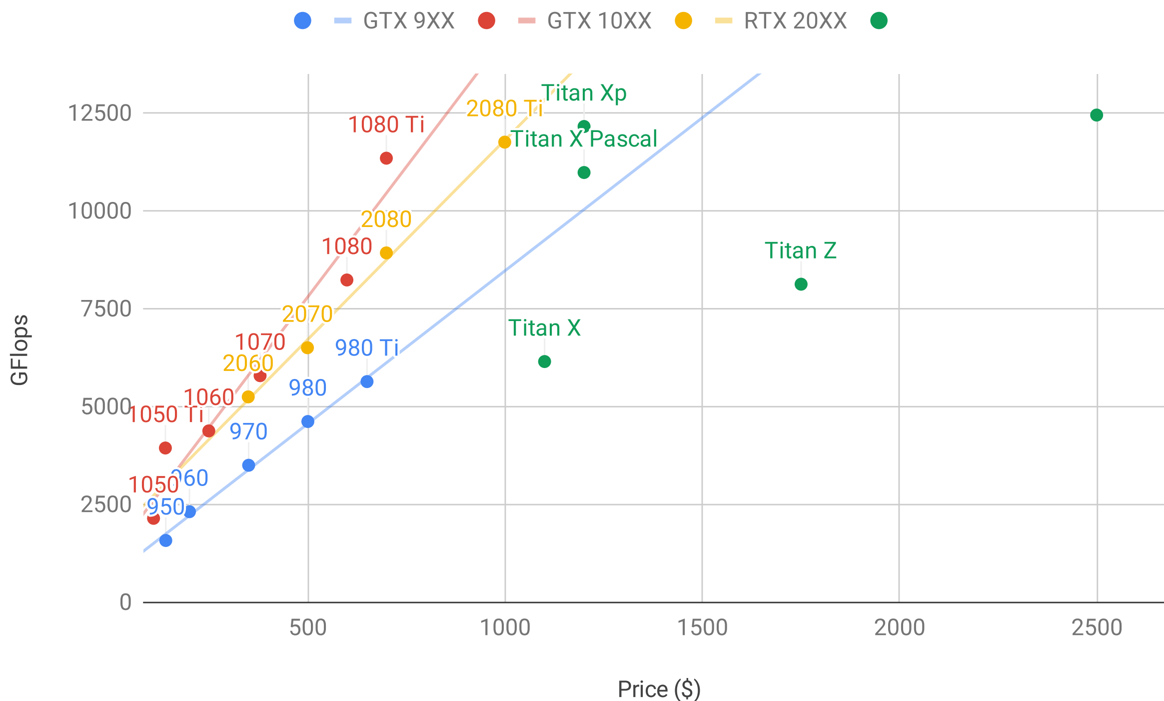


Fig. 19.5.1: Poder de computação de ponto flutuante e comparação de preços.

Podemos ver várias coisas:

1. Dentro de cada série, o preço e o desempenho são aproximadamente proporcionais. Os modelos Titan oferecem um prêmio significativo para o benefício de grandes quantidades de memória GPU. No entanto, os modelos mais novos oferecem melhor relação custo-benefício, como pode ser visto ao comparar o 980 Ti e o 1080 Ti. O preço não parece melhorar muito para a série RTX 2000. No entanto, isso se deve ao fato de que eles oferecem desempenho de baixa precisão muito superior (FP16, INT8 e INT4).
2. A relação desempenho-custo da série GTX 1000 é cerca de duas vezes maior do que a série 900.
3. Para a série RTX 2000, o preço é uma função *afim* do preço.

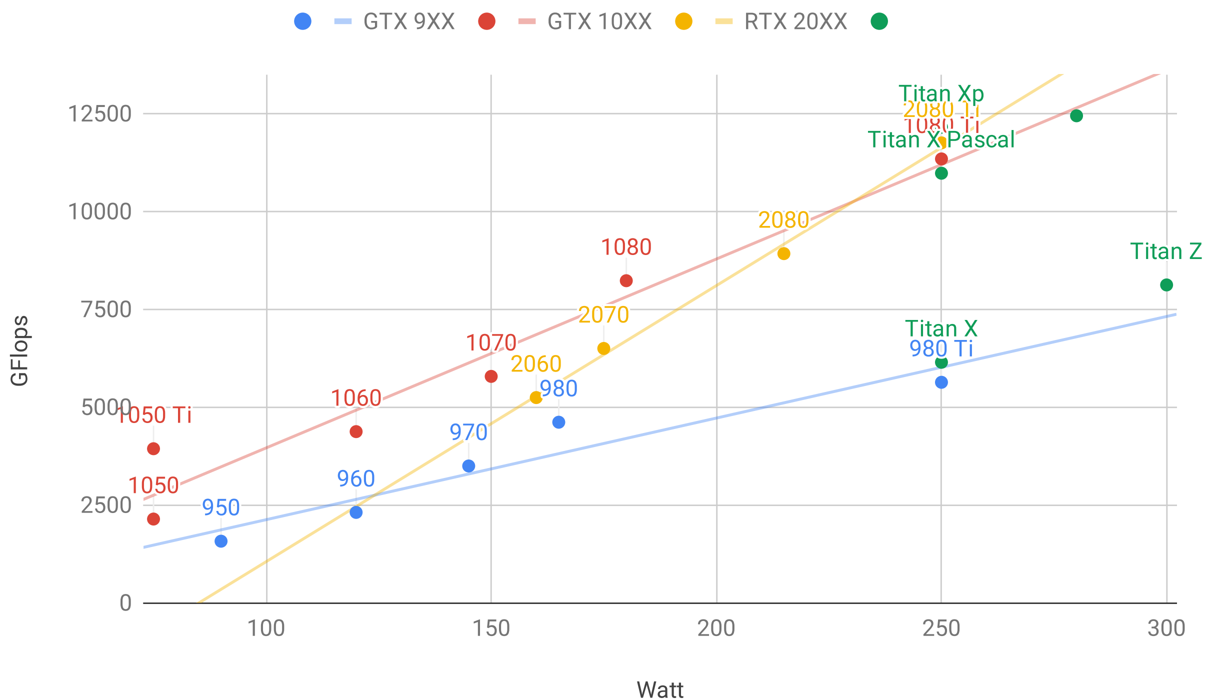


Fig. 19.5.2: Potência de computação de ponto flutuante e consumo de energia.

Fig. 19.5.2 mostra como o consumo de energia aumenta linearmente com a quantidade de computação. Em segundo lugar, as gerações posteriores são mais eficientes. Isso parece ser contradito pelo gráfico correspondente à série RTX 2000. No entanto, isso é uma consequência dos TensorCores que consomem energia desproporcionalmente.

19.5.2 Sumário

- Cuidado com a energia, faixas de barramento PCIe, velocidade de thread único da CPU e resfriamento ao construir um servidor.
- Você deve comprar a geração de GPU mais recente, se possível.
- Use a nuvem para grandes implantações.
- Os servidores de alta densidade podem não ser compatíveis com todas as GPUs. Verifique as especificações mecânicas e de resfriamento antes de comprar.
- Use FP16 ou precisão inferior para alta eficiência.

Discussão[227]

[227] https://discuss.d2l.ai/t/425

19.6 Contribuindo para este livro

Contribuições de leitores[228] nos ajudam a melhorar este livro. Se você encontrar um erro de digitação, um link desatualizado, algo onde você acha que perdemos uma citação, onde o código não parece elegante ou onde uma explicação não é clara, contribua de volta e ajude-nos a ajudar nossos leitores. Embora em livros normais o atraso entre as tiragens (e, portanto, entre as correções de digitação) possa ser medido em anos, normalmente leva horas ou dias para incorporar uma melhoria neste livro. Tudo isso é possível devido ao controle de versão e teste de integração contínua. Para fazer isso, você precisa enviar uma solicitação de pull[229] para o repositório GitHub. Quando sua solicitação pull for mesclada ao repositório de código pelo autor, você se tornará um contribuidor.

19.6.1 Pequenas alterações de texto

As contribuições mais comuns são editar uma frase ou corrigir erros de digitação. Recomendamos que você encontre o arquivo de origem no github repo[230] e edite o arquivo diretamente. Por exemplo, você pode pesquisar o arquivo através do botão Find file[231] (Fig. 19.6.1) para localizar o arquivo de origem, que é um arquivo de redução. Em seguida, você clica no botão "Editar este arquivo" no canto superior direito para fazer as alterações no arquivo de redução.

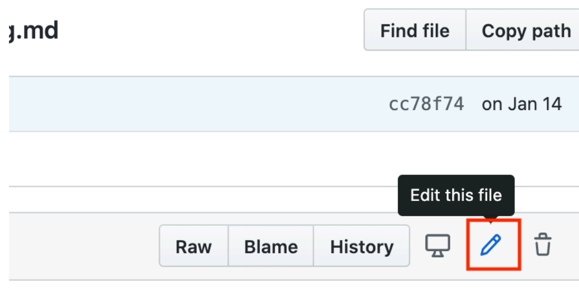


Fig. 19.6.1: Edite o arquivo no Github.

Depois de terminar, preencha as descrições das alterações no painel "Propor alteração do arquivo" na parte inferior da página e clique no botão "Propor alteração do arquivo". Ele irá redirecioná-lo para uma nova página para revisar suas alterações (Fig. 19.6.7). Se tudo estiver certo, você pode enviar uma solicitação de pull clicando no botão "Criar solicitação de pull".

