Dive into Deep Learning

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola

May 05, 2020

Contents

1

まえがき

ほんの2、３年前は、大きな企業やスタートアップにおいて知的な製品やサービスを開発するような、深層学習の科学者のチームは存在しませんでした。われわれ著者のうち、最も若い世代がこの分野に入ったときも、日々の新聞で機械学習が新聞の見出しにでることはありませんでした。われわれの両親は、われわれが医薬や法律といった関係の職業よりも機械学習を好んでいることはもちろん、機械学習が何なのかということについても知りません。機械学習は、狭い領域における実世界の応用に向けた、先進的な学習的領域だったのです。例えば音声認識やコンピュータビジョンといった応用では、機械学習というのは非常に小さな構成要素の1つで、それとは別に、多くのドメイン知識を必要としたのでした。ニューラルネットワークは、われわれがこの本で着目する深層学習モデルの元になるものですが、時代遅れなツールだと考えられていました。

5年ほど前、深層学習はコンピュータビジョン、自然言語処理、自動音声認識、強化学習、統計的モデリングといった多様な分野において急速な進歩をみせ、世界を驚かせました。このような発展を手に、われわれは自動運転を開発したり（車の自動性を高めたり）、日常の応答を予測する賢い応答システムを開発したり、山のようなメールを探すことを助けたり、碁のようなボードゲームで世界のトッププレイヤーに打ち勝つソフトウェアエージェントを開発したり、数十年はかかると思われていたものを開発したりできるようになりました。すでに、これらのツールはそのインパクトを拡大し続けており、映画の作成方法を変え、病気の診断方法を変え、天体物理学から生物学に至るまでの基礎科学における役割を大きくしています。

1.1 この書籍について

この本では、深層学習をより身近なものとするための試みを紹介し、読者に概念、具体的な内容、コードのすべてをお伝えします。

1.1.1 コード、数学、HTMLを結びつける手段

どのようなコンピュータの技術もそのインパクトを十分に発揮するためには、十分に理解され、文書化され、成熟して十分に保守されたツールによって支援される必要があります。キーとなるアイデアを明確な形で切り出して、新たに深層学習に取り組む人が最新の技術を身につけるための時間を最小化すべきです。成熟したライブラリは共通のタスクを自動化し、お手本となるコードは、取り組む人が必要とするものにあわせて、アプリケーションを簡単に修正、応用、拡張する手助けとなるべきです。動的なWebアプリケーションを例にあげましょう。Amazonのような多くの企業が1990年代にデータベースを利用したWebアプリケーションの開発に成功しましたが、創造的な事業家を支援する潜在的な技術は、ここ10年で大きく実現されたものであり、それは強力で十分に文書化されたフレームワークの開発のおかげだったのです。

どんなアプリケーションも様々な学問によって成り立っているので、深層学習の潜在的な能力を試すことは、他にはない挑戦となるでしょう。深層学習を適用するためには以下を理解する必要があります。

(i) ある特定の手段によって問題を投げかけるための動機(ii) 与えられたモデリングアプローチを構成する数学(iii) モデルをデータに適合させるための最適化アルゴリズム(iv) モデルを効率的に学習させるため工学、つまり数値計算の落とし穴にはまらないようにしたり、利用可能なハードウェアを最大限生かすこと

問題を定式化するために必要なクリティカルシンキングのスキル、その問題を解くために必要な数学、その解法を実装するためのソフトウェアについて、1つの場所で教えることは恐ろしいほどの挑戦です。この本における、われわれのゴールは、将来の実践者に対して必要な情報を提供するための統一的なリソースを提示することです。

われわれは、この本のプロジェクトを2017年7月に開始し、そのころ、MXNetの新しいGluonのインターフェースをユーザに説明する必要がありました。当時、(1) 常に最新で、(2) 技術的な深さをもちあわせて、現代の機械学習を幅広くカバーし、(3) 魅力的な教科書に期待される説明文の中に、ハンズオンの資料で求められるような整備された実行可能なコードが含まれるようなリソースは存在しませんでした。世の中には、深層学習のフレームワークの利用方法（例えば、Tensorflowにおける行列の基本的な数値計算）や、特定の技術を実装する方法（たとえば、LeNet、AlexeNet、ResNetなどのコードスニペット）に関する十分な数のコード例が、ブログの記事やGitHubに存在しています。しかし、これらの例は典型的には与えられたアプローチをどのように実装するかに重点が置かれていて、なぜそのアルゴリズムに決定したのかという議論はなされていません。ブログの記事は散発的にこのような内容をカバーしていて、例えばDistillというウェブサイトや、個人のブログなどがありますが、深層学習における限定的な内容をカバーするだけで、しばしば関連するコードがありません。一方で、いくつかの教科書が登場し、最も著名なGoodfellow, Bengio and Courville, 2016は、深層学習を支える概念につ

いて丁寧に調査、説明しているが、これらのリソースは説明文とコードによる実装を合わせておらず、それを実装する方法について、ときどき読者に手がかかりを与えないままになっています。加えて、非常に多くのリソースは、商用の教育コースの提供者によって、課金者のみにアクセスできるようになっており隠れてしまっています。

そこで、以下のことが可能なリソースの作成に着手しました。

(1) あらゆる人にとって無料で利用できる(2) 実際に機械学習の応用ができるサイエンティストになるための、起点となるような十分な技術的な深さを提供する(3) 読者にどうやっても問題を解くかを示すような実行可能なコードを含む(4) 我々や大部分はコミュニティによって、すばやくアップデートされる(5) 技術的な詳細についての対話的な議論や質問に回答するためのforumによって補足される

これらのゴールはしばしば衝突してしまいます。数式、理論、そして引用は、最善な形で管理されて、LaTeXによって組版されます。コードはPythonで最善の形で記述される。そして、ウェブページはもともとHTMLやJavaScriptによって成り立っている。さらに、実行可能なコードとして、物理的な書籍として、ダウンロード可能なPDFとして、インターネット上のWeb Siteとして、これらのいずれの場合でもアクセス可能なコンテンツを欲しいと考えていました。現在もこれらの要求に完全に応えられるツールやワークフローは存在しません。そこでわれわれは、われわれ自信でこれを構築する必要がありました。われわれは、このアプローチの詳細について appendixに記載しています。そのソースの共有や編集の許可のためのGithub、コードと数式と文章を調和させるJupyter ノートブック、複数の出力(Webページ、PDFなど)を生成するためのSphinx、フォーラムのためのDisclosure、これらを利用することを決めました。われわれのシステムはまだ完全ではない一方で、これらの選択は互いに競合してきた関心事を互いに歩み寄らせているでしょう。この場がこのような統合的なワークフローを利用して最初に出版される本になるだろうと、われわれは信じています。

1.2 やってみて学ぶ

多くの教科書は、一連のトピックについて網羅的に詳細を交えて説明します。例えば、Chris Bishopの素晴らしい書籍:cite:`Bishop.2006` は、各トピックを徹底的に説明し、線形回帰の章でもとてつもない量の内容を理解する必要があります。専門家は、その書籍の完全さゆえに非常に気に入っていますが、初心者にとっての導入の書籍としてはその有用性を活かしきれません。

この本では、たいていの考え方を *just in time*、つまり必要なときに教える方針です。言い換えれば、いくつかの実践的な目標を達成するために、ある考え方が必要になったときに、それを教えていきます。基礎となる予備知識、例えば、線形代数や確率といったものを伝える最初の段階では少し時間がかかると思います。風変わりな確率分布について心配する前に、最初のモデルを学習することに満足さを感じてほしいと思っています。

基礎的な数学の知識に関する集中講義を提供するための、一部の初歩的なノートブックは別として、それ以降のノートブックはほどよい量の新しいコンセプトを紹介し、全てがそろった単一で実行可能な例を実際のデータセットとともに提供します。これは、１つの構成上のチャレンジと呼んでいいと思います。いくつかのモデルは、単一のノートブックの中に論理的にひと

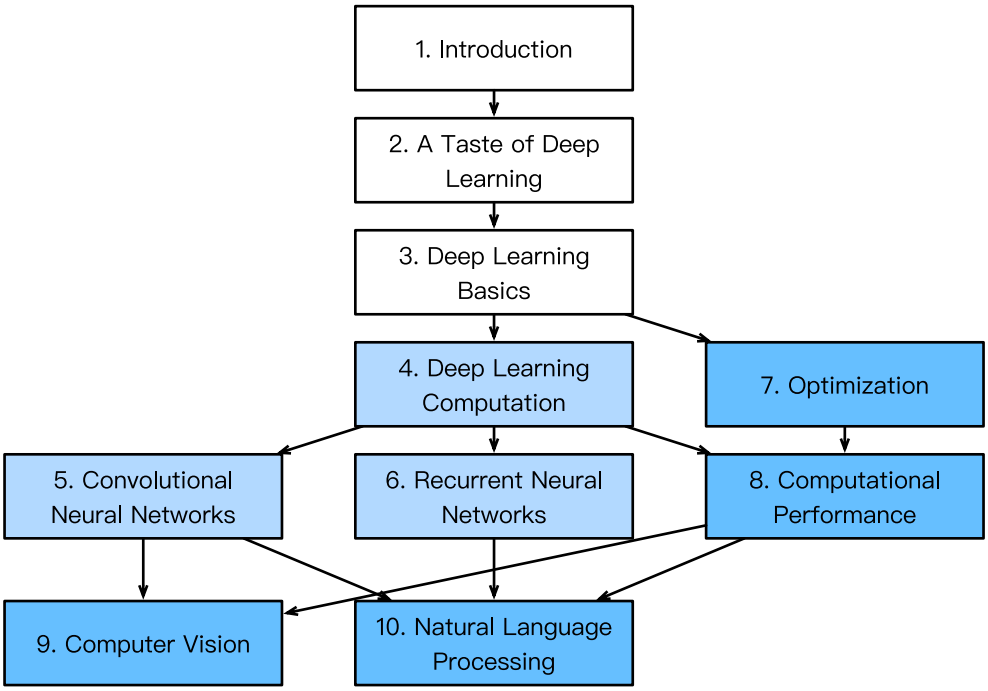
まとめにすることもできるでしょう。そして、そのいくつかのモデルを連続的に実行することによって、考え方を教えたほうが良いかもしれません。一方、* 1つの実行可能な例と1つのノートブック* のポリシーに固執するのには大きな利点があります。われわれのコードを活用して、読者のみなさんの研究プロジェクトを、なるべく簡単に立ち上げるのを助けることができます。1つのノートブックをコピーしてから、それを修正していけばよいのです。

われわれは、必要に応じて実行可能なコードなかに、それを理解するための情報を含めます。一般的に、ツールを十分に説明する前に、そのツールを使えるようにしようとして失敗しすることもあるでしょう（そして、あとになってフォローアップの説明をします）。例えば、*stochastic gradient descent* について、それがなぜ有用で、なぜうまくいくのかを十分に説明する前に使うことがあると思います。この方法は、短時間のうちのいくつかの判断だけで、読者にわれわれを信用してもらうことになりますが、実際に機械学習を行う人に対して、問題をすばやく解くための武器を与えることができます。

ここでは最初から最後まで、MXNetのライブラリを利用して進める予定です。MXNetのライブラリは研究用途にも十分に柔軟で、本番環境の用途にも十分高速であるという得がたい特徴をもっています。深層学習の考え方についてゼロから伝えていく予定で、ときどき、Gluonの先進的な特徴によってユーザに隠蔽されたモデルについて、詳細な部分を掘り下げたいと思います。与えられた深層学習のレイヤーの中で起こっているすべてを理解してほしいので、特に基礎的なチュートリアルにおいて掘り下げを行います。この場合に、われわれは次の2種類の例を一般的に提示します。1つはNDArray(多次元配列）や自動微分を利用してゼロか全てを実装するもので、もう1つはGluonによって同じことを簡潔に実装するものです。レイヤーがどのように動くかを伝えたら、以降のチュートリアルでは、Gluonを利用したものを使います。

1.3 内容と構成

この本は、おおまかに次の3つのパートにわかれています。



:label:`fig_book_org`

- 最初のパートは前提条件や基礎の部分を扱います。:numref:`chap_introduction`では深層学習の紹介をします。そして、:numref:`chap_preliminaries`では、ハンズオン形式の深層学習に必要な準備について、例えば、データの保存・加工の方法や、線形代数、微分・積分、確率などの基本的な考え方にもとづいた、演算の適用方法などを紹介します。:numref:`chap_linear`と:numref:`chap_perceptrons`は、深層学習における最も基本的な考え方と技術、例えば、多層パーセプトロンや正則化について説明します。
- 次の5つの章は現代の深層学習の技術に焦点を当てています。:numref:`chap_computation`は、深層学習における計算に関する重要な要素をいくつか説明し、後に複雑なモデルを実装するための土台を築きます。:numref:`chap_cnn`と:numref:`chap_modern_cnn`では、畳み込みニューラルネットワーク (Convolutional Neural Networks; CNNs)という、現在のコンピュータビジョの土台を築く強力なツールを紹介します。続いて、:numref:`chap_rnn`と:numref:`chap_modern_rnn`では、再帰ニューラルネットワーク (Recurrent Neural Networks; RNNs) とよばれる、時系列・系列データを扱うモデルで、自然言語処理や時系列データ予測に利用されるモデルを紹介します。:numref:`chap_attention`では、Attention Mechanisms と呼ばれる技術を用いた新しいモデルについて紹介します。これは、自然言語処理における RNN に取って代わりつつあります。これらの章では、深層学習の最近の多くのアプリケーションを支える基本的なツールについて理解してもらうことを目的としています。
- パート3は、スケーラビリティ、効率性、アプリケーションについて考察します。まず、:numref:`chap_optimization`では、深層学習モデルの学習に利用される、いくつかの最適化アルゴリズムについて考察します。次の章である:numref:`chap_performance`

では、深層学習の性能に影響する重要な要素について調査します。:numref:chap_cvと:numref:chap_nlp では、コンピュータビジョンと自然言語処理のそれぞれにおける、深層学習の主要なアプリケーションを説明します。

1.4 コード

:label:sec_code

この本のほとんどの章では、深層学習におけるインタラクティブな学習体験が重要であることを信じて、実行可能コードを取り上げています。そして、コードを細かく調整し、結果を観察するという試行錯誤を通して確かな感覚を開発することができます。理想的には、洗練された数学的理論が、コードを微調整して目的の結果を達成するための方法を、正確に教えてくれる可能性がなくはありません。残念ながら、現在、そのような洗練された理論は私たちにはありません。私たちがここで最善を尽くしたとしても、これらのモデルを特徴付ける数学は非常に難しく、これらのトピックに関する重要な調査が最近始まったばかりであるため、さまざまな手法の正式な説明はまだ不十分なままです。深層学習の理論が進むにつれて、現在の版では提供できない洞察を、将来の版で提供できるようになることを期待しています。

この書籍のたいていのコードはApache MXNetを利用しています。MXNetは、AWS (Amazon Web Services)が選んだ、深層学習のためのオープンソースフレームワークで、多くの大学と企業で利用されています。この書籍の全てのコードはMXNet1.2.0にもとづくテストをパスしています。しかし、深層学習の急速な発展にともなって、印刷版のコードは、MXNetの将来のバージョンでは動かない可能性があります。一方、オンライン版は常に最新版に維持されるようにします。もしなにか問題が生じたら、:ref:chap_installation を見て、コードや実行環境をアップデートしましょう。

現在、不必要な繰り返し作業を避けるため、この書籍で頻繁にimportされたり、参照されたりする関数、クラス、その他については、d2lのパッケージにカプセル化します。パッケージに保存されている関数、クラス、複数のインポートなどのブロックは、`# Saved in the d2l package for later use`といった注記をします。d2lのパッケージは軽量で、次のパッケージやモジュールのみを依存関係として必要とします。

```
# Saved in the d2l package for later use
import collections
from collections import defaultdict
from IPython import display
import math
from matplotlib import pyplot as plt
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
import os
import pandas as pd
import random
import re
import sys
import tarfile
```

(continues on next page)

(continued from previous page)

```
import time
import zipfile
```

これらの関数やクラスについての詳細は:numref:sec_d2l で説明します。

1.5 対象とする読者

この本は、深層学習の実践的な技術を学びたい大学生 (学部生、大学院生)、エンジニア、研究者を対象にしています。著者らはゼロからすべての考え方を説明するつもりですので、深層学習や機械学習に関する事前知識をもっている必要はありません。深層学習のモデルを完全に説明するためには、いくらかの数学やプログラミングが必要ですが、(非常に基本的な)線形代数、微積分、確率、pythonのプログラミングを含む基礎を理解できるようになれば良いと考えています。そして、Appendix では、この書籍でカバーされる数学を再思い出すための内容を提供します。この書籍では、多くの場合、数学的な厳密さにもとづく直感的な理解や考え方を重視します。興味がある読者がさらに深く学ぶためのすごい書籍はたくさんあります。例えば、Bela Bollobas の Linear Analysis :cite:Bollobas.1999 非常に深い線形代数や関数解析に関してカバーしています。All of Statistics :cite:Wasserman.2013 は統計に関する素晴らしいガイドです。もし Python を以前に利用したことがなければ、Python tutorial を追ってみるのも良いかもしれません。

1.5.1 フォーラム

この書籍に関連して、discuss.mxnet.ioに議論のためのフォーラムを立ち上げています。この書籍のどの部分でも、もし質問があれば、各節の最後にあるQRコードをスキャンして関連する議論のページにたどり着き、議論に参加することができます。この書籍の著者と、幅広いMXNet開発者コミュニティは、このフォーラムでの議論によく参加しています。

1.6 謝辞

われわれは、英語と日本語のドラフト版の作成に貢献した数百の人に恩があります。彼らは、内容を改善するための手助けをしてくれ、価値あるフィードバックを提供してくれました。特に、あらゆる人のために英語のドラフト版を改善した人々に感謝したいです。かれらのGithubのIDや名前を順不同で記載します。

alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, mohamed-ali, mstewart141, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, vishwesh5, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, IgorDzreyev, trungha-ngx, pmuens, alukovenko,

senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, vfdev-5, bowen0701, arush15june, prasanth5reddy.