[228] https://github.com/d2l-ai/d2l-en/graphs/contributors
[229] https://github.com/d2l-ai/d2l-en/pulls
[230] https://github.com/d2l-ai/d2l-en
[231] https://github.com/d2l-ai/d2l-en/find/master

19.6.2 Propor uma mudança importante

Se você planeja atualizar uma grande parte do texto ou código, precisa saber um pouco mais sobre o formato que este livro está usando. O arquivo de origem é baseado no formato markdown[232] com um conjunto de extensões por meio do d2lbook[233] pacote, como referência a equações, imagens, capítulos e citações. Você pode usar qualquer editor do Markdown para abrir esses arquivos e fazer suas alterações.

Se você deseja alterar o código, recomendamos que você use o Jupyter para abrir esses arquivos Markdown conforme descrito em Section 19.1. Para que você possa executar e testar suas alterações. Lembre-se de limpar todas as saídas antes de enviar suas alterações, nosso sistema de CI executará as seções que você atualizou para gerar saídas.

Algumas seções podem suportar múltiplas implementações de framework, você pode usar `d2lbook` para ativar um framework particular, então outras implementações de framework tornam-se blocos de código Markdown e não serão executados quando você "Executar Tudo" no Jupyter. Em outras palavras, primeiro instale `d2lbook` executando

```
pip install git+https://github.com/d2l-ai/d2l-book
```

Então, no diretório raiz de `d2l-en`, você pode ativar uma implementação particular executando um dos seguintes comandos:

```
d2lbook activate mxnet chapter_multilayer-perceptrons/mlp-scratch.md
d2lbook activate pytorch chapter_multilayer-perceptrons/mlp-scratch.md
d2lbook activate tensorflow chapter_multilayer-perceptrons/mlp-scratch.md
```

Antes de enviar suas alterações, limpe todas as saídas do bloco de código e ative todas por

```
d2lbook activate all chapter_multilayer-perceptrons/mlp-scratch.md
```

Se você adicionar um novo bloco de código não para a implementação padrão, que é MXNet, use `#@tab` para marcar este bloco na linha inicial. Por exemplo, `#@tab pytorch` para um bloco de código PyTorch, `#@tab tensorflow` para um bloco de código TensorFlow ou `#@tab all` um bloco de código compartilhado para todas as implementações. Você pode consultar d2lbook[234] para obter mais informações.

19.6.3 Adicionando uma nova seção ou uma nova implementação de estrutura

Se você deseja criar um novo capítulo, por ex. aprendizado de reforço ou adicionar implementações de novas estruturas, como TensorFlow, entre em contato com os autores primeiro, por e-mail ou usando questões do github[235].

[232] https://daringfireball.net/projects/markdown/syntax
[233] http://book.d2l.ai/user/markdown%20.html
[234] http://book.d2l.ai/user/code_tabs.html
[235] https://github.com/d2l-ai/d2l-en/issues

19.6.4 Enviando uma Mudança Principal

Sugerimos que você use o processo `git` padrão para enviar uma grande mudança. Em poucas palavras, o processo funciona conforme descrito em Fig. 19.6.2.

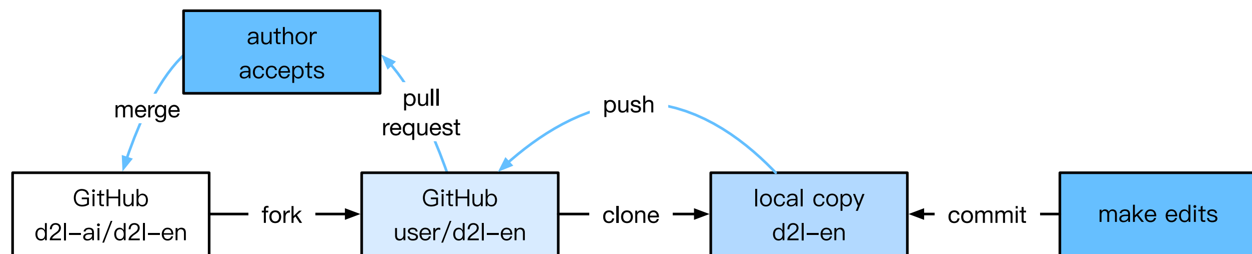


Fig. 19.6.2: Contribuindo para o livro.

Iremos acompanhá-lo detalhadamente nas etapas. Se você já estiver familiarizado com o Git, pode pular esta seção. Para concretizar, assumimos que o nome de usuário do contribuidor é "astonzhang".

Instalando Git

O livro de código aberto Git descreve como instalar o Git[236]. Isso normalmente funciona via `apt install git` no Ubuntu Linux, instalando as ferramentas de desenvolvedor Xcode no macOS ou usando o cliente de desktop do GitHub[237]. Se você não tem uma conta GitHub, você precisa se inscrever para uma.

Login no GitHub

Digite o endereço[238] do repositório de código do livro em seu navegador. Clique no botão `Fork` na caixa vermelha no canto superior direito de Fig. 19.6.3, para fazer uma cópia do repositório deste livro. Esta agora é *sua cópia* e você pode alterá-la da maneira que desejar.

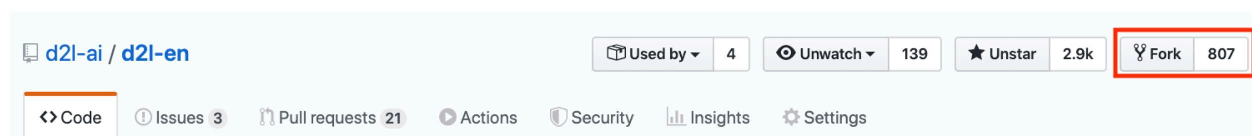


Fig. 19.6.3: A página do repositório de código.

Agora, o repositório de código deste livro será bifurcado (ou seja, copiado) para seu nome de usuário, como `astonzhang/ d2l-en` mostrado no canto superior esquerdo da imagem Fig. 19.6.4.

[236] https://git-scm.com/book/en/v2
[237] https://desktop.github.com
[238] https://github.com/d2l-ai/d2l-en/

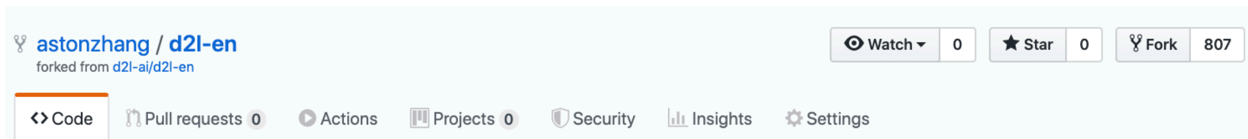


Fig. 19.6.4: Bifurque o repositório de código.

Clonando o repositório

Para clonar o repositório (ou seja, para fazer uma cópia local), precisamos obter o endereço do repositório. O botão verde em Fig. 19.6.5 exibe isso. Certifique-se de que sua cópia local esteja atualizada com o repositório principal se você decidir manter esta bifurcação por mais tempo. Por enquanto, basta seguir as instruções em *Instalação* (page 9) para começar. A principal diferença é que agora você está baixando *seu próprio fork* do repositório.

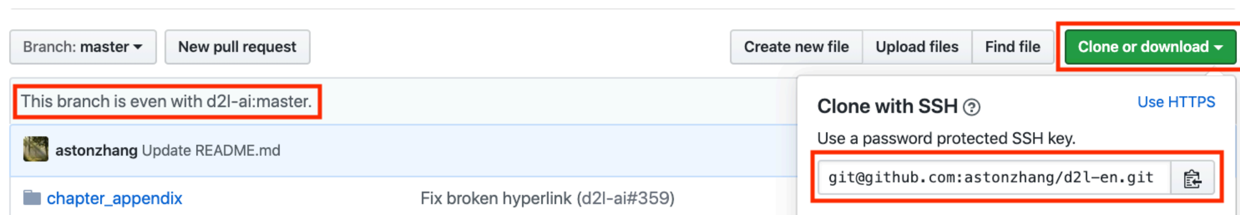


Fig. 19.6.5: Clonando Git.

```
# Replace your_github_username with your GitHub username
git clone https://github.com/your_github_username/d2l-en.git
```

Editando o livro e empurrando

Agora é hora de editar o livro. É melhor editar os blocos de notas no Jupyter seguindo as instruções em Section 19.1. Faça as alterações e verifique se estão corretas. Suponha que modificamos um erro de digitação no arquivo `~/d2l-en/chapter_appendix_tools/how-to-contribute.md`. Você pode então verificar quais arquivos você alterou:

Neste ponto, o Git irá avisar que o arquivo `chapter_appendix_tools/how-to-contribute.md` foi modificado.

```
mylaptop:d2l-en me$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter_appendix_tools/how-to-contribute.md
```

Depois de confirmar que é isso que você deseja, execute o seguinte comando:

```
git add chapter_appendix_tools/how-to-contribute.md
git commit -m 'fix typo in git documentation'
git push
```

O código alterado estará então em sua bifurcação pessoal do repositório. Para solicitar a adição de sua alteração, você deve criar uma solicitação pull para o repositório oficial do livro.

Solicitação de pull

Conforme mostrado em Fig. 19.6.6, vá para o fork do repositório no GitHub e selecione "New pull request". Isso abrirá uma tela que mostra as mudanças entre suas edições e o que está em vigor no repositório principal do livro.