さらに、われわれはAmazon Wen Services、特に、Swami Sivasubramanian、Raju Gulabani、Charlie Bell、Andrew Jassyには、この書籍を執筆するための惜しみないサポートしてくれたことに感謝します。

費やした時間、リソース、同僚との議論、継続的な取り組みなくして、この書籍は生まれなかったでしょう。

1.7 まとめ

- 深層学習は、パターン認識を革新し、コンピュータビジョン、自然言語処理、自動音声認識などの幅広い分野に力を与える技術を導入しました。
- うまく深層学習を適用するためには、問題を明らかにする方法、数学的なモデリング、モデルをデータに当てはめるためのアルゴリズム、それらすべてを実装する工学的な技術が必要です。
- この書籍は、文章、図表、数学、コード、これらすべてを1箇所にまとめた、包括的なリソースを提供します。
- この書籍に関する質問に回答するために、https://discuss.mxnet.io/ のフォーラムに参加してください。
- Apache MXNetは深層学習モデルのコードを実装し、複数のGPUコアで並列に実行するための強力なライブラリです。
- Gluonという高レベルなライブラリを利用することで、Apache MXNetを利用した深層学習モデルの実装を簡単に行うことができます。
- Condaは、すべてのソフトウェアの依存関係が満たされることを保証するPythonのパッケージ管理ツールです。
- すべてのノートブックはGithub上でダウンロードすることが可能で、この書籍のコードを実行するために必要なcondaの設定ファイルは`environment.yml`のファイルの中に記述されています。
- もしここでのコードをGPU上で実行しようと考えているのであれば、必要なドライバをインストールして、構成を更新することを忘れないようにしましょう。

1.8 練習

1. この書籍のフォーラムdiscuss.mxnet.ioでアカウントを登録しましょう。
2. コンピュータにPythonをインストールしましょう。

3. 各セクションの最後にあるフォーラムへのリンクをたどりましょう。フォーラムでは、著者や広いコミュニティに関わることで、助けを求めたり、書籍の内容を議論したり、疑問に対する答えを探すことができるでしょう。
4. フォーラムのアカウントを作成して自己紹介をしましょう。

1.9 議論のためのQRコードをスキャン

2

インストール

:label:chap_installation

ハンズオンを始めてもらうために、Pythonの環境、Jupyterの対話的なノートブック、関連するライブラリ、この書籍を実行するためのコードをセットアップしてもらう必要があります。

2.1 Miniconda のインストール

最もシンプルに始めるために Minicondaをインストールしましょう。Python 3系が推奨です。もし conda がすでにインストールされていれば以下のステップをスキップすることができます。ウェブサイトから、対応する Miniconda の sh ファイルをダウンロードして、コマンドラインから `sh <FILENAME> -b` を実行してインストールします。macOS のユーザは以下のように行います。

```
# The file name is subject to changes
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

そして Linux ユーザは以下のように行います。

```
# The file name is subject to changes
sh Miniconda3-latest-Linux-x86_64.sh -b
```

次に、condaから直接シェルを初期化します。

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

いまのシェルを閉じて再度開いてください。以下のように新しい環境を作ることができるはずです。

```
conda create --name d2l -y
```

2.2 D2L ノートブックのダウンロード

次にこの書籍のコードをダウンロードしましょう。コードをlinkからダウンロードして解凍します。もし unzip がインストール済みであれば (なければ sudo apt install unzip でインストールできます)、代わりに以下でも可能です。

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-0.7.1.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

ここで d2l の環境を activate して、pip をインストールします。このコマンドの最後に y を入れておきましょう。

```
conda activate d2l
conda install python=3.7 pip -y
```

2.3 MXNet と d2l パッケージのインストール

MXNet をインストールする前に、まず、利用する計算機に適切なGPU (標準的なノートPCでグラフィックスのために利用されるGPUは対象外です) が利用可能かどうかを確認してください。GPU サーバにインストールしようとしているなら、GPUサポートのMXNet をインストールするための手順:ref:subsec_gpu に従ってください。

もし GPU がなければ、CPUバージョンをインストールしましょう。最初の数章を行う際には十分な性能でしょうが、より規模の大きいモデルを動かす際には GPU を必要とするかもしれません。

```
# For Windows users
pip install mxnet==1.6.0b20190926

# For Linux and macOS users
pip install mxnet==1.6.0b20191122
```

この書籍でよく使う関数やクラスをまとめた d2l パッケージもインストールしましょう。

```
pip install d2l==0.11.1
```

インストールできたら、実行のために Jupyter ノートブックを開きます。

```
jupyter notebook
```

この段階で、http://localhost:8888 (通常、自動で開きます) をブラウザで開くことができます。そして、この書籍の各章のコードを実行することができます。この書籍のコードを実行したり、MXNet や `d2l` のパッケージを更新する前には、`conda activate d2l` を必ず実行して実行環境を activate しましょう。環境から出る場合は、`conda deactivate` を実行します。

2.4 最新バージョンへ更新

この書籍と MXNet は絶えず改善を続けています。次々とリリースされる最新バージョンをチェックしましょう。

1. https://d2l.ai/d2l-en.zip の URL は常に最新版を保持しています。
2. `d2l` パッケージを `pip install d2l --upgrade` を実行して更新しましょう。
3. CPU バージョンの場合は, `pip install -U --pre mxnet` MXNet を更新できます。

2.5 GPUのサポート

:label:subsec_gpu

デフォルトでは、MXNetはあらゆるコンピュータ (ノートパソコンも含む)で実行できるように、GPUを利用しないようにインストールされます。この書籍の一部は、GPUの利用を必要としたり、推薦したりします。もし読者のコンピュータが、NVIDIAのグラフィックカードを備えていて、CUDAがインストールされているのであれば、GPUを利用可能なMXNet をインストールしましょう。CPU のみをサポートするMXNet をインストールしていた場合は、以下を実行して、まずそれを削除する必要があります。

```
pip uninstall mxnet
```

次に、インストール済みの CUDA のバージョンを知る必要があります。`nvcc --version` や `cat /usr/local/cuda/version.txt` を実行して知ることができるかもしれません。CUDA 10.1 がインストールされているとすれば、以下のコマンドで MXNet をインストールすることができます。

```
# For Windows users
pip install mxnet-cu101==1.6.0b20190926

# For Linux and macOS users
pip install mxnet-cu101==1.6.0b20191122
```

CPU バージョンと同様に、GPUを利用可能な MXNet は `pip install -U --pre mxnet-cu101` で更新できます。CUDAのバージョンに合わせて、最後の数字を変えることができます。例えば、CUDA 10.0であれば `cu100`、CUDA 9.0であれば `cu90` です。利用可能なMXNetのバージョンをすべて調べるためには、`pip search mxnet` を実行します。

2.6 練習

1. この本のコードをダウンロードして、実行環境をインストールしましょう。

2.7 議論のためのQRコード

3

表記法

:label:chap_notation
書籍内の表記法を以下にまとめます。

3.1 変数

- x: スカラー
- $\mathbf{x}$: ベクトル
- $\mathbf{X}$: 行列
- X: テンソル
- $\mathbf{I}$: 単位行列
- x_i, $[\mathbf{x}]_i$: ベクトル $\mathbf{x}$ の i 番目の要素
- x_{ij}, $[\mathbf{X}]_{ij}$: 行列 $\mathbf{X}$ の行 i、列j の要素

3.2 集合

- $\mathcal{X}$: 集合

- $\mathbb{Z}$: 整数の集合
- $\mathbb{R}$: 実数の集合
- $\mathbb{R}^n$: 実数の n 次元ベクトル
- $\mathbb{R}^{a\times b}$: a行b列の実数の行列
- $\mathcal{A}\cup\mathcal{B}$: $\mathcal{A}$ と $\mathcal{B}$の和集合
- $\mathcal{A}\cap\mathcal{B}$: $\mathcal{A}$ と $\mathcal{B}$の積集合
- $\mathcal{A}\setminus\mathcal{B}$: $\mathcal{A}$ から $\mathcal{B}$を引いた差集合

3.3 関数と演算

- $f(\cdot)$: 関数
- $\log(\cdot)$: 自然対数
- $\exp(\cdot)$: 指数関数
- $\mathbf{1}_\mathcal{X}$: 指示関数
- $\mathbf{(\cdot)}^\top$: ベクトルや行列の転置
- $\mathbf{X}^{-1}$: 行列 $\mathbf{X}$ の逆行列
- $\odot$: アダマール (要素ごとの) 積
- $\lvert \mathcal{X} \rvert$: 集合 $\mathcal{X}$ の基数 (カーディナリティ)
- $|\cdot|_p$: ℓ_p ノルム
- $|\cdot|$: ℓ_2 ノルム
- $\langle \mathbf{x}, \mathbf{y} \rangle$: ベクトル $\mathbf{x}$ と $\mathbf{y}$ のドット積
- $\sum$: 総和
- $\prod$: 総乗

3.4 微積分

- $\frac{dy}{dx}$: y の x についての微分
- $\frac{\partial y}{\partial x}$: y の x についての偏微分
- $\nabla_{\mathbf{x}} y$: x についての y の勾配
- $\int_a^b f(x) ;dx$: xに関して a から b までの定積分

- $\int f(x) ;dx$: f の x についての不定積分

3.5 確率と情報理論

- $P(\cdot)$: 確率分布
- $z \sim P$: 確率変数 z が確率分布 P に従う
- $P(X \mid Y)$: $X \mid Y$ の条件付き確率
- $p(x)$: 確率密度関数
- ${E}_{x} [f(x)]$: xに関するfの期待値
- $X \perp Y$: 確率変数 X と Y は独立である
- $X \perp Y \mid Z$: 確率変数 X と Y は、確率変数Z の条件のもとでの条件付き独立
- $\mathrm{Var}(X)$: 確率変数 X の分散
- σ_X: 確率変数 X の標準偏差
- $\mathrm{Cov}(X, Y)$: 確率変数 X と Y の共分散
- $\rho(X, Y)$: X と Y の相関係数
- $H(X)$: 確率変数 X のエントロピー
- $D_{\mathrm{KL}}(P|Q)$: 分布 P と Q の KL-divergence

3.6 複雑さ

- $\mathcal{O}$: Big O notation (ビッグ・オー記法)

3.7 議論

4

はじめに

日々利用しているほとんどすべてのコンピュータのプログラムは、最近になるまで、ソフトウェア開発者によって第一原理にもとづいて開発されています。e-commerceのプラットフォームを管理するアプリケーションを作成したい場合を考えましょう。その問題を考えるために数時間、ホワイトボードに集まって、以下のような上手くいきそうなソリューションを書いたとします。

(i) ユーザはウェブブラウザやモバイルのアプリケーションからインターフェースを介して、そのアプリケーションとやり取りする

(ii) そのアプリケーションは、ユーザの状態を追跡したり、すべての処理履歴を管理するために商用のデータベースを利用する

(iii) アプリケーションの中心に、複数のサーバ上で並列実行される、ビジネスロジック (ブレーンと呼ぶかもしれません)が、考えられる状況に応じた適切なアクションを緻密に計画する

そうしたブレーンをもつアプリケーションを構築するためには、起こりうるすべての特別なケースを考慮して、適切なルールを作成しなければなりません。顧客が商品を買い物かごに入れるためにクリックするたびに、買い物かごデータベースにエントリを追加し、購入する商品のIDと顧客IDを関連付けます。これを一度で完全に理解できる開発者はほとんどいないでしょう（問題を解決するために、何度かテストを実行する必要があるでしょう）。そして、多くの場合、このようなプログラムを第一原理 (First Principles)に従って書くことができ、実際のお客様を見る前に構築することができるでしょう。人間には全く新しい状況においても、製品やシステムを動かす第一原理から、自動的に動くシステムを設計する能力があり、これは素晴

らしい認知能力です。そして、100%動くようなソリューションを開発できるのであれば、機械学習を使わないほうがよいでしょう。

ますます成長している機械学習サイエンティストのコミュニティにとっては幸運なことですが、自動化における多くの問題は、そうした人間の創意工夫だけではまだ簡単に解決されていません。ホワイトボードにあなたが思いつく賢いプログラムを書き出してみましょう。例えば、以下のようなプログラムを書くことを考えます。

- 地理情報、衛星画像、一定間隔の過去の天気データから、明日の天気を予測するプログラムを書く
- フリーフォーマットのテキストで書かれた質問を理解して、適切に回答するプログラムを書く
- 画像に含まれる全ての人間を認識して、それらを枠で囲むプログラムを書く
- ユーザが楽しむような製品で、通常の閲覧の過程ではまだ見ていない製品を提示するプログラムを書く

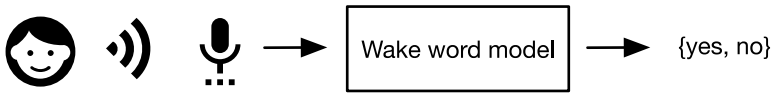
これらの場合では、優秀なプログラマーであってもゼロからプログラムを書くことはできないでしょう。その理由としては様々なものが考えられます。求められるプログラムが時間によって変化するようなパターンをもつこともあり、それに適応するプログラムも必要になります。また、関係性 (例えば、ピクセル値と抽象的なカテゴリとの関係性) が複雑すぎて、私達の意識的な理解を超える数千、数百万の計算を必要とすることもあります (私達の目はこのようなタスクを難なくこなしますが)。機械学習 (Machine Learning, ML) は経験から挙動を学習できる協力な技術の学問です。MLのアルゴリズムは、観測データや環境とのインタラクションから経験を蓄積することで、性能が改善します。さきほどの決定的な電子商取引のプラットフォームは、どれだけ経験が蓄積されても、開発者自身が学習して、ソフトウェアを改修することを決めないかぎり、同じビジネスロジックに従って実行するため、これとは対照的です。この書籍では、機械学習の基礎について説明し、特に深層学習に焦点を当てます。深層学習は、コンピュータビジョン、自然言語処理、ヘルスケア、ゲノミクスのような幅広い領域において、イノベーションを実現している強力な技術です。

4.1 身近な例

この書籍の著者は、これを書き始めるようとするときに、多くの仕事を始める場合と同様に、カフェインを摂取していました。そして車に飛び乗って運転を始めたのです。iPhoneをもって、Alexは“Hey Siri”と言って、スマートフォンの音声認識システムを起動します。Muは「ブルーボトルのコーヒーショップへの行き方」と言いました。するとスマートフォンは、すぐに彼の指示内容を文章で表示しました。そして、行き方に関する要求を認識して、意図に合うようにマップのアプリを起動したのです。起動したマップはたくさんのルートを提示しました。この本で伝えたいことのために、この物語をでっちあげてみましたが、たった数秒の間における、スマートフォンとの日々のやりとりに、様々な機械学習モデルが利用されているのです。

“Alexa”、“Okay Google”、“Siri”といった起動のための言葉に反応するコードを書くことを考えてみます。コンピュータとコードを書くエディタだけをつかって部屋でコードを書いてみまし

ょう。どうやって第一原理に従ってコードを書きますか？考えてみましょう。問題は難しいです。だいたい44,000/秒でマイクからサンプルを集めます。音声の生データに対して、起動の言葉が含まれているかを調べて、YesかNoを正確に対応付けるルールとは何でしょうか？行き詰まってしまうと思いますが心配ありません。このようなプログラムを、ゼロから書く方法はだれもわかりません。これこそが機械学習を使う理由なのです。



:label:fig_wake_word

ひとつの考え方を紹介したいと思います。私達が、入力と出力を対応付ける方法をコンピュータに陽に伝えることができなくても、私達自身はそのような認識を行う素晴らしい能力を持っています。言い換えれば、たとえ“Alexa”といった言葉を認識するようにコンピュータのプログラムを書けなくても、あなた自身は“Alexa”の言葉を認識できます。従って、私達人間は、音声のデータと起動の言葉を含んでいるか否かのラベルをサンプルとして含む巨大なデータセットをつくることができます。機械学習のアプローチでは、起動の言葉を認識するようなシステムを陽に実装しません。代わりに、膨大なパラメータによって挙動を決められるような、柔軟なプログラムを実装します。そして、興味あるタスクの性能を測る指標に関して、プログラムの性能を改善するような、もっともらしいパラメータを決定するため、データセットを利用します。

パラメータは、プログラムの挙動を決めるために、私達が回すことのできるノブのようなものと考えられるでしょう。パラメータを確定すると、そのプログラムはモデルと呼ばれます。異なるプログラム (入力と出力のマッピング) をパラメータを変更するだけで生成可能な場合、それらをモデルのファミリ *(familiy)* と呼ばれます。パラメータを選ぶために、データセットを利用するメタなプログラムを学習アルゴリズムと呼びます。

機械学習アルゴリズムに進んで取り組んでいく前に、問題を正確に定義する必要があり、つまり、入力と出力の性質を正確にはっきりさせ、モデルのファミリを適切に選ぶことが必要です。この場合、モデルは入力として音声の一部を受け取って、出力として{はい、いいえ}からの選択を生成します。そして、このあとの書籍の内容通りに進むのであれば、音声の一部が起動語を含んでいそうか（そうでないか）を近似的に決定すること、といえます。

もし私達が、正しいモデルを選んだのであれば、そのモデルは'Alexa'という言葉を聞いたときにyesを起動するような、1つの設定を切り替えるノブが存在するでしょう。起動する言葉は任意なので、'Apricot'ということばで起動するような別のノブもありえます。入出力が変われば根本的に別のモデルを必要とする場合もあります。例えば、画像とラベルを対応付けるタスクと、英語と中国語を対応付けるタスクには、異なるモデルを利用する必要があるでしょう。

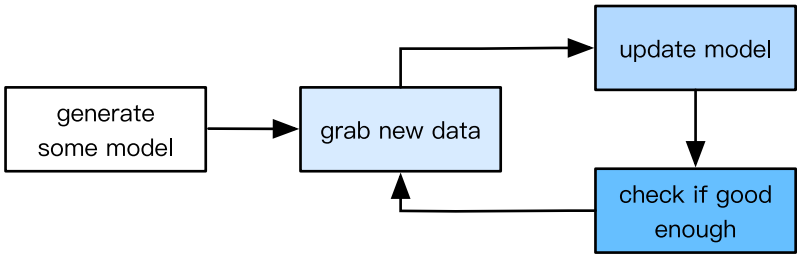
想像できるとは思いますが、“Alexa”と“Yes”のようなペアがランダムなものであれば、モデルは“Alexa”も“Apricot”も他の英語も何も認識できないでしょう。Deep Learningという言葉の中のLearningというのは、モデルから望ましい挙動を強制できるノブの正しい設定を獲得するプロセスと言えます。

:numref:`fig_ml_loop` に示すように, 学習のプロセスは以下のとおりです。

1. まず最初にランダムに初期化されたモデルから始めます。このモデルは最初は使い物に

なりません

2. ラベルデータを取り込みます（例えば、部分的な音声データと対応するYes/Noのラベルです）
3. そのペアを利用してモデルを改善します
4. モデルが良いものになるまで繰り返します



:label:`fig_ml_loop`

まとめると、起動の言葉を認識できるようなコードを直接書くよりも、もしたくさんのラベル付きのデータで認識機能を表現できるなら、その認識機能を学習するようなコードを書くべきです。これは、プログラムの挙動をデータセットで決める、つまりデータでプログラムを書くようなものだと考えることができます。

私達は、以下に示すようなネコとイヌの画像サンプルを大量に集めて、機械学習を利用することで、ネコ認識器をプログラムすることができます。

ネコ	ネコ	イヌ	イヌ

このケースでは、認識器はネコであれば非常に大きな正の値、イヌであれば非常に大きな負の値、どちらかわからない場合はゼロを出力するように学習するでしょう。そしてこれは、MLが行っていることを、ゼロからプログラミングしているわけではありません。

深層学習は、機械学習の問題を解く最も有名なフレームワークのたった一つです。ここまで、機械学習について広く話をしてきましたが、深層学習については話をしていません。深層学習がなぜ重要かを知るために、重要ないくつかの点に焦点をあててみましょう。

まず、これまで議論した問題として、音声情報の生データ、画像のピクセル値、任意の長さの文章や言語間の関係からの学習を議論しましたが、これらの問題では深層学習が優れており、古くからの機械学習ツールは影響力がなくなりつつあります。多くの計算レイヤーを学習するという点において、深層学習モデルは深いです。これらの多くのレイヤーからなるモデル (もしくは階層的なモデル) は、これまでのツールではできない方法で、低レベルな認識に関わるデータを取り扱うことが可能です。

過去には、これらの問題にMLを適用する際の重要な部分として、データを浅いモデルでも扱える形式に変換する方法を人手で構築することがありました。深層学習の1つの重要な利点として、古くからの機械学習パイプラインの最後に利用されていた浅いモデルを置き換えただけでなく、負荷の大きい特徴作成（feature engineering) も置き換えた点があります。また、ドメインに特化した前処理の多くを置き換え、コンピュータビジョン、音声認識、自然言語処理、医療情報学、それ以外にも様々な分野を分割していた境界を多く取り除き、多様な問題を扱える統一的なツールとなりつつあります。

4.2 機械学習の核となる要素: データ・モデル・アルゴリズム

起動語に関する例では、音声の一部と2値のラベルからなるデータセットによって、その音声からラベルの分類へのマッピングを近似するモデルが学習されることを述べました。この種の問題は、ある入力 (特徴や共変量とも呼ばれる) から、事前に定義されたラベルで未知なものを推測しようとするもので、教師あり学習と呼ばれ、多くの機械学習の問題の一つです。次の節では、異なる機械学習の問題について深く考えます。まず、これから取り組むMLの問題の種別にかかわらず、知っておくべき重要な次の要素に光を当てたいと思います。

1. 学習に利用する**データ**
2. データを変換して推定するための**モデル**
3. そのモデルの悪さを定量化する**ロス関数**
4. ロス関数を最小化するような、モデルのパラメータを探す**アルゴリズム**

4.2.1 データ

データなしでデータサイエンスはできない、というのは常識的なことかもしれません。データの細かい性質について考えるための数百のページを手放すことができましたが、ここでは、実践的な側面から過ちを犯してしまったときのために、注意すべき重要な特徴について注目しましょう。一般的に、データ例 *(examples,* データ点、サンプル、インスタンスとも*)*の集合に関心があります。うまくデータを利用するには、適した数値表現を持ち出す必要があります。それぞれのデータ例は、特徴や共変量と呼ばれる数値属性の集合です。

画像データを利用する場合、各写真はデータ例を構成していて、それぞれは各ピクセルの輝度に対応した数値データの順序付きのリストで表現されます。200×200のカラー写真は、$200 \times 200 \times 3 = 120000$個の数値をもっていて、それらは赤、緑、青のチャンネルの輝度と、空間的

な位置に対応しています。より古くからあるタスクには、年齢、バイタルサイン、診断などの特徴から、患者が生存するかしないかを予測するというのもあります。

すべてのデータ例が、同じ数の数値データで表現される場合、そのデータは固定長のベクトルで構成されていると言え、そのベクトルの (一定の) 長さはデータの次元と表現することができます。想像できるかもしれませんが、固定長というのは便利な性質です。顕微鏡検査の画像のなかから、がんを認識するモデルを学習する場合、固定長の入力は心配事項を1つ減らすことができます。