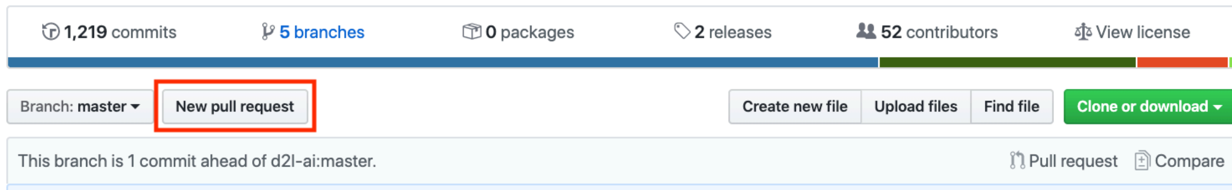


Fig. 19.6.6: Solicitação de Pull.

Enviando solicitação pull

Finalmente, envie uma solicitação de pull clicando no botão conforme mostrado em Fig. 19.6.7. Certifique-se de descrever as alterações feitas na solicitação pull. Isso tornará mais fácil para os autores revisá-lo e mesclá-lo com o livro. Dependendo das mudanças, isso pode ser aceito imediatamente, rejeitado ou, mais provavelmente, você receberá algum feedback sobre as mudanças. Depois de incorporá-los, você está pronto para prosseguir.

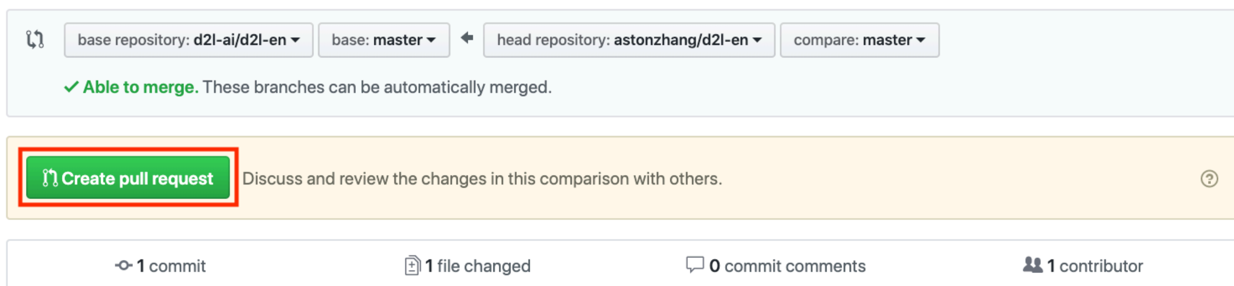


Fig. 19.6.7: Criar solicitação de pull.

Sua solicitação pull aparecerá na lista de solicitações no repositório principal. Faremos todos os esforços para processá-lo rapidamente.

19.6.5 Sumário

- Você pode usar o GitHub para contribuir com este livro.
- Você pode editar o arquivo no GitHub diretamente para pequenas alterações.
- Para uma grande mudança, bifurque o repositório, edite as coisas localmente e só contribua quando estiver pronto.
- Solicitações pull são como as contribuições estão sendo agrupadas. Tente não enviar grandes solicitações de pull, pois isso as torna difíceis de entender e incorporar. É melhor enviar vários menores.

19.6.6 Exercícios

1. Marque com estrela e bifurque o repositório `d2l-en`.
2. Encontre algum código que precise de melhorias e envie uma solicitação pull.
3. Encontre uma referência que perdemos e envie uma solicitação pull.
4. Normalmente, é uma prática melhor criar uma solicitação pull usando um novo branch. Aprenda como fazer isso com ramificação Git[239].

Discussão[240]

19.7 Documento da API `d2l`

As implementações dos seguintes membros do pacote `d2l` e seções onde eles são definidos e explicados podem ser encontrados no arquivo fonte[241].

`class d2l.torch.Accumulator(`*n*`)`

Bases: `object`

For accumulating sums over *n* variables.

`add(`**args*`)`

`reset()`

`class d2l.torch.AddNorm(`*normalized_shape, dropout, **kwargs*`)`

Bases: `torch.nn.modules.module.Module`

`forward(`*X, Y*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

[239] https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[240] https://discuss.d2l.ai/t/426
[241] https://github.com/d2l-ai/d2l-en/tree/master/d2l

training: bool

class d2l.torch.AdditiveAttention(*key_size*, *query_size*, *num_hiddens*, *dropout*, ***kwargs*)

Bases: torch.nn.modules.module.Module

forward(*queries*, *keys*, *values*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.Animator(*xlabel=None*, *ylabel=None*, *legend=None*, *xlim=None*, *ylim=None*, *xscale='linear'*, *yscale='linear'*, *fmts=('-', 'm--', 'g-.', 'r:')*, *nrows=1*, *ncols=1*, *figsize=(3.5, 2.5)*)

Bases: object

For plotting data in animation.

add(*x*, *y*)

class d2l.torch.AttentionDecoder(***kwargs*)

Bases: d2l.torch.Decoder (page 956)

The base attention-based decoder interface.

property attention_weights

training: bool

class d2l.torch.BERTEncoder(*vocab_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *max_len=1000*, *key_size=768*, *query_size=768*, *value_size=768*, ***kwargs*)

Bases: torch.nn.modules.module.Module

forward(*tokens*, *segments*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.BERTModel(*vocab_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *max_len=1000*, *key_size=768*, *query_size=768*, *value_size=768*, *hid_in_features=768*, *mlm_in_features=768*, *nsp_in_features=768*)

Bases: torch.nn.modules.module.Module

`forward`(*tokens*, *segments*, *valid_lens=None*, *pred_positions=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.BananasDataset`(*is_train*)

Bases: `torch.utils.data.dataset.Dataset`

`class d2l.torch.Benchmark`(*description='Done'*)

Bases: `object`

`class d2l.torch.Decoder`(***kwargs*)

Bases: `torch.nn.modules.module.Module`

The base decoder interface for the encoder-decoder architecture.

`forward`(*X*, *state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`init_state`(*enc_outputs*, **args*)

`training: bool`

`class d2l.torch.DotProductAttention`(*dropout*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Scaled dot product attention.

`forward`(*queries*, *keys*, *values*, *valid_lens=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.Encoder`(***kwargs*)

Bases: `torch.nn.modules.module.Module`

The base encoder interface for the encoder-decoder architecture.

forward(*X*, **args*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.EncoderBlock(*key_size*, *query_size*, *value_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *dropout*, *use_bias=False*, ***kwargs*)
Bases: `torch.nn.modules.module.Module`

forward(*X*, *valid_lens*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.EncoderDecoder(*encoder*, *decoder*, ***kwargs*)
Bases: `torch.nn.modules.module.Module`

The base class for the encoder-decoder architecture.

forward(*enc_X*, *dec_X*, **args*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.MaskLM(*vocab_size*, *num_hiddens*, *num_inputs=768*, ***kwargs*)
Bases: `torch.nn.modules.module.Module`

forward(*X*, *pred_positions*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes

care of running the registered hooks while the latter silently ignores them.

`training: bool`

class `d2l.torch.MaskedSoftmaxCELoss`(*weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = - 100, reduce=None, reduction: str = 'mean'*)

Bases: `torch.nn.modules.loss.CrossEntropyLoss`

The softmax cross-entropy loss with masks.

`forward`(*pred, label, valid_len*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`ignore_index: int`

class `d2l.torch.MultiHeadAttention`(*key_size, query_size, value_size, num_hiddens, num_heads, dropout, bias=False, **kwargs*)

Bases: `torch.nn.modules.module.Module`

`forward`(*queries, keys, values, valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

class `d2l.torch.NextSentencePred`(*num_inputs, **kwargs*)

Bases: `torch.nn.modules.module.Module`

`forward`(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

class `d2l.torch.PositionWiseFFN`(*ffn_num_input, ffn_num_hiddens, ffn_num_outputs, **kwargs*)

Bases: `torch.nn.modules.module.Module`

forward(*X*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.PositionalEncoding(*num_hiddens*, *dropout*, *max_len=1000*)
Bases: `torch.nn.modules.module.Module`

forward(*X*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.RNNModel(*rnn_layer*, *vocab_size*, ***kwargs*)
Bases: `torch.nn.modules.module.Module`

The RNN model.

begin_state(*device*, *batch_size=1*)

forward(*inputs*, *state*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.RNNModelScratch(*vocab_size*, *num_hiddens*, *device*, *get_params*, *init_state*, *forward_fn*)
Bases: `object`

A RNN Model implemented from scratch.

begin_state(*batch_size*, *device*)

class d2l.torch.RandomGenerator(*sampling_weights*)
Bases: `object`

Draw a random int in [0, n] according to n sampling weights.

draw()

class d2l.torch.Residual(*input_channels*, *num_channels*, *use_1x1conv=False*, *strides=1*)
Bases: torch.nn.modules.module.Module

The Residual block of ResNet.

forward(*X*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.SNLIDataset(*dataset*, *num_steps*, *vocab=None*)
Bases: torch.utils.data.dataset.Dataset

A customized dataset to load the SNLI dataset.

class d2l.torch.Seq2SeqEncoder(*vocab_size*, *embed_size*, *num_hiddens*, *num_layers*, *dropout=0*, ***kwargs*)
Bases: d2l.torch.Encoder (page 956)

The RNN encoder for sequence to sequence learning.

forward(*X*, **args*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.SeqDataLoader(*batch_size*, *num_steps*, *use_random_iter*, *max_tokens*)
Bases: object

An iterator to load sequence data.

class d2l.torch.Timer
Bases: object

Record multiple running times.

avg()
Return the average time.

cumsum()
Return the accumulated time.