しかし、すべてのデータが固定長のベクトルで簡単に表現されるわけではありません。顕微鏡の画像については、標準的な機器から撮影されたものを期待できるかもしれませんが、インターネットから収集された画像は、すべて同じサイズで現れるとは限りません。標準的なサイズに画像を切り落とすことを考えることもできるでしょう。しかし、テキストデータは固定長の表現がさらに難しいものです。AmazonやTrip Advisorのような電子商取引のサイトにおける、商品レビューを考えてみましょう。何人かは「ひどい !」のように短く、ほかには数ページに渡るレビューもあるでしょう。古くからの方式に比べて、深層学習の主要な利点として、可変長のデータを扱える最新のモデルと親和性が高いことです。

一般的に、多くのデータを持っていれば持っているほど、問題を簡単に解くことができます。多くのデータを持っているなら、より性能の良いモデルを構築することができるからです。比較的小規模なデータからビッグデータと呼ばれる時代への移り変わりによって、現代の深層学習は成り立っているといえます。話をもとに戻しますが、深層学習における最も素晴らしいモデルの多くは大規模なデータセットなしには機能しません。いくつかは小規模なデータの時代においても有効でしたが、従来からのアプローチと大差はありません。

最後に、データをたくさん集めて、それを適切に処理することは十分とは限りません。必要なのは正しいデータなのです。データが間違いだらけだったり、選んだ特徴が興味ある指標を予測しえないものであれば、学習は失敗するでしょう。その状況はよくある決まり文句で、*garbage in, garbage out (*不正確なデータを入れると不正確なデータが出力される*)*といわれています。さらに、不十分な予測能力は、それだけが重大な結果というだけではありません。機械学習のセンシティブなアプリケーション、例えば、予測警備、レジュメの書類審査、貸付のリスクモデルでは、不正確なデータの結果に対して、特に警戒しなければなりません。いくつかのグループの人々が、学習データのなかに表現されていないようなデータセットにおいて、ある失敗が発生しました。これまで黒人の皮膚を見たことがない、皮膚がんの認識システムをあらゆる人に適用することを考えてみてください。データが一部のグループを過小評価してなくとも、社会的な偏見を含む場合には、失敗が起こりうるでしょう。例えば、過去の採用判定が、履歴書を審査するための予測モデルの学習に利用されている場合、機械学習モデルは注意なくこれまでの偏見を取り入れて、自動的に審査するでしょう。このことは、データサイエンティストが意図的でなくとも、気づかないうちに起こってしまう可能性があります。

4.2.2 モデル

ほとんどの機械学習はある意味でデータを変換します。例えば、写真を取り込んで笑顔の度合いを予測するシステムを構築したいと思うかもしれません。あるいは、センサーの測定値が正常なのか異常なのかを予測したいと思うかもしれません。モデルは、あるタイプのデータを取り込んで、おそらく異なるタイプの予測を出力する計算機構を表します。ここでは特に、デー

タから推定できる統計モデルに興味があります。単純なモデルは適切な単純な問題に完全に対処することができますが、この本で焦点を当てる問題は古典的な方法の限界を広げます。深層学習は、主に焦点を当てている一連の強力なモデルによって、古典的なアプローチとは区別されます。これらのモデルは、最初(上)から最後(下)までつながった、連続した多くのデータ変換で成り立っています。したがって、深層学習という名前なのです。深層学習を議論する過程において、古くからの方法についてもより議論するでしょう。

4.2.3 目的関数

Earlier, we introduced machine learning as "learning from experience". By *learning* here, we mean *improving* at some task over time. But who is to say what constitutes an improvement? You might imagine that we could propose to update our model, and some people might disagree on whether the proposed update constituted an improvement or a decline.

In order to develop a formal mathematical system of learning machines, we need to have formal measures of how good (or bad) our models are. In machine learning, and optimization more generally, we call these objective functions. By convention, we usually define objective functions so that *lower* is *better*. This is merely a convention. You can take any function f for which higher is better, and turn it into a new function f' that is qualitatively identical but for which lower is better by setting $f' = -f$. Because lower is better, these functions are sometimes called *loss functions* or *cost functions*.

When trying to predict numerical values, the most common objective function is squared error $(y - \hat{y})^2$. For classification, the most common objective is to minimize error rate, i.e., the fraction of instances on which our predictions disagree with the ground truth. Some objectives (like squared error) are easy to optimize. Others (like error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common to optimize a *surrogate objective*.

Typically, the loss function is defined with respect to the model's parameters and depends upon the dataset. The best values of our model's parameters are learned by minimizing the loss incurred on a *training set* consisting of some number of *examples* collected for training. However, doing well on the training data does not guarantee that we will do well on (unseen) test data. So we will typically want to split the available data into two partitions: the training data (for fitting model parameters) and the test data (which is held out for evaluation), reporting the following two quantities:

- **学習誤差**: 学習済みモデルにおける学習データの誤差です。これは、実際の試験に備えるための練習問題に対する学生の得点のようなものです。その結果は、実際の試験の結果が良いことを期待させますが、最終試験での成功を保証するものではありません。
- **テスト誤差**: たことのないテストデータに対する誤差で、学習誤差とは大きく異なる場合があります。見たことのないデータに対して、モデルが対応（汎化）できないとき、その状態を *overfitting* (過適合)と呼びます。実生活でも、練習問題に特化して準備したにもかかわらず、本番の試験で失敗するというのと似ています。

4.2.4 最適化アルゴリズム

元データとその数値的な表現、モデル、上手く定義された目的関数があれば、ロスを最小化するような最適なパラメータを探索するアルゴリズムが必要になります。ニューラルネットワークにおける最も有名な最適化アルゴリズムは、最急降下法と呼ばれる方法にもとづいています。端的に言えば、パラメーラを少しだけ動かしたとき、学習データに対するロスがどのような方向に変化するかを、パラメータごとに見ることです。ロスが小さくなる方向へパラメータを更新します。

4.3 さまざまな機械学習

以下の節では、機械学習の一部の種類についてもう少し詳しく説明します。まず、目的のリスト、すなわち機械学習ができることのリストから始めます。その目的は実現のための一連の技術、言い換えれば、学習やデータの種類などによって補完的に記述されています。以下のリストは、読者を動機づけるために、そして問題について会話するための共通言語を示すためのものです。より多くの問題については、この書籍を進めていく上で紹介していきます。

4.3.1 教師あり学習

教師あり学習は、入力データが与えられて、何らかの対象を予測するものです。その対象というのは、ラベルと呼ばれていて、yで表現することが多いです。入力データ点は、*example* や *instance* と呼ばれることがあり、$\mathbf{x}$で表現されることが多いです。教師あり学習の目的は、入力$\mathbf{x}$から予測$f_\theta(\mathbf{x})$を得るようなモデルf_θを生成することです。

この説明を例を使って具体的に説明したいと思います。ヘルスケアの分野で仕事をしているとき、患者が心臓発作を起こすかどうかを予測したいと思います。実際に観測した内容、この場合、心臓発作か心臓発作でないかがラベルyです。入力データ$\mathbf{x}$は、心拍数や、最高、最低の血圧といった、いわゆるバイタルサインというものになるでしょう。

教師あり学習における、教師するというのは、パラメータθを選ぶためのもので、私達が教師となって、ラベルの付いた入力データ$(\boldsymbol{x}_i, y_i)$とモデルを与えます。各データ$\boldsymbol{x}_i$は正しいラベルと対応しています。

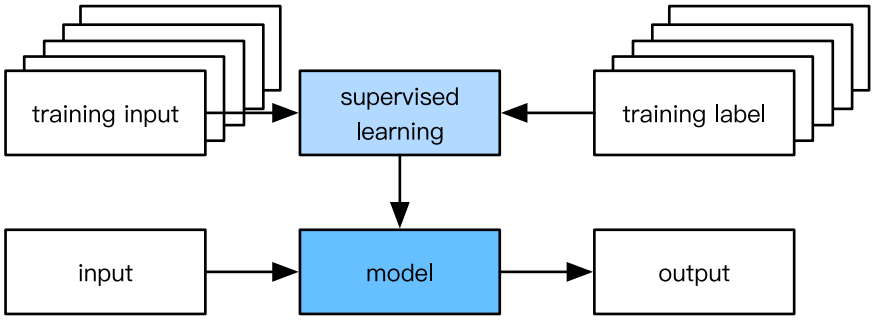
確率の専門用語を用いれば、私達は条件付き確率$P(y|x)$を求めようとしています。機械学習には、いくつかのアプローチがありますが、教師あり学習は実際に利用されている機械学習の大半を占めると言って良いでしょう。重要なタスクは、与えられた事実から未知の事象の確率を推測すること、と大雑把に言うことができるでしょう。例えば、

- **CT画像からガンかガンでないかを予測する**
- **英語の文章から正しいフランス語の翻訳を予測する**
- **今月の財務データから翌月の株価を予測する**

「入力から対象を予測する」というシンプルな説明をしましたが、教師あり学習は、入出力データのタイプ、サイズ、数に応じて、様々な形式やモデル化の方法をとることができます。例えば、文字列や時系列データなどの系列データを処理する場合と、固定長のベクトルで表現されるデータを処理する場合で、異なるモデルを利用することができます。この書籍の序盤では、これらの問題の多くについて深く説明したいと思います。

簡潔に言うと、機械学習のプロセスというのは、こういったものです。互いに関係するデータを大量に収集し、そこからランダムにサブセットを取り出して、各データの正解ラベル (Ground Truth) を取得します。これらのラベルは、収集したデータの中で利用可能な場合もありますし (例えば、ある患者が翌年に亡くなったかどうかなど)、データにラベル付けを行う人を雇う必要があるかもしれません (画像をカテゴリにわけるなど)。

Together, these inputs and corresponding labels comprise the training set. We feed the training dataset into a supervised learning algorithm, a function that takes as input a dataset and outputs another function, *the learned model*. Finally, we can feed previously unseen inputs to the learned model, using its outputs as predictions of the corresponding label. The full process in drawn in :numref:`fig_supervised_learning`.



:label:`fig_supervised_learning`

回帰

最も単純な教師あり学習のタスクとして頭に思い浮かぶものは、おそらく回帰ではないかと思います。例えば、住宅の売上に関するデータベースから、一部のデータセットが得られた場合を考えてみます。各列が異なる住居に、各列は関連する属性、例えば、住宅の面積、寝室の数、トイレの数、中心街まで徒歩でかかる時間に対応するような表を構成するでしょう。形式的に、このようなデータセットの1行を特徴ベクトル、関連する対象（今回の事例では1つの家）をデータ例と呼びます。

もしニューヨークやサンフランシスコに住んでいて、Amazon、Google、Microsoft、FacebookなどのCEOでなければ、その特徴ベクトル（面積、寝室数、トイレ数、中心街までの距離）は$[100, 0, .5, 60]$といった感じでしょう。一方、もしピッツバーグに住んでいれば、その特徴ベクトルは$[3000, 4, 3, 10]$のようになると思います。このような特徴ベクトルは、伝統的な機械学習のあらゆる問題において必要不可欠なものでした。あるデータ例に対する特徴ベクトルを$\mathbf{x_i}$で、全てのテータ例の特徴ベクトルをXとして表します。

何が問題を回帰させるかというと、実はその出力なのです。もしあなたが、新居を購入しようと思っていて、上記のような特徴量を用いて、家の適正な市場価値を推定したいとしましょう。目標値は、販売価格で、これは実数です。If you remember the formal definition of the reals you might be scratching your head now. Homes probably never sell for fractions of a cent, let alone prices expressed as irrational numbers. In cases like this, when the target is actually discrete, but where the rounding takes place on a sufficiently fine scale, we will abuse language just a bit cn continue to describe our outputs and targets as real-valued numbers.

あるデータ例$\mathbf{x_i}$に対する個別の目標値をy_iとし、すべてのデータ例$\mathbf{X}$に対応する全目標を$\mathbf{y}$とします。目標値が、ある範囲内の任意の実数をとるとき、この問題を回帰問題と呼びます。ここで作成するモデルの目的は、実際の目標値に近い予測値(いまの例では、価格の推測値)を生成することです。この予測値を$\hat{y}_i$とします。もしこの表記になじみがなければ、いまのところ無視しても良いです。以降の章では、中身をより徹底的に解説していく予定です。

多くの実践的な問題は、きちんと説明されて、わかりやすい回帰問題となるでしょう。ユーザがある動画につけるレーティングを予測する問題は回帰問題です。もし2009年にその功績をあげるような偉大なアルゴリズムを設計できていれば、Netflixの100万ドルの賞を勝ち取っていたかも知れません。病院での患者の入院日数を予測する問題もまた回帰問題です。ある1つの法則として、どれくらい?という問題は回帰問題を示唆している、と判断することは良いかも知れません。

- 'この手術は何時間かかりますか?' - 回帰
- 'この写真にイヌは何匹いますか?' - 回帰.

しかし、「これは__ですか?」のような問題として簡単に扱える問題であれば、それはおそらく分類問題で、次に説明する全く異なる種類の問題です。たとえ、機械学習をこれまで扱ったことがなかったとしても、形式に沿わない方法で、おそらく回帰問題を扱うことができるでしょう。例えば、下水装置を直すことを考えましょう。工事業者は下水のパイプから汚れを除くのに$x_1 = 3$時間かかって、あなたに$y_1 = 350$の請求をしました。友人が同じ工事業者を雇い、$x_2 = 2$時間かかったとき、友人に$y_2 = 250$の請求をしました。もし、これから汚れを除去する際に、どれくらいの請求が発生するかを尋ねられたら、作業時間が長いほど料金が上がるといった、妥当な想定をすると思います。そして、基本料金があって、１時間当たりの料金もかかるという想定もするでしょう。これらの想定のもと、与えられた2つのデータを利用して、工事業者の値付け方法を特定できるでしょう。1時間当たり$100で、これに加えて$50の出張料金です。もし、読者がこの内容についてこれたのであれば、線形回帰のハイレベルな考え方をすでに理解できているでしょう (そして、バイアス項のついた線形モデルを暗に設計したことになります)。