`start()`
Start the timer.

`stop()`
Stop the timer and record the time in a list.

`sum()`
Return the sum of time.

`class d2l.torch.TokenEmbedding(`*embedding_name*`)`
Bases: `object`

Token Embedding.

`class d2l.torch.TransformerEncoder(`*vocab_size, key_size, query_size, value_size, num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens, num_heads, num_layers, dropout, use_bias=False, **kwargs*`)`
Bases: `d2l.torch.Encoder` (page 956)

`forward(`*X, valid_lens, *args*`)`
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.VOCSegDataset(`*is_train, crop_size, voc_dir*`)`
Bases: `torch.utils.data.dataset.Dataset`

A customized dataset to load VOC dataset.

`filter(`*imgs*`)`

`normalize_image(`*img*`)`

`class d2l.torch.Vocab(`*tokens=None, min_freq=0, reserved_tokens=None*`)`
Bases: `object`

Vocabulary for text.

`to_tokens(`*indices*`)`

`d2l.torch.abs(`*input, *, out=None*`)` → Tensor
Computes the absolute value of each element in `input`.

$$\text{out}_i = |\text{input}_i| \tag{19.7.1}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.abs(torch.tensor([-1, -2, 3]))
tensor([ 1,  2,  3])
```

d2l.torch.accuracy(*y_hat*, *y*)
Compute the number of correct predictions.

d2l.torch.annotate(*text*, *xy*, *xytext*)

d2l.torch.arange(*start=0*, *end*, *step=1*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a 1-D tensor of size $\left\lceil \frac{\text{end}-\text{start}}{\text{step}} \right\rceil$ with values from the interval `[start, end)` taken with common difference `step` beginning from *start*.

Note that non-integer `step` is subject to floating point rounding errors when comparing against `end`; to avoid inconsistency, we advise adding a small epsilon to `end` in such cases.

$$\text{out}_{i+1} = \text{out}_i + \text{step} \tag{19.7.2}$$

Args: start (Number): the starting value for the set of points. Default: `0`. end (Number): the ending value for the set of points step (Number): the gap between each pair of adjacent points. Default: `1`.

Keyword args: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`). If *dtype* is not given, infer the data type from the other input arguments. If any of *start*, *end*, or *stop* are floating-point, the *dtype* is inferred to be the default dtype, see `get_default_dtype()`. Otherwise, the *dtype* is inferred to be *torch.int64*.

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
>>> torch.arange(1, 4)
tensor([ 1,  2,  3])
>>> torch.arange(1, 2.5, 0.5)
tensor([ 1.0000,  1.5000,  2.0000])
```

d2l.torch.argmax(*x*, **args*, ***kwargs*)

d2l.torch.astype(*x*, **args*, ***kwargs*)

d2l.torch.batchify(*data*)

d2l.torch.bbox_to_rect(*bbox*, *color*)
Convert bounding box to matplotlib format.

d2l.torch.bleu(*pred_seq*, *label_seq*, *k*)
Compute the BLEU.

d2l.torch.box_center_to_corner(*boxes*)
Convert from (center, width, height) to (upper_left, bottom_right)

d2l.torch.box_corner_to_center(*boxes*)
Convert from (upper_left, bottom_right) to (center, width, height)

d2l.torch.box_iou(*boxes1*, *boxes2*)
Compute IOU between two sets of boxes of shape (N,4) and (M,4).

d2l.torch.build_array_nmt(*lines*, *vocab*, *num_steps*)
Transform text sequences of machine translation into minibatches.

d2l.torch.build_colormap2label()
Build an RGB color to label mapping for segmentation.

d2l.torch.concat()
cat(tensors, dim=0, *, out=None) -> Tensor

Concatenates the given sequence of `seq` tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

`torch.cat()` can be seen as an inverse operation for `torch.split()` and `torch.chunk()`.

`torch.cat()` can be best understood via examples.

Args:

tensors (sequence of Tensors): any python sequence of tensors of the same type.
Non-empty tensors provided must have the same shape, except in the cat dimension.

dim (int, optional): the dimension over which the tensors are concatenated

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 0)
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 1)
tensor([[ 0.6580, -1.0969, -0.4614,  0.6580, -1.0969, -0.4614,  0.6580,
         -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497, -0.1034, -0.5790,  0.1497, -0.1034,
         -0.5790,  0.1497]])
```

d2l.torch.copyfile(*filename*, *target_dir*)
Copy a file into a target directory.

d2l.torch.corr2d(*X*, *K*)
Compute 2D cross-correlation.

d2l.torch.cos(*input*, *, *out=None*) → Tensor
Returns a new tensor with the cosine of the elements of input.

$$\text{out}_i = \cos(\text{input}_i) \tag{19.7.3}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 1.4309,  1.2706, -0.8562,  0.9796])
>>> torch.cos(a)
tensor([ 0.1395,  0.2957,  0.6553,  0.5574])
```

d2l.torch.cosh(*input*, *, *out=None*) → Tensor
Returns a new tensor with the hyperbolic cosine of the elements of input.

$$\text{out}_i = \cosh(\text{input}_i) \tag{19.7.4}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.1632,  1.1835, -0.6979, -0.7325])
>>> torch.cosh(a)
tensor([ 1.0133,  1.7860,  1.2536,  1.2805])
```

Note: When input is on the CPU, the implementation of torch.cosh may use the Sleef library, which rounds very large results to infinity or negative infinity. See here[242] for details.

---

d2l.torch.count_corpus(*tokens*)
Count token frequencies.

class d2l.torch.defaultdict
Bases: dict

defaultdict(default_factory[, ...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

copy() → a shallow copy of D.

[242] https://sleef.org/purec.xhtml

default_factory
: Factory for default value called by __missing__().

d2l.torch.download(*name, cache_dir='../data'*)
: Download a file inserted into DATA_HUB, return the local filename.

d2l.torch.download_all()
: Download all files in the DATA_HUB.

d2l.torch.download_extract(*name, folder=None*)
: Download and extract a zip/tar file.

d2l.torch.evaluate_accuracy(*net, data_iter*)
: Compute the accuracy for a model on a dataset.

d2l.torch.evaluate_accuracy_gpu(*net, data_iter, device=None*)
: Compute the accuracy for a model on a dataset using a GPU.

d2l.torch.evaluate_loss(*net, data_iter, loss*)
: Evaluate the loss of a model on the given dataset.

d2l.torch.exp(*input, *, out=None*) → Tensor
: Returns a new tensor with the exponential of the elements of the input tensor `input`.

$$y_i = e^{x_i} \tag{19.7.5}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.exp(torch.tensor([0, math.log(2.)]))
tensor([ 1.,  2.])
```

d2l.torch.eye(*n, m=None, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False*) → Tensor
: Returns a 2-D tensor with ones on the diagonal and zeros elsewhere.

Args: n (int): the number of rows m (int, optional): the number of columns with default being `n`

Keyword arguments: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Returns: Tensor: A 2-D tensor with ones on the diagonal and zeros elsewhere

Example:

```
>>> torch.eye(3)
tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

`d2l.torch.get_centers_and_contexts(`*corpus, max_window_size*`)`

`d2l.torch.get_data_ch11(`*batch_size=10, n=1500*`)`

`d2l.torch.get_dataloader_workers()`
Use 4 processes to read the data.

`d2l.torch.get_fashion_mnist_labels(`*labels*`)`
Return text labels for the Fashion-MNIST dataset.

`d2l.torch.get_negatives(`*all_contexts, corpus, K*`)`

`d2l.torch.get_tokens_and_segments(`*tokens_a, tokens_b=None*`)`

`d2l.torch.grad_clipping(`*net, theta*`)`
Clip the gradient.

`d2l.torch.linreg(`*X, w, b*`)`
The linear regression model.

`d2l.torch.linspace(`*start, end, steps, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False*`)` → Tensor
Creates a one-dimensional tensor of size `steps` whose values are evenly spaced from `start` to `end`, inclusive. That is, the value are:

$$(\text{start}, \text{start} + \frac{\text{end} - \text{start}}{\text{steps} - 1}, \ldots, \text{start} + (\text{steps} - 2) * \frac{\text{end} - \text{start}}{\text{steps} - 1}, \text{end}) \tag{19.7.6}$$

> **Warning:** Not providing a value for `steps` is deprecated. For backwards compatibility, not providing a value for `steps` will create a tensor with 100 elements. Note that this behavior is not reflected in the documented function signature and should not be relied on. In a future PyTorch release, failing to provide a value for `steps` will throw a runtime error.

Args: start (float): the starting value for the set of points end (float): the ending value for the set of points steps (int): size of the constructed tensor

Keyword arguments: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.linspace(3, 10, steps=5)
tensor([  3.0000,   4.7500,   6.5000,   8.2500,  10.0000])
>>> torch.linspace(-10, 10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=1)
tensor([-10.])
```

`d2l.torch.load_array`(*data_arrays*, *batch_size*, *is_train=True*)
Construct a PyTorch data iterator.

`d2l.torch.load_corpus_time_machine`(*max_tokens=- 1*)
Return token indices and the vocabulary of the time machine dataset.

`d2l.torch.load_data_bananas`(*batch_size*)
Load the bananas dataset.

`d2l.torch.load_data_fashion_mnist`(*batch_size*, *resize=None*)
Download the Fashion-MNIST dataset and then load it into memory.

`d2l.torch.load_data_imdb`(*batch_size*, *num_steps=500*)

`d2l.torch.load_data_nmt`(*batch_size*, *num_steps*, *num_examples=600*)
Return the iterator and the vocabularies of the translation dataset.

`d2l.torch.load_data_ptb`(*batch_size*, *max_window_size*, *num_noise_words*)

`d2l.torch.load_data_snli`(*batch_size*, *num_steps=50*)
Download the SNLI dataset and return data iterators and vocabulary.

`d2l.torch.load_data_time_machine`(*batch_size*, *num_steps*, *use_random_iter=False*, *max_tokens=10000*)
Return the iterator and the vocabulary of the time machine dataset.

`d2l.torch.load_data_voc`(*batch_size*, *crop_size*)
Download and load the VOC2012 semantic dataset.

`d2l.torch.load_data_wiki`(*batch_size*, *max_len*)

`d2l.torch.log`(*input*, ***, *out=None*) → Tensor
Returns a new tensor with the natural logarithm of the elements of `input`.

$$y_i = \log_e(x_i) \tag{19.7.7}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(5)
>>> a
tensor([-0.7168, -0.5471, -0.8933, -1.4428, -0.1190])
>>> torch.log(a)
tensor([ nan,  nan,  nan,  nan,  nan])
```

`d2l.torch.masked_softmax`(*X*, *valid_lens*)
Perform softmax operation by masking elements on the last axis.

`d2l.torch.match_anchor_to_bbox`(*ground_truth*, *anchors*, *device*, *iou_threshold=0.5*)
Assign ground-truth bounding boxes to anchor boxes similar to them.

`d2l.torch.matmul`(*input*, *other*, ***, *out=None*) → Tensor
Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are broadcasted (and thus must be broadcastable). For example, if `input` is a $(j \times 1 \times n \times n)$ tensor and `other` is a $(k \times n \times n)$ tensor, out will be a $(j \times k \times n \times n)$ tensor.

  Note that the broadcasting logic only looks at the batch dimensions when determining if the inputs are broadcastable, and not the matrix dimensions. For example, if `input` is a $(j \times 1 \times n \times m)$ tensor and `other` is a $(k \times m \times p)$ tensor, these inputs are valid for broadcasting even though the final two dimensions (i.e. the matrix dimensions) are different. out will be a $(j \times k \times n \times p)$ tensor.

This operator supports TensorFloat32.

Note: The 1-dimensional dot product version of this function does not support an `out` parameter.

Arguments: input (Tensor): the first tensor to be multiplied other (Tensor): the second tensor to be multiplied

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> # vector x vector
>>> tensor1 = torch.randn(3)
>>> tensor2 = torch.randn(3)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([])
>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([3])
>>> # batched matrix x broadcasted vector
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3])
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
>>> # batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

`d2l.torch.meshgrid(`**tensors*`)`

Take N tensors, each of which can be either scalar or 1-dimensional vector, and create N N-dimensional grids, where the i $^{\text{th}}$ grid is defined by expanding the i $^{\text{th}}$ input over dimensions defined by other inputs.

Args:

tensors (list of Tensor): list of scalars or 1 dimensional tensors. Scalars will be treated as tensors of size $(1,)$ automatically

Returns: seq (sequence of Tensors): If the input has k tensors of size $(N_1,), (N_2,), \ldots, (N_k,)$, then the output would also have k tensors, where all tensors are of size $(N_1, N_2, \ldots, N_k)$.

Example:

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([4, 5, 6])
>>> grid_x, grid_y = torch.meshgrid(x, y)
>>> grid_x
tensor([[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3]])
>>> grid_y
tensor([[4, 5, 6],
        [4, 5, 6],
        [4, 5, 6]])
```

`d2l.torch.multibox_detection(`*cls_probs, offset_preds, anchors, nms_threshold=0.5, pos_threshold=0.00999999978*`)`

`d2l.torch.multibox_prior(`*data, sizes, ratios*`)`

d2l.torch.multibox_target(*anchors*, *labels*)

d2l.torch.nms(*boxes*, *scores*, *iou_threshold*)

d2l.torch.normal(*mean*, *std*, ***, *generator=None*, *out=None*) → Tensor
Returns a tensor of random numbers drawn from separate normal distributions whose mean and standard deviation are given.

The `mean` is a tensor with the mean of each output element's normal distribution

The `std` is a tensor with the standard deviation of each output element's normal distribution

The shapes of `mean` and `std` don't need to match, but the total number of elements in each tensor need to be the same.

Note: When the shapes do not match, the shape of `mean` is used as the shape for the returned output tensor

---

Args: mean (Tensor): the tensor of per-element means std (Tensor): the tensor of per-element standard deviations

Keyword args: generator (`torch.Generator`, optional): a pseudorandom number generator for sampling out (Tensor, optional): the output tensor.

Example:

```
>>> torch.normal(mean=torch.arange(1., 11.), std=torch.arange(1, 0, -0.1))
tensor([  1.0425,   3.5672,   2.7969,   4.2925,   4.7229,   6.2134,
          8.0505,   8.1408,   9.0563,  10.0566])
```

d2l.torch.normal(*mean=0.0*, *std*, ***, *out=None*) → Tensor

Similar to the function above, but the means are shared among all drawn elements.

Args: mean (float, optional): the mean for all distributions std (Tensor): the tensor of per-element standard deviations

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.normal(mean=0.5, std=torch.arange(1., 6.))
tensor([-1.2793, -1.0732, -2.0687,  5.1177, -1.2303])
```

d2l.torch.normal(*mean*, *std=1.0*, ***, *out=None*) → Tensor

Similar to the function above, but the standard-deviations are shared among all drawn elements.

Args: mean (Tensor): the tensor of per-element means std (float, optional): the standard deviation for all distributions

Keyword args: out (Tensor, optional): the output tensor

Example:

```
>>> torch.normal(mean=torch.arange(1., 6.))
tensor([ 1.1552,  2.6148,  2.6535,  5.8318,  4.2361])
```

`d2l.torch.normal(`*mean, std, size, *, out=None*`)` → Tensor

Similar to the function above, but the means and standard deviations are shared among all drawn elements. The resulting tensor has size given by `size` (page 974).

Args: mean (float): the mean for all distributions std (float): the standard deviation for all distributions size (int...): a sequence of integers defining the shape of the output tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.normal(2, 3, size=(1, 4))
tensor([[-1.3987, -1.9544,  3.6048,  0.7909]])
```

`d2l.torch.numpy(`*x, *args, **kwargs*`)`

`d2l.torch.offset_boxes(`*anchors, assigned_bb, eps=1e-06*`)`

`d2l.torch.offset_inverse(`*anchors, offset_preds*`)`

`d2l.torch.ones(`**size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False*`)` → Tensor

Returns a tensor filled with the scalar value *1*, with the shape defined by the variable argument `size` (page 974).

Args:

size (int...): a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword arguments: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.ones(2, 3)
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
```

(continues on next page)

(continued from previous page)

```
>>> torch.ones(5)
tensor([ 1.,  1.,  1.,  1.,  1.])
```

d2l.torch.plot(*X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None, ylim=None, xscale='linear', yscale='linear', fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None*)
Plot data points.

d2l.torch.predict_ch3(*net, test_iter, n=6*)
Predict labels (defined in Chapter 3).

d2l.torch.predict_ch8(*prefix, num_preds, net, vocab, device*)
Generate new characters following the *prefix*.

d2l.torch.predict_sentiment(*net, vocab, sentence*)

d2l.torch.predict_seq2seq(*net, src_sentence, src_vocab, tgt_vocab, num_steps, device, save_attention_weights=False*)
Predict for sequence to sequence.

d2l.torch.predict_snli(*net, vocab, premise, hypothesis*)

d2l.torch.preprocess_nmt(*text*)
Preprocess the English-French dataset.

d2l.torch.rand(**size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False*) → Tensor
Returns a tensor filled with random numbers from a uniform distribution on the interval $[0, 1)$

The shape of the tensor is defined by the variable argument `size` (page 974).

Args:

size (int...): a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword args: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.rand(4)
tensor([ 0.5204,  0.2503,  0.3525,  0.5673])
>>> torch.rand(2, 3)
tensor([[ 0.8237,  0.5781,  0.6879],
        [ 0.3816,  0.7249,  0.0998]])
```

`d2l.torch.read_csv_labels(`*fname*`)`
: Read fname to return a name to label dictionary.

`d2l.torch.read_data_bananas(`*is_train=True*`)`
: Read the bananas dataset images and labels.

`d2l.torch.read_data_nmt()`
: Load the English-French dataset.

`d2l.torch.read_imdb(`*data_dir, is_train*`)`

`d2l.torch.read_ptb()`

`d2l.torch.read_snli(`*data_dir, is_train*`)`
: Read the SNLI dataset into premises, hypotheses, and labels.

`d2l.torch.read_time_machine()`
: Load the time machine dataset into a list of text lines.

`d2l.torch.read_voc_images(`*voc_dir, is_train=True*`)`
: Read all VOC feature and label images.

`d2l.torch.reduce_sum(`*x, *args, **kwargs*`)`

`d2l.torch.reorg_test(`*data_dir*`)`

`d2l.torch.reorg_train_valid(`*data_dir, labels, valid_ratio*`)`

`d2l.torch.reshape(`*x, *args, **kwargs*`)`

`d2l.torch.resnet18(`*num_classes, in_channels=1*`)`
: A slightly modified ResNet-18 model.

`d2l.torch.seq_data_iter_random(`*corpus, batch_size, num_steps*`)`
: Generate a minibatch of subsequences using random sampling.