この場合では、工事業者の価格に完全に一致するようなパラメータを作ることができましたが、場合によってはそれが不可能なことがあります。例えば、2つの特徴に加えて、いくらかの分散がある要因によって発生する場合です。このような場合は、予測値と観測値の差を最小化するようにモデルを学習します。本書の章の多くでは、以下の2種類の一般的なロスのうち1つに着目します。以下の式で定義されるL1ロスと

$$l(y, y') = \sum_i |y_i - y'_i|$$

以下の式で定義される最小二乗ロス、別名L2ロスです。

$$l(y, y') = \sum_i (y_i - y'_i)^2.$$

のちほど紹介しますが、L_2ロスはガウスノイズによってばらついているデータを想定しており、L_1ロスはラプラス分布のノイズによってばらついていることを想定しています。

分類

回帰モデルは「どれくらい多い?」という質問に回答する上で、優れたものではありますが、多くの問題はこのテンプレートに都合よく当てはめられるとは限りません。例えば、ある銀行ではモバイルアプリに、領収書をスキャンする機能を追加したいと思っています。これには、スマートフォンのカメラで領収書の写真を撮る顧客と、画像内のテキストを自動で理解できる必要のある機械学習モデルが関係するでしょう。より頑健に手書きのテキストを理解する必要もあるでしょう。この種のシステムは光学文字認識 (Optical Character Recognition, OCR) と呼ばれており、OCRが解くこの種の問題は分類と呼ばれています。分類の問題は、回帰に利用されたアルゴリズムとは異なるアルゴリズムが利用されます (ただし、中の技術は同様のものが利用されます)。

分類において、ある画像データのピクセル値のような特徴ベクトルにもとづいて、いくつかの候補の中から、そのデータがどのカテゴリ（正式にはクラス)を予測したいと思うでしょう。例えば、手書きの数字に対しては、0から9までの数値に対応する10クラスになるでしょう。最も単純な分類というのは、2クラスだけを対象にするとき、すなわち2値分類と呼ばれる問題です。例えば、データセットXが動物の画像で構成されていて、ラベルYが$\{\boxtimes\boxtimes, \boxtimes\boxtimes\}$のクラスであるとします。回帰の場合、実数$\hat{y}$を出力するような回帰式を求めていたでしょう。分類では、出力$\hat{y}$は予測されるクラスとなるような分類器を求めることになるでしょう。

この本が目的としているように、より技術的な内容に踏み込んでいきましょう。ネコやイヌといったカテゴリに対して、0か1かで判定するようなモデルを最適化することは非常に難しいです。その代わりに、確率を利用してモデルを表現するほうがはるかに簡単です。あるデータxに対して、モデルは、ラベルkとなる確率$\hat{y}_k$を決定します。これらは確率であるため、正の値であり、その総和は1になります。したがって、K個のカテゴリの確率を決めるためには、$K-1$の数だけ必要であることがわかります。2クラス分類の場合がわかりやすいです。表を向く確率が0.6 (60%) の不正なコインでは、0.4 (40%)の確率で裏を向くでしょう。動物を認識する例に戻ると、分類器は画像を見てネコである確率 $\Pr(y = \text{cat}|x) = 0.9$を出力します。その数値は、画像がネコを表していることを、90%の自信をもって分類器が判定したものだと解釈できるでしょう。ある予測されたクラスに対する確率の大きさは信頼度の一つです。そしてこれは、唯一の信頼度というわけではなく、より発展的な章において、異なる不確実性について議論する予定です。

二つより多いクラスが考えられるときは、その問題を多クラス分類と呼びます。一般的な例として、`[0, 1, 2, 3 ... 9, a, b, c, ...]`のような手書き文字の認識があります。L1やL2のロス関数を最小化することで回帰問題を解こうとしてきましたが、分類問題に対する一般的なロス関数はcross-entropyと呼ばれるものです。MXNet Gluonにおいては、対応する関数がここに記載されています

分類器が最も可能性が高いと判断したクラスが、行動を決定づけるものになるとは限りません。例えば、裏庭で美しいキノコを発見したとします。

タマゴテングタケという毒キノコ - 決して食べてはいけません!

ここで分類器を構築して、画像のキノコが毒キノコかどうかを判定するように分類器を学習させます。そして、その毒キノコ分類器が毒キノコ(death cap)である確率Pr(y =

deathcap|image) = 0.2を出力したとします。言い換えれば、そのキノコは80%の信頼度で毒キノコではないと判定されているのです。しかし、それを食べるような人はいないでしょう。このキノコでおいしい夕食をとる利益が、これを食べて20%の確率で死ぬリスクに見合わないからです。言い換えれば、不確実なリスクが利益よりもはるかに重大だからです。数学でこのことを見てみましょう。基本的には、私たちが引き起こすリスクの期待値を計算する必要があります。つまり、それに関連する利益や損失と、起こりうる確率を掛け合わせます。

$$L(\text{action}|x) = \mathbf{E}_{y \sim p(y|x)}[\text{loss}(\text{action}, y)]$$

そして、キノコを食べることによるロスLは$L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$で、これに対して、キノコを食べない場合のロス$L$は$L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$です。

私たちの注意は正しかったのです。キノコの学者はおそらく、上のキノコは毒キノコであるというでしょう。分類は、そうした2クラス分類、多クラス分類または、マルチラベル分類よりもずっと複雑になることもあります。例えば、階層構造のような分類を扱うものもあります。階層では、多くのクラスの間に関係性があることを想定しています。そして、すべての誤差は同じではありません。つまり、関係性のない離れたクラスを誤分類することは良くないですが、関係性のあるクラスを誤分類することはまだ許容されます。通常、このような分類は階層分類と呼ばれます。初期の例はリンネによるもので、彼は動物を階層的に分類・組織しました。

動物の階層的分類の場合、プードルをシュナウザーに間違えることはそこまで悪いことでないでしょうが、プードルを恐竜と間違えるようであれば、その分類モデルはペナルティを受けるでしょう。階層における関連性は、そのモデルをどのように使うかに依存します。例えば、ガラガラヘビとガータースネークの2種類のヘビは系統図のなかでは近いかもしれませんが、猛毒をもつガラガラヘビを、そうでないガータースネークと間違えることは命取りになるかもしれません。

タグ付け

いくつかの分類問題は、2クラス分類や多クラスのような設定にうまく合わないことがあります。例えば、イヌをネコと区別するための標準的な2クラス分類器を学習するとしましょう。コンピュータービジョンの最新の技術をもって、既存のツールで、これを実現することができます。それにも関わらず、モデルがどれだけ精度が良くなったとしても、ブレーメンの音楽隊のような画像が分類器に入力されると、問題が起こることに気づきませんか。


:width:300px

画像から確認できる通り、画像にはネコ、オンドリ、イヌ、ロバが、背景の木とともに写っています。最終的には、モデルを使ってやりたいことに依存はするのですが、この画像を扱うのに2クラス分類を利用することは意味がないでしょう。代わりに、その画像において、ネコとイヌとオンドリとロバのすべて写っていることを、モデルに判定させたいと考えるでしょう。

どれか1つを選ぶのではなく、相互に排他的でないクラスを予測するように学習する問題は、マ

ルチラベル分類と呼ばれています。ある技術的なブログにタグを付与する場合を考えましょう。例えば、'機械学習'、'技術'、'ガジェット'、'プログラミング言語'、'linux'、'クラウドコンピューティング'、'AWS'です。典型的な記事には5から10のタグが付与されているでしょう。なぜなら、これらの概念は互いに関係しあっているからです。'クラウドコンピューティング'に関する記事は、'AWS'について言及する可能性が高く、'機械学習'に関する記事は'プログラミング言語'も扱う可能性があるでしょう。

また、生体医学の文献を扱うときにこの種の問題を処理する必要があります。論文に正確なタグ付けをすることは重要で、これによって、研究者は文献を徹底的にレビューすることができます。アメリカ国立医学図書館では、たくさんの専門的なタグ付けを行うアノテータが、PubMedにインデックスされる文献一つ一つを見て、2万8千のタグの集合であるMeSHから適切な用語を関連付けます。これは時間のかかる作業で、文献が保管されてからタグ付けが終わるまで、アノテータは通常1年の時間をかけます。個々の文献が人手による正式なレビューを受けるまで、機械学習は暫定的なタグを付与することができるでしょう。実際のところ、この数年間、BioASQという組織がこの作業を正確に実行するためのコンペティションを行っていました。

検索とランキング

上記の回帰や分類のように、データをある実数値やカテゴリに割り当てない場合もあるでしょう。情報検索の分野では、ある商品の集合に対するランキングを作成することになります。Web検索を例にとると、その目的はクエリに関係する特定のページを決定するだけでは十分ではなく、むしろ、大量の検索結果からユーザに提示すべきページを決定することにあります。私達はその検索結果の順番を非常に気にします。学習アルゴリズムは、大きな集合から取り出した一部の集合について、要素に順序をつける必要があります。言い換えれば、アルファベットの最初の5文字を対象としたときに、`A B C D E`と`C A B E D`には違いがあるということです。たとえ、その集合が同じであっても、その集合の中の順序は重要です。

この問題に対する1つの解決策としては、候補となる集合のすべての要素に関連スコアをつけて、そのスコアが高い要素を検索することでしょう。PageRankはその関連スコアの初期の例です。変わった点の1つとしては、それが実際のクエリに依存しないということです。代わりに、そのクエリの単語を含む結果に対して、単純に順序をつけています。現在の機械学習エンジンは、クエリに依存した関連スコアを得るために、機械学習と行動モデルを利用しています。このトピックのみを扱うようなカンファレンスも存在します。

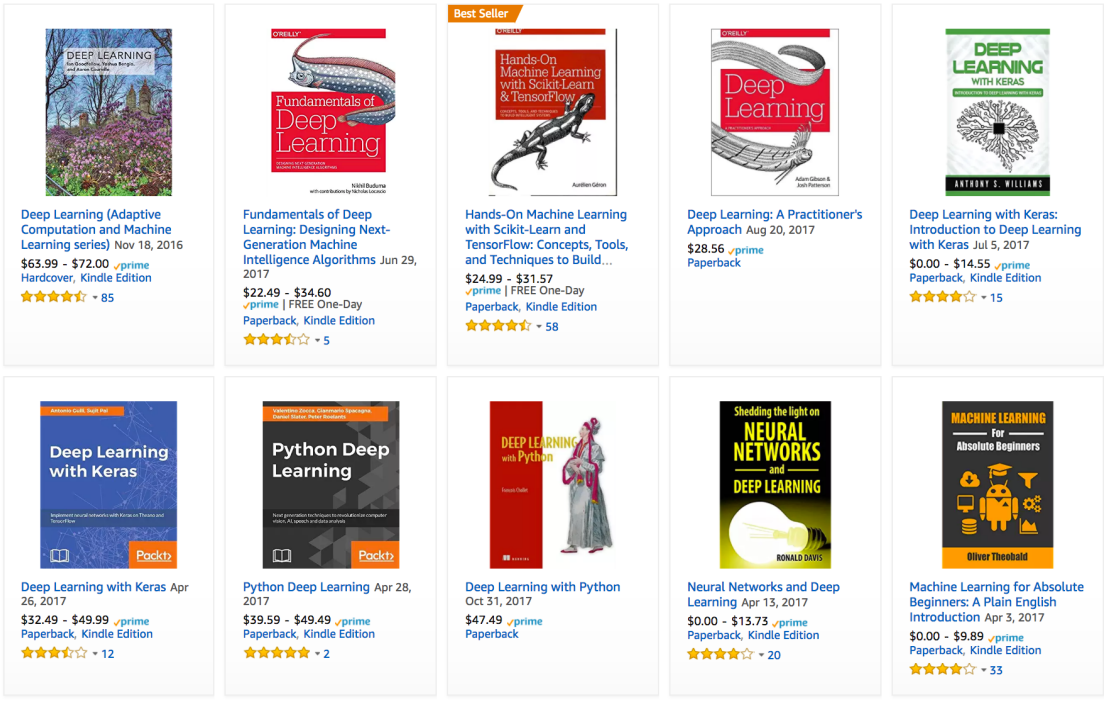
推薦システム

:label:`subsec_recommender_systems`

推薦システムは、検索とランキングに関係するもう一つの問題です。その問題は、ユーザに関連する商品群を提示するという目的においては似ています。主な違いは、推薦システムにおいて特定のユーザの好みに合わせる(*Personalization*)に重きをおいているところです。例えば、映画の推薦の場合、SFのファン向けの推薦結果は、Woody Allenのコメディに詳しい人向けの推薦結果とは大きく異なるでしょう。そのような問題は、映画、製品、音楽の推薦においても見られます。

ときどき、顧客はその商品がその程度好きなのかということを陽に与える場合があります(例えば、Amazonの商品レビュー)。また、プレイリストでタイトルをスキップするように、結果に満足しない場合に、そのフィードバックを送ることもあります。一般的に、こうしたシステムでは、あるユーザu_iと商品p_iがあたえられたとき、商品に付与する評価や購入の確率などを、スコアy_iとして見積もることに力を入れています。

そのようなモデルが与えられると、あるユーザに対して、スコアy_{ij}が最も大きい商品群を検索することできるでしょう。そして、それが推薦として利用されるのです。実際に利用されているシステムはもっと先進的で、スコアを計算する際に、ユーザの行動や商品の特徴を詳細に考慮しています。次の画像は、著者の好みに合わせて調整したpersonalizationのアルゴリズムを利用して、Amazonで推薦される深層学習の書籍の例を表しています。



:label:`fig_deeplearning_amazon`

Despite their tremendous economic value, recommendation systems naively built on top of predictive models suffer some serious conceptual flaws. To start, we only observe *censored feedback*. Users preferentially rate movies that they feel strongly about: you might notice that items receive many 5 and 1 star ratings but that there are conspicuously few 3-star ratings. Moreover, current purchase habits are often a result of the recommendation algorithm currently in place, but learning algorithms do not always take this detail into account. Thus it is possible for feedback loops to form where a recommender system preferentially pushes an item that is then taken to be better (due to greater purchases) and in turn is recommended even more frequently. Many of these problems about how to deal with censoring, incentives, and feedback loops, are important open research questions.

系列学習

これまで、固定長の入力に対して、固定長の出力を生成する問題について見てきました。これまでに、広さ、ベッドルームの数、バスルームの数、ダウンタウンまでの徒歩時間といった特徴の決められた集合から住宅価格を予測することを考えました。また、（固定次元の）画像から、それが固定数のクラスに属する確率へのマッピング、またはユーザーIDと製品IDからユーザのレーティングを予測することも説明しました。このような場合、固定長の入力をモデルに渡して出力を生成すると、モデルはさきほど見たデータをすぐに忘れてしまいます。

入力が実際にすべて同じ次元をもち、連続する入力が本当に互いに何の関係もない場合、問題はないでしょう。しかし、断片的な動画を扱う場合はどうでしょうか。この場合、各動画は異なる数のフレームで構成される場合があります。そして、各フレームで起こっていることを推測する際、前後のフレームを考慮に入れることによって、より強い推測とすることができるでしょう。言語についても同様です。よく知られている深層学習の問題の1つに機械翻訳があります。それは、ある翻訳元の言語の文を取り込んで、別の言語への翻訳を予測するタスクです。

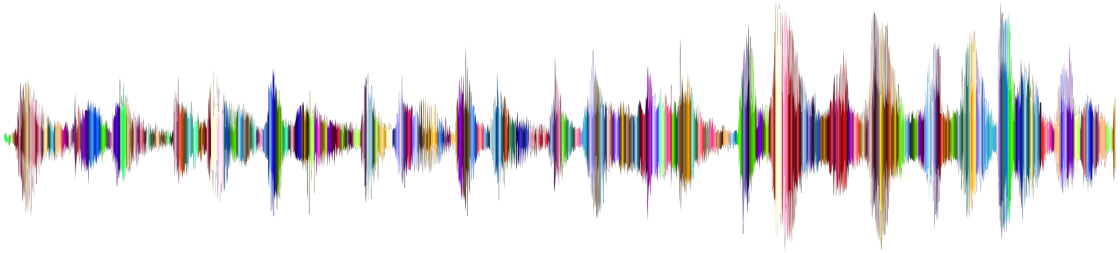
これらの問題は医療の分野でも起こります。集中治療室で患者を監視し、次の24時間以内に死亡するリスクがあるしきい値を超えた場合、警告を発するモデルを望んでいるとしましょう。この場合、モデルが過去の1時間ごとに患者の状況を全て捨て去り、ただ最新の測定値にもとづいて予測する、といったことはきっと望まないはずです。

これらの問題は、機械学習のより刺激的なアプリケーションの1つであり、それらは系列学習の一つです。入力の系列を取り込むか、または出力の系列を生成する（あるいはその両方）モデルが必要です。系列を生成する後者の問題は seq2seqの問題と呼ばれることがあります。言語翻訳は seq2seqの問題です。会話の音声をテキストを書き起こすことも seq2seq問題です。すべてのタイプの系列変換を考慮することは不可能ですが、いくつかの特別な場合について以下で述べます。

文章のタグ付けとパーステキストの系列に属性を付けることも、タグ付けやパースの一つです。入力と出力の数は基本的に同じです。例えば、動詞と主語がどこにあるかを知りたいと思うかもしれません。あるいは、どの単語が固有表現であるかを知りたいと思うかもしれません。一般に、アノテーションを行うための構造や文法に関する前提にもとづいて、文章を分解したりアノテーションしたりすることが目標になります。これは実際よりも複雑に思えるかもしれません。以下は、どの単語が固有表現であるかを示すタグを文に付与する非常に簡単な例です。

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

自動音声認識音声認識では、入力の系列データxは話者の音声であり、出力yは話者が話した内容を書き起こしたテキストです。難しい点としては、テキストよりもはるかに多くの音声フレーム（音声は通常8kHzまたは16kHzでサンプリングされる）があることです。すなわち何千もの音声サンプルが、話された単語の一つに対応するので、オーディオとテキストとの間に1:1の対応がありません。これらは seq2seqの問題ですが、出力は入力よりずっと短い系列です。



:width:700px :label:fig_speech

テキスト読み上げテキスト読み上げ (TTS, Text-To-Speech)は音声認識の逆です。つまり、入力xはテキストで、出力yは音声です。この場合、出力は入力よりもはるかに長い系列となります。人間にとっては品質の悪い音声を認識するのは簡単ですが、コンピュータにとってそれほど簡単ではありません。

機械翻訳音声認識では、対応する入力および出力が同じ順序で発生しますが (ある程度、整列は必要ですが)、機械翻訳では、順序の逆転を生じる可能性があります。言い換えれば、ある系列データを別の系列データに変換している間は、入力と出力の数も、対応するデータの順序も、同じと考えてはいけません。以下の例は、動詞を文の末尾に置くというドイツ語の傾向を表したものです。