`d2l.torch.seq_data_iter_sequential(`*corpus, batch_size, num_steps*`)`
: Generate a minibatch of subsequences using sequential partitioning.

`d2l.torch.sequence_mask(`*X, valid_len, value=0*`)`
: Mask irrelevant entries in sequences.

`d2l.torch.set_axes(`*axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend*`)`
: Set the axes for matplotlib.

`d2l.torch.set_figsize(`*figsize=(3.5, 2.5)*`)`
: Set the figure size for matplotlib.

`d2l.torch.sgd(`*params, lr, batch_size*`)`
: Minibatch stochastic gradient descent.

`d2l.torch.show_bboxes(`*axes, bboxes, labels=None, colors=None*`)`
: Show bounding boxes.

`d2l.torch.show_heatmaps(`*matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5), cmap='Reds'*`)`

`d2l.torch.show_images(`*imgs, num_rows, num_cols, titles=None, scale=1.5*`)`
Plot a list of images.

`d2l.torch.show_trace_2d(`*f, results*`)`
Show the trace of 2D variables during optimization.

`d2l.torch.sin(`*input, *, out=None*`)` → Tensor
Returns a new tensor with the sine of the elements of `input`.

$$\text{out}_i = \sin(\text{input}_i) \tag{19.7.8}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.5461,  0.1347, -2.7266, -0.2746])
>>> torch.sin(a)
tensor([-0.5194,  0.1343, -0.4032, -0.2711])
```

`d2l.torch.sinh(`*input, *, out=None*`)` → Tensor
Returns a new tensor with the hyperbolic sine of the elements of `input`.

$$\text{out}_i = \sinh(\text{input}_i) \tag{19.7.9}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.5380, -0.8632, -0.1265,  0.9399])
>>> torch.sinh(a)
tensor([ 0.5644, -0.9744, -0.1268,  1.0845])
```

Note: When `input` is on the CPU, the implementation of torch.sinh may use the Sleef library, which rounds very large results to infinity or negative infinity. See here[243] for details.

`d2l.torch.size(`*x, *args, **kwargs*`)`

`d2l.torch.split_batch(`*X, y, devices*`)`
Split X and y into multiple devices.

`d2l.torch.squared_loss(`*y_hat, y*`)`
Squared loss.

`d2l.torch.stack(`*tensors, dim=0, *, out=None*`)` → Tensor
Concatenates a sequence of tensors along a new dimension.

All tensors need to be of the same size.

[243] https://sleef.org/purec.xhtml

Arguments: tensors (sequence of Tensors): sequence of tensors to concatenate dim (int): dimension to insert. Has to be between 0 and the number

of dimensions of concatenated tensors (inclusive)

Keyword args: out (Tensor, optional): the output tensor.

`d2l.torch.subsampling(`*sentences, vocab*`)`

`d2l.torch.synthetic_data(`*w, b, num_examples*`)`
Generate y = Xw + b + noise.

`d2l.torch.tanh(`*input, *, out=None*`)` → Tensor
Returns a new tensor with the hyperbolic tangent of the elements of `input`.

$$\text{out}_i = \tanh(\text{input}_i) \tag{19.7.10}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.8986, -0.7279,  1.1745,  0.2611])
>>> torch.tanh(a)
tensor([ 0.7156, -0.6218,  0.8257,  0.2553])
```

`d2l.torch.tensor(`*data, *, dtype=None, device=None, requires_grad=False, pin_memory=False*`)` → Tensor
Constructs a tensor with `data`.

> **Warning:** `torch.tensor()` always copies `data`. If you have a Tensor `data` and want to avoid a copy, use `torch.Tensor.requires_grad_()` or `torch.Tensor.detach()`. If you have a NumPy `ndarray` and want to avoid a copy, use `torch.as_tensor()`.

> **Warning:** When data is a tensor *x*, `torch.tensor()` reads out 'the data' from whatever it is passed, and constructs a leaf variable. Therefore `torch.tensor(x)` is equivalent to `x.clone().detach()` and `torch.tensor(x, requires_grad=True)` is equivalent to `x.clone().detach().requires_grad_(True)`. The equivalents using `clone()` and `detach()` are recommended.

Args:

data (array_like): Initial data for the tensor. Can be a list, tuple, NumPy `ndarray`, scalar, and other types.

Keyword args:

dtype (`torch.dtype`**, optional): the desired data type of returned tensor.** Default: if `None`, infers data type from `data`.

device (`torch.device`**, optional): the desired device of returned tensor.** Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

pin_memory (bool, optional): If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: `False`.

Example:

```
>>> torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])

>>> torch.tensor([0, 1])  # Type inference on data
tensor([ 0,  1])

>>> torch.tensor([[0.11111, 0.222222, 0.3333333]],
...              dtype=torch.float64,
...              device=torch.device('cuda:0'))  # creates a torch.cuda.DoubleTensor
tensor([[ 0.1111,  0.2222,  0.3333]], dtype=torch.float64, device='cuda:0')

>>> torch.tensor(3.14159)  # Create a scalar (zero-dimensional tensor)
tensor(3.1416)

>>> torch.tensor([])  # Create an empty tensor (of size (0,))
tensor([])
```

`d2l.torch.to`(*x, *args, **kwargs*)

`d2l.torch.tokenize`(*lines, token='word'*)
Split text lines into word or character tokens.

`d2l.torch.tokenize_nmt`(*text, num_examples=None*)
Tokenize the English-French dataset.

`d2l.torch.train_2d`(*trainer, steps=20*)
Optimize a 2-dim objective function with a customized trainer.

`d2l.torch.train_batch_ch13`(*net, X, y, loss, trainer, devices*)

`d2l.torch.train_ch11`(*trainer_fn, states, hyperparams, data_iter, feature_dim, num_epochs=2*)

`d2l.torch.train_ch13`(*net, train_iter, test_iter, loss, trainer, num_epochs, devices=[device(type='cuda', index=0), device(type='cuda', index=1), device(type='cuda', index=2), device(type='cuda', index=3)]*)

`d2l.torch.train_ch3`(*net, train_iter, test_iter, loss, num_epochs, updater*)
Train a model (defined in Chapter 3).

`d2l.torch.train_ch6`(*net, train_iter, test_iter, num_epochs, lr, device*)
Train a model with a GPU (defined in Chapter 6).

`d2l.torch.train_ch8(`*net, train_iter, vocab, lr, num_epochs, device, use_random_iter=False*`)`
Train a model (defined in Chapter 8).

`d2l.torch.train_concise_ch11(`*trainer_fn, hyperparams, data_iter, num_epochs=4*`)`

`d2l.torch.train_epoch_ch3(`*net, train_iter, loss, updater*`)`
The training loop defined in Chapter 3.

`d2l.torch.train_epoch_ch8(`*net, train_iter, loss, updater, device, use_random_iter*`)`
Train a net within one epoch (defined in Chapter 8).

`d2l.torch.train_seq2seq(`*net, data_iter, lr, num_epochs, tgt_vocab, device*`)`
Train a model for sequence to sequence.

`d2l.torch.transpose(`*x, *args, **kwargs*`)`

`d2l.torch.transpose_output(`*X, num_heads*`)`
Reverse the operation of *transpose_qkv*

`d2l.torch.transpose_qkv(`*X, num_heads*`)`

`d2l.torch.truncate_pad(`*line, num_steps, padding_token*`)`
Truncate or pad sequences.

`d2l.torch.try_all_gpus()`
Return all available GPUs, or [cpu(),] if no GPU exists.

`d2l.torch.try_gpu(`*i=0*`)`
Return gpu(i) if exists, otherwise return cpu().

`d2l.torch.update_D(`*X, Z, net_D, net_G, loss, trainer_D*`)`
Update discriminator.

`d2l.torch.update_G(`*Z, net_D, net_G, loss, trainer_G*`)`
Update generator.

`d2l.torch.use_svg_display()`
Use the svg format to display a plot in Jupyter.

`d2l.torch.voc_label_indices(`*colormap, colormap2label*`)`
Map an RGB color to a label.

`d2l.torch.voc_rand_crop(`*feature, label, height, width*`)`
Randomly crop for both feature and label images.

`d2l.torch.zeros(`**size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False*`)` → Tensor
Returns a tensor filled with the scalar value *0*, with the shape defined by the variable argument `size` (page 974).

Args:

size (int...): a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword args: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.zeros(2, 3)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

>>> torch.zeros(5)
tensor([ 0.,  0.,  0.,  0.,  0.])
```

Bibliography

Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining* (pp. 123–132).

Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE transactions on Information Theory, 46*(2), 325–343.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: speeded up robust features. *European conference on computer vision* (pp. 404–417).

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research, 3*(Feb), 1137–1155.

Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation, 7*(1), 108–116.

Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.

Brown, N., & Sandholm, T. (2017). Libratus: the superhuman ai for no-limit poker. *IJCAI* (pp. 5226–5228).

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L., & Roossin, P. (1988). A statistical approach to language translation. *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*.

Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence, 134*(1-2), 57–83.

Canny, J. (1987). A computational approach to edge detection. *Readings in computer vision* (pp. 184–203). Elsevier.

Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. (2017). Semeval-2017 task 1: semantic textual similarity multilingual and crosslingual focused evaluation. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)* (pp. 1–14).

Cheng, J., Dong, L., & Lapata, M. (2016). Long short-term memory-networks for machine reading. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (pp. 551–561).

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Csiszár, I. (2008). Axiomatic characterizations of information measures. *Entropy*, *10*(3), 261–273.

De Cock, D. (2011). Ames, iowa: alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, *19*(3).

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., … Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* (pp. 205–220).

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., … others. (2021). An image is worth 16x16 words: transformers for image recognition at scale. *International Conference on Learning Representations*.

Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. *Sequential Monte Carlo methods in practice* (pp. 3–14). Springer.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(Jul), 2121–2159.

Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a step-size. *Conference on Learning Theory* (pp. 658–695).

Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2414–2423).

Ginibre, J. (1965). Statistical ensembles of complex, quaternion, and real matrices. *Journal of Mathematical Physics*, *6*(3), 440–449.

Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Goh, G. (2017). Why momentum really works. *Distill*. URL: http://distill.pub/2017/momentum, doi:10.23915/distill.00006[244]

[244] https://doi.org/10.23915/distill.00006

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, *35*(12), 61–71.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems* (pp. 2672–2680).

Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, *18*(5-6), 602–610.

Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*.

Hazan, E., Rakhlin, A., & Bartlett, P. L. (2008). Adaptive online gradient descent. *Advances in Neural Information Processing Systems* (pp. 65–72).

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., & others (2001). *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in neural information processing systems* (pp. 689–696).

Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7132–7141).

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. *2008 Eighth IEEE International Conference on Data Mining* (pp. 263–272).

Hu, Z., Lee, R. K.-W., & Aggarwal, C. C. (2020). Text style transfer: a review and experiment evaluation. *arXiv preprint arXiv:2010.12742*.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).

Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* (pp. 1945–1953).

Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. Vol. 5. GMD-Forschungszentrum Informationstechnik Bonn.

James, W. (2007). *The principles of psychology*. Vol. 1. Cosimo, Inc.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., … others. (2018). Highly scalable deep learning training system with mixed-precision: training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., … others. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Kolter, Z. (2008). Linear algebra review and reference. *Available online: http*.

Koren, Y. (2009). Collaborative filtering with temporal dynamics. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 447–456).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (pp. 1097–1105).

Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., & others. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.

Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., … Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th $\$USENIX$\$ Symposium on Operating Systems Design and Implementation ($\$OSDI$\$ 14)* (pp. 583–598).

Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).

Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., … others. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*.

Lin, Z., Feng, M., Santos, C. N. d., Yu, M., Xiang, B., Zhou, B., & Bengio, Y. (2017). A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*.

Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: single shot multibox detector. *European conference on computer vision* (pp. 21–37).

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431–3440).

Loshchilov, I., & Hutter, F. (2016). Sgdr: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, *60*(2), 91–110.

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint*.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1* (pp. 142–150).

McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in translation: contextualized word vectors. *Advances in Neural Information Processing Systems* (pp. 6294–6305).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* (pp. 3111–3119).

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., … Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 2430–2439).

Mnih, V., Heess, N., Graves, A., & others. (2014). Recurrent models of visual attention. *Advances in neural information processing systems* (pp. 2204–2212).

Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & Its Applications*, *9*(1), 141–142.

Nesterov, Y., & Vial, J.-P. (2000). *Confidence level solutions for stochastic programming, Stochastic Programming E-Print Series*.

Nesterov, Y. (2018). *Lectures on convex optimization*. Vol. 137. Springer.

Neyman, J. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, *236*(767), 333–380.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311–318).

Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.

Park, T., Liu, M.-Y., Wang, T.-C., & Zhu, J.-Y. (2019). Semantic image synthesis with spatially-adaptive normalization. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2337–2346).

Paulus, R., Xiong, C., & Socher, R. (2017). A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*.

Pennington, J., Schoenholz, S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in neural information processing systems* (pp. 4785–4795).

Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543).

Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

Peters, M., Ammar, W., Bhagavatula, C., & Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1756–1765).

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (pp. 2227–2237).

Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark*, *7*(15), 510.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

Quadrana, M., Cremonesi, P., & Jannach, D. (2018). Sequence-aware recommender systems. *ACM Computing Surveys (CSUR)*, *51*(4), 66.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.

Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99).

Rumelhart, D. E., Hinton, G. E., Williams, R. J., & others. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).

Sarwar, B. M., Karypis, G., Konstan, J. A., Riedl, J., & others. (2001). Item-based collaborative filtering recommendation algorithms. *Www, 1*, 285–295.

Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 253–260).

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing, 45*(11), 2673–2681.

Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

Shannon, C. E. (1948 , 7). A mathematical theory of communication. *The Bell System Technical Journal, 27*(3), 379–423.

Shao, H., Yao, S., Sun, D., Zhang, A., Liu, S., Liu, D., … Abdelzaher, T. (2020). Controlvae: controllable variational autoencoder. *Proceedings of the 37th International Conference on Machine Learning*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., … others. (2016). Mastering the game of go with deep neural networks and tree search. *nature, 529*(7587), 484.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Smola, A., & Narayanamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment, 3*(1-2), 703–710.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929–1958.

Strang, G. (1993). *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA.

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence, 2009*.

Sukhbaatar, S., Weston, J., Fergus, R., & others. (2015). End-to-end memory networks. *Advances in neural information processing systems* (pp. 2440–2448).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning* (pp. 1139–1147).

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems* (pp. 3104–3112).

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).

Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*.

Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: a survey. *arXiv preprint arXiv:2009.06732*.

Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, *104*(2), 154–171.

Van Loan, C. F., & Golub, G. H. (1983). *Matrix computations*. Johns Hopkins University Press.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems* (pp. 5998–6008).

Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *NETWORKS*, *2*(3), 2–3.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the gpu. *ACM SIGPLAN Notices* (p. 11).

Warstadt, A., Singh, A., & Bowman, S. R. (2019). Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, *7*, 625–641.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 359–372.

Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681–688).

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.

Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math* (pp. 325–327).

Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. *Proceedings of the tenth ACM international conference on web search and data mining* (pp. 495–503).

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., … others. (2016). Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018). Dynamical isometry and a mean field theory of cnns: how to train 10,000-layer vanilla convolutional neural networks. *International Conference on Machine Learning* (pp. 5393–5402).

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).

Ye, M., Yin, P., Lee, W.-C., & Lee, D.-L. (2011). Exploiting geographical influence for collaborative point-of-interest recommendation. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (pp. 325–334).

You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, A., Tay, Y., Zhang, S., Chan, A., Luu, A. T., Hui, S. C., & Fu, J. (2021). Beyond fully-connected layers with quaternions: parameterization of hypercomplex multiplications with 1/n parameters. *International Conference on Learning Representations*.

Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys (CSUR)*, *52*(1), 5.

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE international conference on computer vision* (pp. 2223–2232).

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning books and movies: towards story-like visual explanations by watching movies and reading books. *Proceedings of the IEEE international conference on computer vision* (pp. 19–27).

Python Module Index

d

`d2l.torch`, 954

Index

A

abs() (*in module d2l.torch*), 961
Accumulator (*class in d2l.torch*), 954
accuracy() (*in module d2l.torch*), 961
add() (*d2l.torch.Accumulator method*), 954
add() (*d2l.torch.Animator method*), 955
AdditiveAttention (*class in d2l.torch*), 955
AddNorm (*class in d2l.torch*), 954
Animator (*class in d2l.torch*), 955
annotate() (*in module d2l.torch*), 962
arange() (*in module d2l.torch*), 962
argmax() (*in module d2l.torch*), 962
astype() (*in module d2l.torch*), 962
attention_weights (*d2l.torch.AttentionDecoder property*), 955
AttentionDecoder (*class in d2l.torch*), 955
avg() (*d2l.torch.Timer method*), 960

B

BananasDataset (*class in d2l.torch*), 956
batchify() (*in module d2l.torch*), 962
bbox_to_rect() (*in module d2l.torch*), 962
begin_state() (*d2l.torch.RNNModel method*), 959
begin_state() (*d2l.torch.RNNModelScratch method*), 959
Benchmark (*class in d2l.torch*), 956
BERTEncoder (*class in d2l.torch*), 955
BERTModel (*class in d2l.torch*), 955
bleu() (*in module d2l.torch*), 962
box_center_to_corner() (*in module d2l.torch*), 963
box_corner_to_center() (*in module d2l.torch*), 963
box_iou() (*in module d2l.torch*), 963
build_array_nmt() (*in module d2l.torch*), 963
build_colormap2label() (*in module d2l.torch*), 963

C

concat() (*in module d2l.torch*), 963
copy() (*d2l.torch.defaultdict method*), 964
copyfile() (*in module d2l.torch*), 963
corr2d() (*in module d2l.torch*), 963
cos() (*in module d2l.torch*), 964
cosh() (*in module d2l.torch*), 964
count_corpus() (*in module d2l.torch*), 964
cumsum() (*d2l.torch.Timer method*), 960

D

d2l.torch
    module, 954
Decoder (*class in d2l.torch*), 956
default_factory (*d2l.torch.defaultdict attribute*), 964
defaultdict (*class in d2l.torch*), 964
DotProductAttention (*class in d2l.torch*), 956
download() (*in module d2l.torch*), 965
download_all() (*in module d2l.torch*), 965
download_extract() (*in module d2l.torch*), 965
draw() (*d2l.torch.RandomGenerator method*), 959

E

Encoder (*class in d2l.torch*), 956
EncoderBlock (*class in d2l.torch*), 957
EncoderDecoder (*class in d2l.torch*), 957
evaluate_accuracy() (*in module d2l.torch*), 965
evaluate_accuracy_gpu() (*in module d2l.torch*), 965
evaluate_loss() (*in module d2l.torch*), 965
exp() (*in module d2l.torch*), 965
eye() (*in module d2l.torch*), 965