```
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?
```

他にも関連する問題はあります。例えば、ユーザがウェブページを読む順序を決定する問題は、二次元の配置分析問題と考えることができます。同様に、自動的な対話を実現するためには、実世界の知識であったり、あるいは事前知識のようなものを考慮する必要があり、活発な研究分野となっています。

4.3.2 教師なし学習

これまでのすべての例は教師あり学習、つまり一連のデータと対応する目標値をモデルに与えるというものでした。教師あり学習は、非常に専門的な仕事をもっていて、非常に神経質な上司がいるような状況とみなせるでしょう。上司はあなたを見張って、あなたが状況に対応する行動を学習するまで、あらゆる状況に対して取るべき行動を伝えるでしょう。そのような上司のもとで働くことは、かなり不自由に思えます。一方、この上司を喜ばせるのは簡単です。ただそのパターンをできるだけ早く認識し、その行動を真似するだけです。

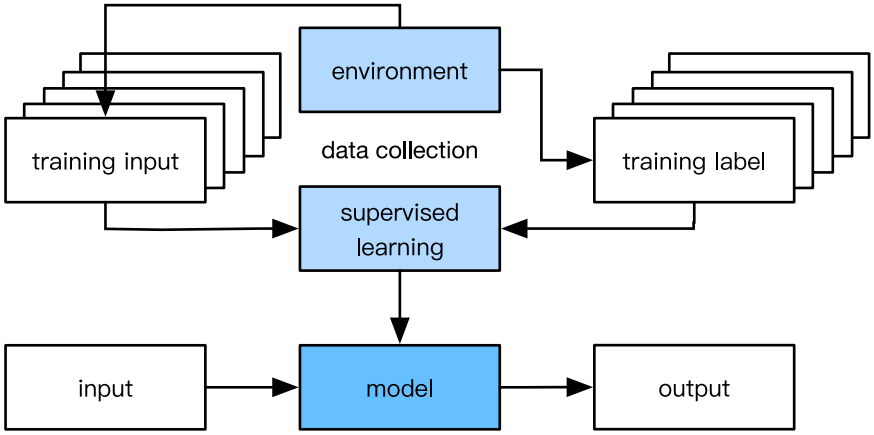
まったく正反対のこととして、あなたに何を望んでいるのかわからない上司のもとで働くことはイライラするかもしれません。しかし、データサイエンティストになろうと思っていれば、それに慣れたほうがいいでしょう。ボスはあなたに巨大なデータを渡して、それを使ってデータサイエンスをやるように指示するかもしれません。このような指示は曖昧に思えるでしょう。このクラスの問題は教師なし学習と呼ばれていて、どのような質問の種類あるいは数に対応できるかは、使う人の創造性に委ねられているのです。以降の章では、教師なしの学習手法

について、いくつか取り上げます。ここではみなさんの動機づけのために、私達はあなたが尋ねるかもしれない質問のいくつかを説明します。

- データを正確に要約して、それらの模範的な少数のデータを見つけることができますか?ある写真の集まりを考えると、それらを風景写真、犬、赤ちゃん、猫、山頂などの写真にグループ分けすることができますか? 同様に、たくさんのユーザーの閲覧動作を考えたとき、同じような動作を行うユーザーへとグループ化できますか? この問題は通常**クラスタリング**として知られています。
- データに関連する特性を正確にとらえた、少数のパラメータを見つけることができますか? ボールの軌道は、ボールの速度、直径、および質量によってうまく説明することができます。仕立て屋は、服を身体に合わせるために、人間の形状をかなり正確に記述するパラメータをいくつか作成しました。これらの問題は**部分空間推定**問題と呼ばれます。依存性が線形の場合、特に**主成分分析**と呼ばれます。
- ユークリッド空間（つまり、$\mathbb{R}^n$のベクトルの空間）において、シンボルの性質がよく適合するような、（任意の構造をもつ）オブジェクトの表現はありますか? これは**表現学習**と呼ばれ、ローマ - イタリア+フランス=パリなど、エンティティとその関係を記述するために利用されます。
- 観察したデータの多くを構成する、根本的な原因を説明できるでしょうか? たとえば、住宅価格、公害、犯罪、場所、教育、給与などに関する人口統計データがある場合、それらがどのように関連しているのか、経験的なデータにもとづいて発見できますか。**有向グラフィカルモデル**と**因果関係**の分野はこれを扱います。
- 最近、重要かつ活発に開発されているものとして**敵対的生成ネットワーク**があります。敵対的生成ネットワークとは、基本的にデータを合成する方法です。根底にある統計的メカニズムとしては、実際のデータと偽のデータが同じかどうかを確認するためのテストを行うというものです。この書籍では、これらに関するノートブックを紹介する予定です。

4.3.3 環境とのインタラクション

これまでのところ、データが実際にどこからもたらされるのか、あるいは機械学習モデルが出力を生成するときに何が実際に起こるのか、については説明していません。教師あり学習と教師なし学習は、非常に洗練された方法で、これらの問題に対処する必要がないからです。どちらの場合も、大量のデータを事前に取得してから、環境とやり取りすることなくパターン認識を行います。すなわち、アルゴリズムが環境から切り離されてから、すべての学習が行われるので、これはオフライン学習と呼ばれます。教師あり学習の場合、プロセスは:numref:`fig_data_collection`のようになります。



:label:fig_data_collection

オフライン学習が単純さは魅力です。したがって、良い点としては他に扱うべき問題とは無関係にパターン認識のみを心配すれば良いという点ですが、一方、欠点としては問題の定式化が非常に限定的になってしまうという点があります。もし、もっと意欲的な人、AsimovのRobot Seriesを読んで成長したような人にとっては、予測を行うだけでなく、実世界で行動を起こすことができる人工知能のボットを想像するかもしれません。そこで、予測型のモデルだけでなく、知的なエージェントについても考えたいと思います。つまり、単に予測を立てるのではなく、アクションを選ぶことを考える必要があるということです。さらに、予測とは異なり、行動は実際には環境に影響を与えます。知的エージェントを訓練したい場合は、その行動がエージェントの将来の観察にどのように影響するかを定義しなければなりません。

環境との相互作用を考えると、モデリングに関する新しい疑問が山ほど出てきます。環境は：

- 私達の過去の行動を記憶しているでしょうか?
- 私たちを助けようとするでしょうか? たとえば、ユーザが音声認識器に向かってテキストを読んでいたりしますか?
- もしくは私達を打ち負かそうとするでしょうか? つまり、スパムをばら撒く人に対するスパムフィルタリングや対戦相手に対するゲームプレイなどのような敵対的な設定でしょうか?
- 何も気にしないでしょうか? (多くの場合でそうあるように)
- 変化するダイナミクス (安定したものと時間の経過とともに変化するもの) がありますか？

この最後の質問は、共変量シフトの問題を引き起こします（トレーニングデータとテストデータが異なる場合)。その問題は、例えば、講師のTAが宿題を作成し、講師が試験wの作成したとき、よく私達が経験するものです。以下では、強化学習と敵対的学習という、環境との相互作用を明示的に考慮する2つの設定について簡単に説明します。

4.3.4 強化学習

機械学習を使用して環境と対話して行動を起こすエージェントを開発することに興味があるなら、おそらく強化学習（RL, Reinforcement Learning）に焦点を当てることになるでしょう。これは、ロボット工学、対話システム、さらに、ビデオゲーム用のAIの開発にも応用されうるでしょう。深層強化学習 (Deep Reinforcement Learning, DRL) は注目を集めています。そのブレークスルーである deep Q-network that beat humans at Atari games using only the visual input と AlphaGo program that dethroned the world champion at the board game Go は有名な例です。

強化学習は、一連の時間ステップにわたって、エージェントが環境と相互作用するという非常に一般的な問題設定を与えます。時間ステップtごとに、エージェントは環境から何らかの観測値o_tを受け取り、そのあと環境に送り返す行動a_tを選択しなければなりません。最後に、エージェントは環境から報酬r_tを受け取ります。その後もエージェントは観測を受け取り、アクションを選択します。強化学習エージェントの振る舞いはポリシーによって管理されています。簡単に言うと、ポリシーは（環境の）観察を行動にマッピングする単なる関数です。強化学習の目標は、良いポリシーを作成することです。

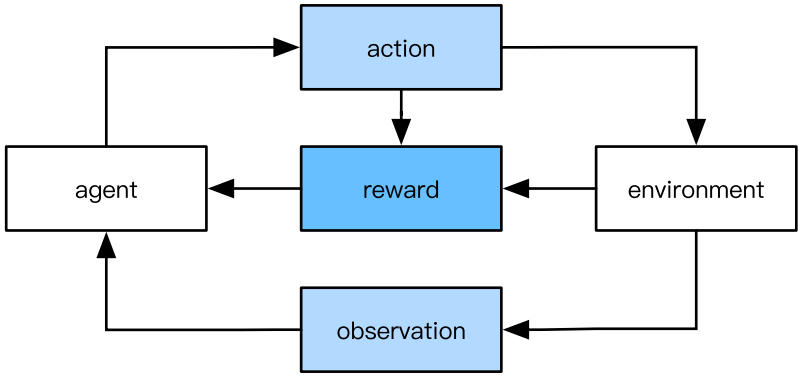


Fig. 4.1: The interaction between reinforcement learning and an environment.

強化学習のフレームワークの一般性を誇張しすぎるのは難しいです。たとえば、教師あり学習の問題を強化学習の問題として扱うことができます。分類問題について考えてみましょう。まず、各クラスに1つの行動が対応するような強化学習エージェントを作成します。そして、元の教師あり学習の問題における損失関数とまったく等しい報酬を与えるような環境を作ります。

強化学習は教師あり学習では不可能な多くの問題にも対処できます。たとえば、教師あり学習では、入力される学習データが正しいラベルに関連付けられていることを仮定します。しかし強化学習では、観測ごとに環境が最適な動作を示していると想定しません。一般的に強化学習では、報酬が与えられるだけで、どの行動が報酬につながったかさえも環境は伝えないかもしれません。

例えばチェスのゲームを考えてみましょう。唯一の報酬として、勝ったときに1、負けたときに-1の報酬を割り当てることができ、これらはゲームの終わりになってはじめて得られるもの

です。強化学習を行う際は、この信用割当問題 *(credit assignment problem)*に対処する必要があります。同じことは、10月11日に昇進した、とある従業員にも当てはまります。その昇進は、前年に比べて適切に選択された多数の行動の結果と考えられるでしょう。将来に昇進を増やすには、その過程でどのような行動が昇進につながったのかを把握する必要があります。

強化学習を行う際は、部分的にしか観測できないという問題にも対処しなければならないかもしれません。つまり、現在観察している内容からは、現在の状態に関するすべてのことがわかるわけではないのです。家の中に全く同じクローゼットがあったとして、掃除ロボットがそのうちの1つに閉じ込められていることに気づいたとしましょう。ロボットの正確な位置（つまりは状態）を推測するには、クローゼットに入る前の観察を考慮する必要があります。

最後に、強化学習によって、どのような時点であっても、なんらかの良い方針を知ることができるでしょうが、エージェントが試したことがない良い方針が他にもあるかもしれません。強化学習では、ポリシーとして現在知られている最善の戦略を活用 *(exploit)*するか、さらなる知識を得るために短期的な報酬を諦めて戦略の空間を探索するかを絶えず選択する必要があります。

マルコフ決定過程、バンディット問題とその周辺

強化学習の問題は非常に一般的な設定です。行動はその後の観測に影響し、報酬はある行動を実行した結果によってのみ観察されます。環境は完全にまたは部分的に観測されるかもしれません。このような複雑さをすべて一度に説明するためには、研究者に対して非常に多くの質問をする必要があるでしょう。さらに、すべての実践的な問題がこの複雑さをもっているわけではありません。この結果として、研究者たちは強化学習の問題のいくつかの特殊なケースを研究してきました。

環境が完全に観察されるとき、その強化学習の問題を*Markov Decision Process*（MDP）と呼びます。状態が前の行動に依存していないときは*Contextual Bandit Problem*と呼びます。状態が存在せず、最初は報酬が未知であるような行動の集合が与えられているとき、古典的な*multi-armed bandit problem*と呼ばれています。

4.4 起源

深層学習は最近の発明ですが、人間は何世紀にも渡って、データの分析や将来の予測をしたいと考えてきました。実際、自然科学の多くが深層学習の起源になっています。例えば、ベルヌーイ分布はJacob Bernoulli (1655-1705)の名をとっており、ガウス分布はCarl Friedrich Gauss (1777-1855)によって発見されました。彼は、例えば最小二乗法を発見し、今日でも保険の計算や医療診断に至る様々な問題に利用されています。これらのツールは自然科学における実験的なアプローチによってうまれたものです。例えば、抵抗における電流と電圧に関係するオームの法則は、完全に線形モデルとして記述することができます。

中世においても、数学者は予測に対して熱意を注いできました。例えば、Jacob Köbel (1460-1533)による幾何学の本は、平均的な足の長さを得るために、16人の成人男性の足の長さの平均を計算することを記述しています。

:width:500px :label:fig_koebel

:numref:fig_koebel はこの予測がどの機能するのかを示しています。16人の成人男性が、教会から出る際に1列に並ぶように言われていました。1フィートの値を見積もるために、彼らの足の長さの総和を16で割ったのでした。そのアルゴリズムは後に、特異な長さを処理するように改善されており、最も短いものと最も長いものを取り除き、残りについて平均をとっています。これはtrimmed mean とよばれる推定手法の始まりといえます。

統計学者は、データを収集して、利用できるようにするところからはじめました。偉大な統計学者の一人、Ronald Fisher (1890-1962)は、その理論と遺伝学への応用に関して多大な貢献をしました。彼の多くの線形判別分析のようなアルゴリズムやフィッシャー情報行列といった数式は、今日でも頻繁に利用されています（そして、彼が1936年にリリースしたIrisのデータセットも、機械学習を図解するために利用されています）。

機械学習にとって2番目の影響は、(Claude Shannon, 1916-2001)による情報理論とAlan Turing (1912-1954)による計算理論によるものでしょう。Turingは、Computing machinery and intelligence (Mind, October 1950)という著名な論文の中で、「機械は考えることができるか?」という質問を投げかけました。彼がTuring testと述べていたものは、もし人間が文章で対話をして

いるときに、相手からの返答が機械か人間かどちらか判別がつかなければ、機械は知能があるとみなすものでした。今日に至るまで、知的な機械の開発は急速かつ継続的に変わり続けています。

もう一つの影響は神経科学や心理学の分野にみられるでしょう。人間というものは知的な行動をはっきりと示します。そこで、知的な行動を説明したり、その本質を逆解析したりできないかと考えることは当然のように思います。これを達成した最古のアルゴリズムの1つはDonald Hebb (1904-1985)によって定式化されました。

彼の革新的な書籍 The Organization of Behavior (John Wiley & Sons, 1949)では、ニューロンは正の強化 (Positive reinforcement)によって学習すると彼は断言しています。このことは、Hebbian learning rule として知られるようになりました。。それは、Rosenblatt のパーセプトロンの学習アルゴリズムのプロトタイプであり、今日の深層学習を支えている確率的勾配降下法の多くの基礎を築きました。つまり、ニューラルネットワークにおいて適切な重みを学習するために、求められる行動を強化し、求められない行動を減らすことなのです。

生物学的な観点からの着想という意味では、まず、ニューラルネットワークが名前として与えられている通りです。1世紀以上にわたって（1873年のAlexander Bainや1890年のJames Sherringtonにさかのぼります）、研究者はニューロンを相互作用させるネットワークに類似した計算回路を組み立てようとしていました。時代をこえ、生物学にの解釈はやや薄まってしまいましたが、名前は依然として残っています。その核となる部分には、今日のたいていのネットワークにもみられる重要な原理がわずかに残っています。

- 線形あるいは非線形の演算ユニットについて、しばしば「レイヤー」と呼ばれます。
- 全体のネットワークのパラメータをただちに調整するために連鎖律 (Chain rule, 誤差逆伝播法とも呼ばれる)を使用します。

最初の急速な進歩のあと、ニューラルネットワークの研究は1995年から2005年まで衰えてしまいました。これにはたくさんの理由があります。まず、ネットワークを学習することは膨大な計算量を必要とします。RAMは20世紀の終わりには十分なものとなりましたが、計算力は乏しいものでした。次に、データセットが比較的小さいものでした。実際のところ、1932年から存在するFisherの'Iris dataset'はアルゴリズムの能力をテストするためによく利用されたツールでした。60,000もの手書き数字からなるMNISTは巨大なものと捉えられていました。

データや計算力が乏しいがゆえに、カーネル法や決定木、グラフィカルモデルといった強力な統計的ツールが、経験的に優れていることを示していました。これらのツールは、ニューラルネットワークとは違って、学習に数週間もかかりませんし、強力な理論的保証のもと予測結果をもたらしてくれました。

4.5 深層学習への道程

World Wide Web、数百万のオンラインユーザをかかえる企業の到来、安価で高性能なセンサー、安価なストレージ (Kryderの法則)、安価な計算機 (Mooreの法則)、特に当初はコンピュータゲーム用に開発されたGPUの普及によって、大量のデータを利用することが可能になり、多くのことが変わりました。突然、これまでは計算量的に実行困難と思われたアルゴリズムやモ

デルが価値をもつようになったのです（逆もまた然りで、このようなアルゴリズム、モデルがあったからこそ、データが価値をもったのです）。以下の表は、このことを最も良く表しています。

年代	データセット	メモリ	1秒あたりの浮動小数点演算
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (Nvidia C2050)
2020	1 T (social network)	100 GB	1 PF (Nvidia DGX-2)

非常に明白な点としては、RAMはデータの成長に追いついていないということです。同時に、演算能力は利用可能なデータよりも、速い速度で進歩しています。従って、統計モデルはメモリ効率がより良いものである必要があります（これは通常、非線形性を導入することで実現されます）。また同時に、向上した演算能力によって、より多くの時間をパラメータの最適化に使用することができます。次に、機械学習と統計学の大きな強みは、（一般化）線形モデルやカーネル法から深層学習へと移りました。この理由の一つとしては、深層学習の主力である多層パーセプトロン:cite:McCulloch.Pitts.1943、畳み込みニューラルネットワーク:cite:LeCun.Bottou.Bengio.ea.1998、Long Short Term Memory :cite:Hochreiter.Schmidhuber.1997、Q-Learning :cite:Watkins.Dayan.1992 が、長い間眠っていた後に、ここ10年で根本的に再発見されたことが挙げられます。

統計モデル、アプリケーション、そしてアルゴリズムにおける最近の進歩は、カンブリア紀の爆発、つまり生命の種の進化における急速な発展とリンクするかもしれません。実際のところ、現在の最新の技術というものは、利用可能なリソースから成り行きでできあがったものではなく、過去数十年のアルゴリズムにも利用されているのです。

- ネットワークの能力を制御する新しい手法の一例であるDropout :cite:Srivastava.Hinton.Krizhevsky.ea.2014 は、過学習のリスクをおさえる、つまり大部分の学習データをそのまま記憶してしまうことを避けて、比較的大きなネットワークを学習することを可能にします。これは、ノイズをネットワークに追加することによって実現され:cite:Bishop.1995、学習のために重みを確率変数に置き換えます。
- Attentionの仕組みは、1世紀に渡って統計学の分野で悩みとなっていた問題を解決しました。その問題とは、学習可能なパラメータの数を増やすこと無く、システムの記憶量や複雑性を向上させるというものです。:cite:Bahdanau.Cho.Bengio.2014 らは、学習可能なポインタ構造とみなせる方法を利用した、洗練された解法を発見しました。ある固定次元の表現を利用した機械翻訳を例にとると、全体の文章を記憶するというよりはむしろ、翻訳プロセスにおける中間表現へのポインタを保存すればよいということを意味します。翻訳文のような文章を生成するまえに、翻訳前の文章全体を記憶する必要がもはやなくなり、長文に対して大きな精度の向上を可能にしました。
- Memory Networks :cite:Sukhbaatar.Weston.Fergus.ea.2015 やNeural Programmer-Interpreter :cite:Reed.De-Freitas.2015のような多段階設計は、統計的なモデル化を行う人々に対して、推論のための反復的なアプローチを記述可能にし

ます。これらの手法は、繰り返し修正される深層学習の内部状態を想定し、推論の連鎖において以降のステップを実行します。それは、計算のためにメモリの内容が変化するプロセッサに似ています。

- もうひとつの重要な成果は、Generative Adversarial Networks :cite:Goodfellow.Pouget-Abadie.Mirza.ea.2014 の発明でしょう。従来、確率密度を推定する統計的手法と生成モデルは、妥当な確率分布を発見することと、その確率分布からサンプリングを行う（しばしば近似的な）アルゴリズムに着目してきました。結果としてこれらのアルゴリズムは、統計的モデルにみられるように柔軟性が欠けており、大きく制限されたものになっていました。GANにおける重要な発明は、そのサンプリングを行うsamplerを、微分可能なパラメータをもつ任意のアルゴリズムで置き換えたことにあります。これらのパラメータは、discriminator (真偽を判別するテスト)が、真のデータと偽のデータの区別がつかないように調整されます。任意のアルゴリズムでそのデータを生成できるため、様々な種類の技術において、確率密度の推定が可能になりました。人が乗って走るシマウマ:cite:Zhu.Park.Isola.ea.2017 や偽の有名人の顔:cite:Karras.Aila.Laine.ea.2017 の例は、両方とも、この技術の進歩のおかげです。
- 多くの場合において、学習に利用可能な大量のデータを処理するにあたって、単一のGPUは不十分です。過去10年にわたって、並列分散学習を実装する能力は非常に進歩しました。スケーラブルなアルゴリズムを設計する際に、最も重要な課題のうちの一つとなったのは、確率的勾配降下法のような深層学習の最適化の能力が、処理される比較的小さなミニバッチに依存するという点です。同時に、小さなバッチはGPUの効率を制限します。それゆえに、ミニバッチのサイズが、例えば1バッチあたり32画像で、それを1024個のGPUで学習することは、32,000枚の画像の集約されたミニバッチと等しくなります。最近の研究として、まずはじめにLi :cite:Li.2017、次にYouら:cite:You.Gitman.Ginsburg.2017、そしてJiaら:cite:Jia.Song.He.ea.2018 が、そのサイズを64,000の観測データにまで向上させ、ImageNetを利用したResNet50の学習時間を7分以下にまで短縮しました。比較のため、当初の学習時間は日単位で計測されるようなものでした。
- 計算を並列化する能力は、強化学習の発展、少なくともシミュレーションが必要となる場面において、大きな貢献を果たしています。これによって、碁、Atariのゲーム、Starcraftにおいてコンピュータが人間の能力を超え、物理シミュレーション(例えば、MuJoCoの利用)においても大きな発展をもたらしました。AlphaGoがこれをどのように達成したかを知りたい場合は、Silverらの文献:cite:Silver.Huang.Maddison.ea.2016 を見てください。強化学習が最も上手くいくのは、(状態, 行動, 報酬)の3つ組が十分に利用できるとき、つまり、それらの3つ組が互いにどう関係し合うかを学習するために多くの試行が可能なときなのです。シミュレーションはそのための手段を与えてくれます。
- 深層学習のフレームワークは、考えを広める上で重要な役割を果たしてきました。容易なモデリングを可能にする最初の世代のフレームワークとしては、Caffe、Torch、Theanoがあります。多くの影響力のある論文はこれらのフレームワークを用いて記述されました。今までに、こうしたフレームワークはTensorFlowや、そのハイレベルなAPIとして利用されるKeras、CNTK、Caffe 2、そしてApache MXNetによって置き換えられてきました。第3世代のフレームワークは、主に深層学習のために必要な指示を記述していくimperativeな手法で、おそらくChainerによって普及し、それはPython NumPyを利用してモデルを記述するのに似た文法を使用しました。この考え方はPyTorchやMXNetのGluon APIにも採用されています。後者のGluon APIは、このコースで深層学習を教えるために利用します。

学習のためにより良い手法を構築するシステム研究者と、より良いネットワークを構築する統計的なモデル化を行う人々の仕事を分けることは、ものごとを非常にシンプルにしてきました。例えば、線形ロジスティック回帰モデルを学習することは簡単な宿題とはいえず、2014年のカーネギーメロン大学における、機械学習を専門とする博士課程の新しい学生に与えるようなものでした。今では、このタスクは10行以下のコードで実装でき、このことはプログラマーによって確かに理解されるようになりました。

4.6 サクセスストーリー

人工知能は、他の方法では実現困難だったものに対して、成果を残してきた長い歴史があります。例えば、郵便は光学式文字読取装置(OCR)によって並び替えらています。これらのシステムは（有名なMNISTやUSPSといった手書き数字のデータセットをもとにして)、90年台に登場したものです。同様のことが、預金額の読み取りや、申込者の信用力の評価にt形容されています。金融の取引は不正検知のために自動でチェックされています。これによって、PayPal、Stripe、AliPay、WeChat、Apple、Visa、MasterCardといった多くの電子商取引の支払いシステムの基幹ができあがりました。機械学習は、インターネット上の検索、レコメンデーション、パーソナライゼーション、ランキングも動かしています。言い換えれば、人工知能と機械学習はあらゆるところに浸透していて、しばしば見えないところで動いているのです。

最近になって、以前は解くことが困難と思われた問題へのソリューションとして、AIが脚光を浴びています。

- AppleのSiri、AmazonのAlexa、Google assistantといった知的なアシスタントは、利用可能なレベルの精度で話し言葉の質問に回答することができます。これらは、証明のスイッチをONにする（身体障害者にとってはありがたい)、理髪店の予約をする、電話サポートにおける会話を提供する、といった技術を必要としないタスクをこなします。これは、AIがわれわれの生活に影響をもたらしている最も顕著なサインであるように思えます。
- デジタルなアシスタントなかで重要な構成要素となっているのは、音声を精度よく認識する能力です。音声認識の精度は、特定のアプリケーションにおいて、人間と同等の精度にまで徐々に改善してきました [14]。
- 同様に物体の認識も長い道のりを経ています。画像内の物体を推定することは2010年においては全く困難なタスクでした。ImageNetのベンチーマークにおいては、Linら[15]がtop-5のエラー率 (推定結果の上位5件に正解がない割合)は28%でした。2017年までには、Huら[16]がこのエラー率を2.25%まで低減しました。同様に素晴らしい結果が、鳥の認識や皮膚がんの診断においても達成されています。
- 以前はゲームは人間の知性の砦だったでしょう。Temporal difference (TD)を利用した強化学習でBackgrammonをプレイするTDGammon[23]から始まり、アルゴリズムや演算能力の進歩は、幅広いアプリケーションを対象としたアルゴリズムをリードしてきました。Backgammonとは異なり、チェスはさらに複雑な状態空間と行動集合をもっています。DeepBlueは、Campbellら:cite:`Campbell.Hoane-Jr.Hsu.2002` が巨大な並列処理、特別なハードウェア、ゲーム木による効率な探索を利用し、Gary Kasparovを破りました。囲碁はその巨大な状態空間のためにさらに困難です。AlphaGoは2015年に人間と肩を並

べるようになり、Silverら:cite:Silver.Huang.Maddison.ea.2016 はモンテカルロ木のサンプリングと深層学習を組み合わせて利用しました。ポーカーにおいては、その状態空間は大規模で完全に観測されない（相手のカードを知ることができない）という難しさがあります。Libratusは、BrownとSandholm :cite:Brown.Sandholm.2017 による、効率的で構造化された戦略を用いることで、ポーカーにおいて人間の能力を超えました。このことは、ゲームにおける優れた進歩と、先進的なアルゴリズムがゲームにおける重要な部分で機能した事実を表しています。

- AIの進歩における別の兆候としては、車やトラックの自動運転の登場でしょう。完全自動化には全く及んでいないとはいえ、この方向性には重要な進歩がありました。例えば、Momenta、Tesla, NVIDIA、MobilEye、Waymoは、少なくとも部分的な自動化を可能にした商品を販売しています。完全自動化がなぜ難しいかというと、正しい運転にはルールを認識して論理的に考え、システムに組み込むことが必要だからです。現在は、深層学習はこれらの問題のうち、コンピュータビジョンの側面で主に利用されています。残りはエンジニアによって非常に調整されているのです。

繰り返しますが、上記のリストは知性というものがとういうもので、機械学習がある分野において素晴らしい進歩をもたらしたということについて、表面的な内容を走り書きしたに過ぎません。例えば、ロボティクス、ロジスティクス、計算生物学、粒子物理学、天文学においては、それらの優れた近年の進歩に関して、部分的にでも機械学習によるところがあります。機械学習は、こうしてエンジニアやサイエンティストにとって、広く利用されるツールになったのです。

しばしば、AIの黙示録や特異点に関する質問が、AIに関する技術的でない記事に取り上げられることがあります。機械学習システムが知覚を持ち、プログラマー（や管理者）の意図にかかわらず、人間の生活に直接的に影響を及ぼすものごとを決定することについて、恐れられているのです。ある程度は、AIはすでに人間の生活に直接的に影響を与えています。つまり、信用力は自動で評価されえいますし、自動パイロットは車をほとんど安全にナビゲーションしますし、保釈をしてよいかどうかを決める際には統計データが利用されています。取るに足らない例ではありますが、われわれは、インターネットにつながってさえいれば、Alexaにコーヒーマシンのスイッチを ON にするようお願いでき、Alexaは要望に応えることができるでしょう。

幸運にも人間を奴隷として扱う（もしくはコーヒーを焦がしてしまう）ような知覚のあるAIシステムはまだ遠い未来にあります。まず、AIシステムは、ある目的に特化した方法で開発され、学習され、展開されます。AIシステムの行動はあたかも汎用的な知性をもっているかのように幻想をみせるかもしれませんが、それらは設計されたルールや経験則、統計モデルの組み合わせになっています。次に現時点では、あらゆるタスクを解決するために、自分自身を改善し、それらを論理的に考え、アーキテクチャを修正、拡張、改善するような、汎用人工知能の方法というのは存在していません。

さらに現実的な心配ごととしては、AIがどのように日々の生活に利用されるかです。トラックの運転手や店舗のアシスタントが担っている、技術を必要としないたくさんのタスクは自動化されるでしょう。農業ロボットは有機栽培のコストを下げるでしょうが、収穫作業も自動化してしまうしょう。この産業革命となるフェーズでは、社会における広い範囲に重大な結果を及ぼすでしょう（トラックの運転手や店舗のアシスタントは、多くの州において最も広く行われている仕事です）。さらに、統計モデルは注意せずに利用されれば、人種、性別、年齢による差別を生じる可能性があります。これらのアルゴリズムを、必ず注意して利用ことは重要です。

このことは、人類を滅亡させるような悪意ある超人的な知能や意思を心配するよりもずっと、懸念されることなのです。

4.7 まとめ

- 機械学習は、コンピュータのシステムが性能を改善するために、データの利用方法を研究するものです。それは、統計学、データマイニング、人工知能、最適化の考え方を組み合わせています。そして、人工知能的なソリューションを実装するためによく利用されます。
- 機械学習の一種で、表現学習というものは、データの最適な表現を自動的に探す方法に焦点を当てています。多くの場合、これは学習によるデータ変換の発展によって実現されました。
- 最近の進歩の多くは、安価なセンサー、インターネット上に広がるアプリケーションから得られる豊富なデータと、GPUを主にする演算能力の素晴らしい進歩がきっかけになっています。
- 全体のシステム最適化は、良い性能を得るために重要な構成要素です。効率的な深層学習フレームワークによって、最適化を非常に簡単に設計して、実装することができます。

4.8 課題

1. あなたが現在書いているコードの中で、学習可能な部分はどこでしょうか。言い換えれば、コードの中で設計に関する選択をしている部分で、学習によって改善できて、自動的に決定できる部分がありますか?あなたのコードは、経験則にもとづいて設計に関する選択する部分を含んでいますか?
2. あなたが直面した問題で、その問題を解くのに十分なデータがあるけども、自動化する手法がないような問題はどういうものでしょうか？これらは、深層学習を適用できる最初の候補になるかもしれません。
3. 人工知能の発展を新たな産業革命としてみたとき、アルゴリズムとデータの関係は何にあたるでしょうか?それは蒸気機関と石炭に似ています。根本的な違いは何でしょうか。
4. End-to-endの学習を適用できる他の分野はどこでしょうか? 物理学? 工学?経済学?

4.9 参考文献

[1] Turing, A. M. (1950). Computing machinery and intelligence. Mind, 59(236), 433.

[2] Hebb, D. O. (1949). The organization of behavior; a neuropsychological theory. A Wiley Book in Clinical Psychology. 62-78.

[3] 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.

[4] Bishop, C. M. (1995). Training with noise is equivalent to Tikhonov regularization. Neural computation, 7(1), 108-116.

[5] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

[6] Sukhbaatar, S., Weston, J., & Fergus, R. (2015). End-to-end memory networks. In Advances in neural information processing systems (pp. 2440-2448).

[7] Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. arXiv preprint arXiv:1511.06279.

[8] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ⋯& Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680).

[9] Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. arXiv preprint.

[10] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196.

[11] Li, M. (2017). Scaling Distributed Machine Learning with System and Algorithm Co-design (Doctoral dissertation, PhD thesis, Intel).

[12] You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. ArXiv e-prints.

[13] Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ⋯& Chen, T. (2018). Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. arXiv preprint arXiv:1807.11205.

[14] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., ⋯& Zweig, G. (2017, March). The Microsoft 2016 conversational speech recognition system. In Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on (pp. 5255-5259). IEEE.

[15] Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ⋯& Huang, T. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. Large scale visual recognition challenge.

[16] Hu, J., Shen, L., & Sun, G. (2017). Squeeze-and-excitation networks. arXiv preprint arXiv:1709.01507, 7.

[17] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. Artificial intelligence, 134 (1-2), 57-83.

[18] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ⋯& Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature,

529 (7587), 484.

[19] Brown, N., & Sandholm, T. (2017, August). Libratus: The superhuman ai for no-limit poker. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence.

[20] Canny, J. (1986). A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence, (6), 679-698.

[21] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. International journal of computer vision, 60(2), 91-110.

[22] Salton, G., & McGill, M. J. (1986). Introduction to modern information retrieval.

[23] Tesauro, G. (1995), Transactions of the ACM, (38) 3, 58-68

4.10 議論のためのQRコードをスキャン

5

事前準備

深層学習を始めるには、いくつかの基本的なスキルを身につける必要があります。すべての機械学習はデータから情報を抽出することに関係しています。そこで、データの保存・操作・前処理に関するスキルを学ぶことから始めます。

さらに機械学習では通常、大きなデータセットを扱う必要があります。データセットは、行がデータ例に対応し、列が属性に対応する表と考えることができます。線形代数は、表形式のデータを扱うための強力な手法を提供します。細かい部分にあまり深く入り込まずに、むしろ行列演算の基本とそれらの実装に着目します。

そして、深層学習は最適化そのものです。いくつかのパラメータをもつモデルに対して、データに最も良く合うものを見つけたいのです。アルゴリズムの各ステップで、パラメータをどのように動かせばよいか決めるためには、少し微積分が必要です。幸いなことに、`autograd`パッケージが自動で微分を計算することができるので、それについて後に説明します。

次に、機械学習は予測を行うことと関係しています。つまり、観測した情報にもとづいて、未知のいくつかの属性に関する尤もらしい値を考えることです。不確実性のもとで、厳密な推論を行うためには、確率に関する用語を用いる必要があるでしょう。

最後に、公式のドキュメントはこの書籍以上に豊富な説明と例を提供しています。この章の締めくくる際は、必要な情報を探すためのドキュメントの調べ方をお伝えします。

この本は深層学習に関する適切な理解を得るために、数学的な内容は最小限にとどめています。だからといって、この本が数学と無関係であるということを言っているわけではありません。この章では、だれもがこの書籍の数学的内容のほとんどを最低限理解できるように、基礎

とよく使われる数学について駆け足で紹介します。もし、数学の内容をすべて理解したい場合は、:numref:chap_appendix_math をさらに読むと十分でしょう。