F

filter() (*d2l.torch.VOCSegDataset method*), 961
forward() (*d2l.torch.AdditiveAttention method*), 955
forward() (*d2l.torch.AddNorm method*), 954
forward() (*d2l.torch.BERTEncoder method*), 955
forward() (*d2l.torch.BERTModel method*), 955
forward() (*d2l.torch.Decoder method*), 956
forward() (*d2l.torch.DotProductAttention method*), 956
forward() (*d2l.torch.Encoder method*), 956

forward() (*d2l.torch.EncoderBlock method*), 957
forward() (*d2l.torch.EncoderDecoder method*), 957
forward() (*d2l.torch.MaskedSoftmaxCELoss method*), 958
forward() (*d2l.torch.MaskLM method*), 957
forward() (*d2l.torch.MultiHeadAttention method*), 958
forward() (*d2l.torch.NextSentencePred method*), 958
forward() (*d2l.torch.PositionalEncoding method*), 959
forward() (*d2l.torch.PositionWiseFFN method*), 958
forward() (*d2l.torch.Residual method*), 960
forward() (*d2l.torch.RNNModel method*), 959
forward() (*d2l.torch.Seq2SeqEncoder method*), 960
forward() (*d2l.torch.TransformerEncoder method*), 961

G

get_centers_and_contexts() (*in module d2l.torch*), 966
get_data_ch11() (*in module d2l.torch*), 966
get_dataloader_workers() (*in module d2l.torch*), 966
get_fashion_mnist_labels() (*in module d2l.torch*), 966
get_negatives() (*in module d2l.torch*), 966
get_tokens_and_segments() (*in module d2l.torch*), 966
grad_clipping() (*in module d2l.torch*), 966

I

ignore_index (*d2l.torch.MaskedSoftmaxCELoss attribute*), 958
init_state() (*d2l.torch.Decoder method*), 956

L

linreg() (*in module d2l.torch*), 966
linspace() (*in module d2l.torch*), 966
load_array() (*in module d2l.torch*), 967
load_corpus_time_machine() (*in module d2l.torch*), 967
load_data_bananas() (*in module d2l.torch*), 967
load_data_fashion_mnist() (*in module d2l.torch*), 967
load_data_imdb() (*in module d2l.torch*), 967
load_data_nmt() (*in module d2l.torch*), 967
load_data_ptb() (*in module d2l.torch*), 967
load_data_snli() (*in module d2l.torch*), 967
load_data_time_machine() (*in module d2l.torch*), 967
load_data_voc() (*in module d2l.torch*), 967
load_data_wiki() (*in module d2l.torch*), 967
log() (*in module d2l.torch*), 967

M

masked_softmax() (*in module d2l.torch*), 968
MaskedSoftmaxCELoss (*class in d2l.torch*), 958
MaskLM (*class in d2l.torch*), 957
match_anchor_to_bbox() (*in module d2l.torch*), 968
matmul() (*in module d2l.torch*), 968
meshgrid() (*in module d2l.torch*), 969
module
    d2l.torch, 954
multibox_detection() (*in module d2l.torch*), 969
multibox_prior() (*in module d2l.torch*), 969
multibox_target() (*in module d2l.torch*), 970
MultiHeadAttention (*class in d2l.torch*), 958

N

NextSentencePred (*class in d2l.torch*), 958
nms() (*in module d2l.torch*), 970
normal() (*in module d2l.torch*), 970, 971
normalize_image() (*d2l.torch.VOCSegDataset method*), 961
numpy() (*in module d2l.torch*), 971

O

offset_boxes() (*in module d2l.torch*), 971
offset_inverse() (*in module d2l.torch*), 971
ones() (*in module d2l.torch*), 971

P

plot() (*in module d2l.torch*), 972
PositionalEncoding (*class in d2l.torch*), 959
PositionWiseFFN (*class in d2l.torch*), 958
predict_ch3() (*in module d2l.torch*), 972
predict_ch8() (*in module d2l.torch*), 972
predict_sentiment() (*in module d2l.torch*), 972
predict_seq2seq() (*in module d2l.torch*), 972
predict_snli() (*in module d2l.torch*), 972
preprocess_nmt() (*in module d2l.torch*), 972

R

rand() (*in module d2l.torch*), 972
RandomGenerator (*class in d2l.torch*), 959
read_csv_labels() (*in module d2l.torch*), 973
read_data_bananas() (*in module d2l.torch*), 973

read_data_nmt() (*in module d2l.torch*), 973
read_imdb() (*in module d2l.torch*), 973
read_ptb() (*in module d2l.torch*), 973
read_snli() (*in module d2l.torch*), 973
read_time_machine() (*in module d2l.torch*), 973
read_voc_images() (*in module d2l.torch*), 973
reduce_sum() (*in module d2l.torch*), 973
reorg_test() (*in module d2l.torch*), 973
reorg_train_valid() (*in module d2l.torch*), 973
reset() (*d2l.torch.Accumulator method*), 954
reshape() (*in module d2l.torch*), 973
Residual (*class in d2l.torch*), 960
resnet18() (*in module d2l.torch*), 973
RNNModel (*class in d2l.torch*), 959
RNNModelScratch (*class in d2l.torch*), 959

S

Seq2SeqEncoder (*class in d2l.torch*), 960
seq_data_iter_random() (*in module d2l.torch*), 973
seq_data_iter_sequential() (*in module d2l.torch*), 973
SeqDataLoader (*class in d2l.torch*), 960
sequence_mask() (*in module d2l.torch*), 973
set_axes() (*in module d2l.torch*), 973
set_figsize() (*in module d2l.torch*), 973
sgd() (*in module d2l.torch*), 973
show_bboxes() (*in module d2l.torch*), 973
show_heatmaps() (*in module d2l.torch*), 973
show_images() (*in module d2l.torch*), 974
show_trace_2d() (*in module d2l.torch*), 974
sin() (*in module d2l.torch*), 974
sinh() (*in module d2l.torch*), 974
size() (*in module d2l.torch*), 974
SNLIDataset (*class in d2l.torch*), 960
split_batch() (*in module d2l.torch*), 974
squared_loss() (*in module d2l.torch*), 974
stack() (*in module d2l.torch*), 974
start() (*d2l.torch.Timer method*), 960
stop() (*d2l.torch.Timer method*), 961
subsampling() (*in module d2l.torch*), 975
sum() (*d2l.torch.Timer method*), 961
synthetic_data() (*in module d2l.torch*), 975

T

tanh() (*in module d2l.torch*), 975
tensor() (*in module d2l.torch*), 975
Timer (*class in d2l.torch*), 960
to() (*in module d2l.torch*), 976
to_tokens() (*d2l.torch.Vocab method*), 961
TokenEmbedding (*class in d2l.torch*), 961
tokenize() (*in module d2l.torch*), 976
tokenize_nmt() (*in module d2l.torch*), 976
train_2d() (*in module d2l.torch*), 976
train_batch_ch13() (*in module d2l.torch*), 976
train_ch3() (*in module d2l.torch*), 976
train_ch6() (*in module d2l.torch*), 976
train_ch8() (*in module d2l.torch*), 976
train_ch11() (*in module d2l.torch*), 976
train_ch13() (*in module d2l.torch*), 976
train_concise_ch11() (*in module d2l.torch*), 977
train_epoch_ch3() (*in module d2l.torch*), 977
train_epoch_ch8() (*in module d2l.torch*), 977
train_seq2seq() (*in module d2l.torch*), 977
training (*d2l.torch.AdditiveAttention attribute*), 955
training (*d2l.torch.AddNorm attribute*), 954
training (*d2l.torch.AttentionDecoder attribute*), 955
training (*d2l.torch.BERTEncoder attribute*), 955
training (*d2l.torch.BERTModel attribute*), 956
training (*d2l.torch.Decoder attribute*), 956
training (*d2l.torch.DotProductAttention attribute*), 956
training (*d2l.torch.Encoder attribute*), 957
training (*d2l.torch.EncoderBlock attribute*), 957
training (*d2l.torch.EncoderDecoder attribute*), 957
training (*d2l.torch.MaskLM attribute*), 958
training (*d2l.torch.MultiHeadAttention attribute*), 958
training (*d2l.torch.NextSentencePred attribute*), 958
training (*d2l.torch.PositionalEncoding attribute*), 959
training (*d2l.torch.PositionWiseFFN attribute*), 959
training (*d2l.torch.Residual attribute*), 960
training (*d2l.torch.RNNModel attribute*), 959
training (*d2l.torch.Seq2SeqEncoder attribute*), 960
training (*d2l.torch.TransformerEncoder attribute*), 961
TransformerEncoder (*class in d2l.torch*), 961
transpose() (*in module d2l.torch*), 977
transpose_output() (*in module d2l.torch*), 977
transpose_qkv() (*in module d2l.torch*), 977
truncate_pad() (*in module d2l.torch*), 977
try_all_gpus() (*in module d2l.torch*), 977
try_gpu() (*in module d2l.torch*), 977

U

update_D() (*in module d2l.torch*), 977
update_G() (*in module d2l.torch*), 977
use_svg_display() (*in module d2l.torch*), 977

V

voc_label_indices() (*in module d2l.torch*), 977
voc_rand_crop() (*in module d2l.torch*), 977
Vocab (*class in d2l.torch*), 961
VOCSegDataset (*class in d2l.torch*), 961

Z

zeros() (*in module d2l.torch*), 977