```
:maxdepth: 2

ndarray
pandas
linear-algebra
calculus
autograd
probability
lookup-api
```

6

Deep Learning Basics

This chapter introduces the basics of deep learning. This includes network architectures, data, loss functions, optimization, and capacity control. In order to make things easier to grasp, we begin with very simple concepts, such as linear functions, linear regression, and stochastic gradient descent. This forms the basis for slightly more complex techniques such as the softmax (statisticians refer to it as multinomial regression) and multilayer perceptrons. At this point we are already able to design fairly powerful networks, albeit not with the requisite control and finesse. For this we need to understand the notion of capacity control, overfitting and underfitting. Regularization techniques such as dropout, numerical stability and initialization round out the presentation. Throughout, we focus on applying the models to real data, such as to give the reader a firm grasp not just of the concepts but also of the practice of using deep networks. Issues of performance, scalability and efficiency are relegated to the next chapters.

6.1 Linear Regression

To get our feet wet, we'll start off by looking at the problem of regression. This is the task of predicting a *real valued target* y given a data point x. Regression problems are extremely common in practice. For example, they are used for predicting continuous values, such as house prices, temperatures, sales, and so on. This is quite different from classification problems (which we study later), where the outputs are discrete (such as apple, banana, orange, etc. in image classification).

6.1.1 Basic Elements of Linear Regression

In linear regression, the simplest and still perhaps the most useful approach, we assume that prediction can be expressed as a *linear* combination of the input features (thus giving the name *linear* regression).

Linear Model

For the sake of simplicity we will use the problem of estimating the price of a house (e.g. in dollars) based on area (e.g. in square feet) and age (e.g. in years) as our running example. In this case we could model

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b$$

While this is quite illustrative, it becomes extremely tedious when dealing with more than two variables (even just naming them becomes a pain). This is what mathematicians have invented vectors for. In the case of d variables we get

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

Given a collection of data points X, and corresponding target values $\mathbf{y}$, we'll try to find the *weight* vector w and bias term b (also called an *offset* or *intercept*) that approximately associate data points x_i with their corresponding labels y_i. Using slightly more advanced math notation, we can express the long sum as $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. Finally, for a collection of data points $\mathbf{X}$ the predictions $\hat{\mathbf{y}}$ can be expressed via the matrix-vector product:

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

It's quite reasonable to assume that the relationship between x and y is only approximately linear. There might be some error in measuring things. Likewise, while the price of a house typically decreases, this is probably less the case with very old historical mansions which are likely to be prized specifically for their age. To find the parameters w we need two more things: some way to measure the quality of the current model and secondly, some way to manipulate the model to improve its quality.

Training Data

The first thing that we need is data, such as the actual selling price of multiple houses as well as their corresponding area and age. We hope to find model parameters on this data to minimize the error between the predicted price and the real price of the model. In the terminology of machine learning, the data set is called a 'training data' or 'training set', a house (often a house and its price) is called a 'sample', and its actual selling price is called a 'label'. The two factors used to predict the label are called 'features' or 'covariates'. Features are used to describe the characteristics of the sample.

Typically we denote by n the number of samples that we collect. Each sample (indexed as i) is described by $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$, and the label is $y^{(i)}$.

Loss Function

In model training, we need to measure the error between the predicted value and the real value of the price. Usually, we will choose a non-negative number as the error. The smaller the value, the smaller the error. A common choice is the square function. The expression for evaluating the error of a sample with an index of i is as follows:

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

The constant $1/2$ ensures that the constant coefficient, after deriving the quadratic term, is 1, which is slightly simpler in form. Obviously, the smaller the error, the closer the predicted price is to the actual price, and when the two are equal, the error will be zero. Given the training data set, this error is only related to the model parameters, so we record it as a function with the model parameters as parameters. In machine learning, we call the function that measures the error the 'loss function'. The squared error function used here is also referred to as 'square loss'.

To make things a bit more concrete, consider the example below where we plot such a regression problem for a one-dimensional case, e.g. for a model where house prices depend only on area.

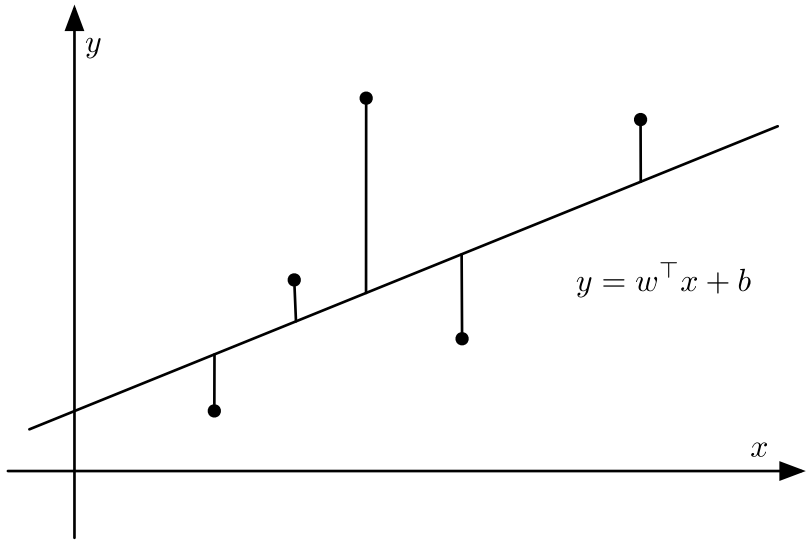


Fig. 6.1: Linear regression is a single-layer neural network.

As you can see, large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger

contributions in terms of the loss, due to the quadratic dependence. To measure the quality of a model on the entire dataset, we can simply average the losses on the training set.

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

In model training, we want to find a set of model parameters, represented by $\mathbf{w}^*, b^*$, that can minimize the average loss of training samples:

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

Optimization Algorithm

When the model and loss function are in a relatively simple format, the solution to the aforementioned loss minimization problem can be expressed analytically in a closed form solution, involving matrix inversion. This is very elegant, it allows for a lot of nice mathematical analysis, *but* it is also very restrictive insofar as this approach only works for a small number of cases (e.g. multilayer perceptrons and nonlinear layers are no go). Most deep learning models do not possess such analytical solutions. The value of the loss function can only be reduced by a finite update of model parameters via an incremental optimization algorithm.

The mini-batch stochastic gradient descent is widely used for deep learning to find numerical solutions. Its algorithm is simple: first, we initialize the values of the model parameters, typically at random; then we iterate over the data multiple times, so that each iteration may reduce the value of the loss function. In each iteration, we first randomly and uniformly sample a mini-batch $\mathcal{B}$ consisting of a fixed number of training data examples; we then compute the derivative (gradient) of the average loss on the mini batch with regard to the model parameters. Finally, the product of this result and a predetermined step size $\eta > 0$ is used to change the parameters in the direction of the minimum of the loss. In math we have

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

For quadratic losses and linear functions we can write this out explicitly as follows. Note that $\mathbf{w}$ and $\mathbf{x}$ are vectors. Here the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots w_d$.

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

In the above equation $|\mathcal{B}|$ represents the number of samples (batch size) in each mini-batch, η is referred to as 'learning rate' and takes a positive number. It should be emphasized that

the values of the batch size and learning rate are set somewhat manually and are typically not learned through model training. Therefore, they are referred to as *hyper-parameters*. What we usually call *tuning hyper-parameters* refers to the adjustment of these terms. In the worst case this is performed through repeated trial and error until the appropriate hyper-parameters are found. A better approach is to learn these as parts of model training. This is an advanced topic and we do not cover them here for the sake of simplicity.

Model Prediction

After model training has been completed, we then record the values of the model parameters $\mathbf{w}, b$ as $\hat{\mathbf{w}}, \hat{b}$. Note that we do not necessarily obtain the optimal solution of the loss function minimizer, $\mathbf{w}^*, b^*$ (or the true parameters), but instead we gain an approximation of the optimal solution. We can then use the learned linear regression model $\hat{\mathbf{w}}^\top x + \hat{b}$ to estimate the price of any house outside the training data set with area (square feet) as x_1 and house age (year) as x_2. Here, estimation also referred to as 'model prediction' or 'model inference'.

Note that calling this step 'inference' is actually quite a misnomer, albeit one that has become the default in deep learning. In statistics 'inference' means estimating parameters and outcomes based on other data. This misuse of terminology in deep learning can be a source of confusion when talking to statisticians. We adopt the incorrect, but by now common, terminology of using 'inference' when a (trained) model is applied to new data (and express our sincere apologies to centuries of statisticians).

6.1.2 From Linear Regression to Deep Networks

So far we only talked about linear functions. Neural Networks cover a lot more than that. That said, linear functions are an important building block. Let's start by rewriting things in a 'layer' notation.

Neural Network Diagram

While in deep learning, we can represent model structures visually using neural network diagrams. To more clearly demonstrate the linear regression as the structure of neural network, Figure 3.1 uses a neural network diagram to represent the linear regression model presented in this section. The neural network diagram hides the weight and bias of the model parameter.

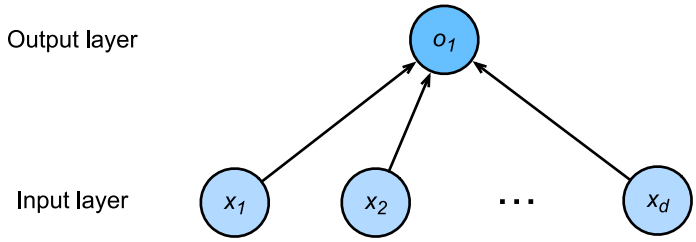


Fig. 6.2: Linear regression is a single-layer neural network.

In the neural network shown above, the inputs are $x_1, x_2, \ldots x_d$. Sometimes the number of inputs is also referred as feature dimension. In the above cases the number of inputs is d and the number of outputs is 1. It should be noted that we use the output directly as the output of linear regression. Since the input layer does not involve any other nonlinearities or any further calculations, the number of layers is 1. Sometimes this setting is also referred to as a single neuron. Since all inputs are connected to all outputs (in this case it's just one), the layer is also referred to as a 'fully connected layer' or 'dense layer'.

A Detour to Biology

Neural networks quite clearly derive their name from Neuroscience. To understand a bit better how many network architectures were invented, it is worth while considering the basic structure of a neuron. For the purpose of the analogy it is sufficient to consider the *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals) which connect to other neurons via *synapses*.

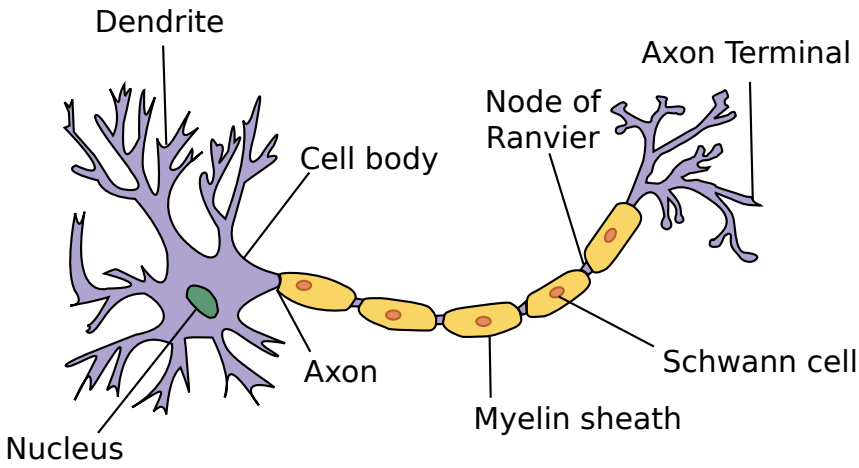


Fig. 6.3: The real neuron

Information x_i arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i which determine how to respond to the inputs (e.g. activation or inhibition via $x_i w_i$). All this is aggregated in the nucleus $y = \sum_i x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$. From there it either reaches its destination (e.g. a muscle) or is fed into another neuron via its dendrites.

Brain *structures* can be quite varied. Some look rather arbitrary whereas others have a very regular structure. E.g. the visual system of many insects is quite regular. The analysis of such structures has often inspired neuroscientists to propose new architectures, and in some cases, this has been successful. Note, though, that it would be a fallacy to require a direct correspondence - just like airplanes are *inspired* by birds, they have many distinctions. Equal sources of inspiration were mathematics and computer science.

Vectorization for Speed

In model training or prediction, we often use vector calculations and process multiple observations at the same time. To illustrate why this matters, consider two methods of adding vectors. We begin by creating two 1000 dimensional ones first.

```
In [1]: from mxnet import nd
        from time import time

        a = nd.ones(shape=10000)
        b = nd.ones(shape=10000)
```

One way to add vectors is to add them one coordinate at a time using a for loop.

```
In [2]: start = time()
        c = nd.zeros(shape=10000)
        for i in range(10000):
            c[i] = a[i] + b[i]
        time() - start
```

```
Out[2]: 1.4604191780090332
```

Another way to add vectors is to add the vectors directly:

```
In [3]: start = time()
        d = a + b
        time() - start
```

```
Out[3]: 0.00021910667419433594
```

Obviously, the latter is vastly faster than the former. Vectorizing code is a good way of getting order of magnitude speedups. Likewise, as we saw above, it also greatly simplifies the mathematics and with it, it reduces the potential for errors in the notation.

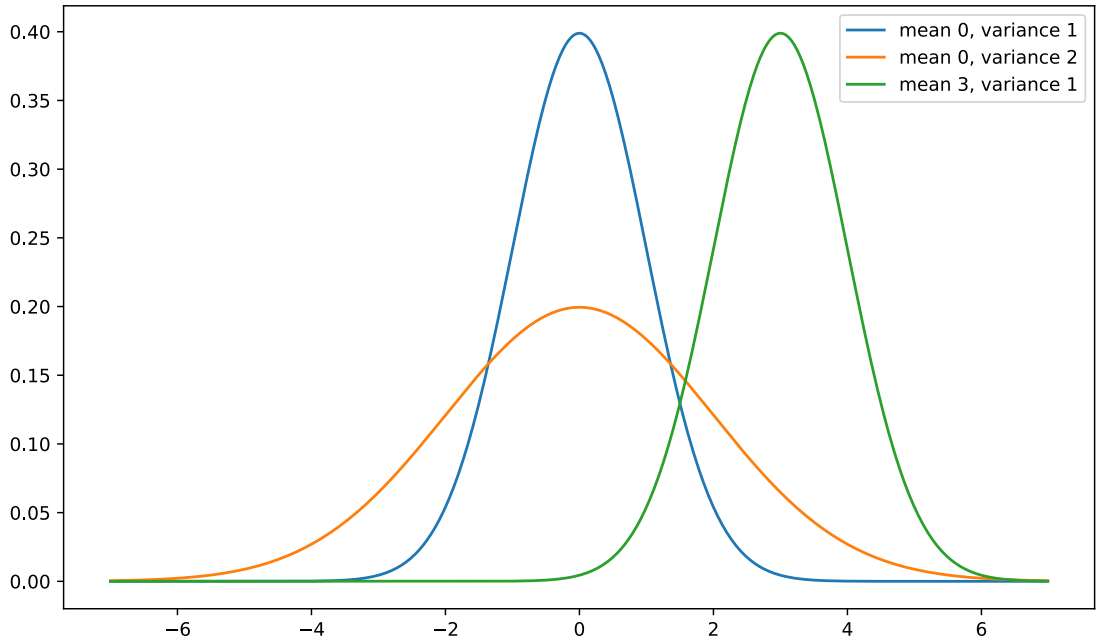
6.1.3 The Normal Distribution and Squared Loss

The following is optional and can be skipped but it will greatly help with understanding some of the design choices in building deep learning models. As we saw above, using the squared loss $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ has many nice properties, such as having a particularly simple derivative $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$. That is, the gradient is given by the difference between estimate and observation. You might reasonably point out that linear regression is a classical statistical model. Legendre first developed the method of least squares regression in 1805, which was shortly thereafter rediscovered by Gauss in 1809. To understand this a bit better, recall the normal distribution with mean μ and variance σ^2.

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

It can be visualized as follows:

```
In [4]: %matplotlib inline
        from matplotlib import pyplot as plt
        from IPython import display
        from mxnet import nd
        import math

        x = nd.arange(-7, 7, 0.01)
        # Mean and variance pairs
        parameters = [(0,1), (0,2), (3,1)]

        # Display SVG rather than JPG
        display.set_matplotlib_formats('svg')
        plt.figure(figsize=(10, 6))
        for (mu, sigma) in parameters:
            p = (1/math.sqrt(2 * math.pi * sigma**2)) * nd.exp(-(0.5/sigma**2) *
 ↪  (x-mu)**2)
            plt.plot(x.asnumpy(), p.asnumpy(), label='mean ' + str(mu) + ', variance '
 ↪  + str(sigma))

        plt.legend()
        plt.show()
```



As can be seen in the figure above, changing the mean shifts the function, increasing the variance makes it more spread-out with a lower peak. The key assumption in linear regression with least mean squares loss is that the observations actually arise from noisy observations, where noise is added to the data, e.g. as part of the observations process.

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

This allows us to write out the *likelihood* of seeing a particular y for a given $\mathbf{x}$ via

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right)$$

A good way of finding the most likely values of b and $\mathbf{w}$ is to maximize the *likelihood* of the entire dataset

$$p(Y|X) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)})$$

The notion of maximizing the likelihood of the data subject to the parameters is well known as the *Maximum Likelihood Principle* and its estimators are usually called *Maximum Likelihood Estimators* (MLE). Unfortunately, maximizing the product of many exponential functions is pretty awkward, both in terms of implementation and in terms of writing it out on paper. Instead, a much better way is to minimize the *Negative Log-Likelihood* $-\log P(Y|X)$. In the above case this

works out to be

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

A closer inspection reveals that for the purpose of minimizing $-\log P(Y|X)$ we can skip the first term since it doesn't depend on $\mathbf{w}, b$ or even the data. The second term is identical to the objective we initially introduced, but for the multiplicative constant $\frac{1}{\sigma^2}$. Again, this can be skipped if we just want to get the most likely solution. It follows that maximum likelihood in a linear model with additive Gaussian noise is equivalent to linear regression with squared loss.

6.1.4 Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.
- Vectorizing makes everything better (mostly math) and faster (mostly code).
- Minimizing an objective function and performing maximum likelihood can mean the same thing.
- Linear models are neural networks, too.

6.1.5 Problems

1. Assume that we have some data $x_1, \ldots x_n \in \mathbb{R}$. Our goal is to find a constant b such that $\sum_i (x_i - b)^2$ is minimized.
   - Find the optimal closed form solution.
   - What does this mean in terms of the Normal distribution?
2. Assume that we want to solve the optimization problem for linear regression with quadratic loss explicitly in closed form. To keep things simple, you can omit the bias b from the problem.
   - Rewrite the problem in matrix and vector notation (hint - treat all the data as a single matrix).
   - Compute the gradient of the optimization problem with respect to w.
   - Find the closed form solution by solving a matrix equation.
   - When might this be better than using stochastic gradient descent (i.e. the incremental optimization approach that we discussed above)? When will this break (hint - what happens for high-dimensional x, what if many observations are very similar)?.
3. Assume that the noise model governing the additive noise ϵ is the exponential distribution. That is, $p(\epsilon) = \frac{1}{2}\exp(-|\epsilon|)$.

- Write out the negative log-likelihood of the data under the model $-\log p(Y|X)$.
- Can you find a closed form solution?
- Suggest a stochastic gradient descent algorithm to solve this problem. What could possibly go wrong (hint - what happens near the stationary point as we keep on updating the parameters). Can you fix this?

4. Compare the runtime of the two methods of adding two vectors using other packages (such as NumPy) or other programming languages (such as MATLAB).

6.1.6 Scan the QR Code to Discuss

6.2 Linear Regression Implementation from Scratch

After getting some background on linear regression, we are now ready for a hands-on implementation. While a powerful deep learning framework minimizes repetitive work, relying on it too much to make things easy can make it hard to properly understand how deep learning works. This matters in particular if we want to change things later, e.g. define our own layers, loss functions, etc. Because of this, we start by describing how to implement linear regression training using only NDArray and `autograd`.

Before we begin, let's import the package or module required for this section's experiment; `matplotlib` will be used for plotting and will be set to embed in the GUI.

```
In [1]: %matplotlib inline
        from IPython import display
        from matplotlib import pyplot as plt
        from mxnet import autograd, nd
        import random
```

6.2.1 Generating Data Sets

By constructing a simple artificial training data set, we can visually compare the differences between the parameters we have learned and the actual model parameters. Set the number of examples in the training data set as 1000 and the number of inputs (feature number) as 2. Using the randomly generated batch example feature $\mathbf{X} \in \mathbb{R}^{1000\times 2}$, we use the actual weight

$\mathbf{w} = [2, -3.4]^\top$ and bias $b = 4.2$ of the linear regression model, as well as a random noise item ϵ to generate the tag

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

The noise term ϵ (or rather each coordinate of it) obeys a normal distribution with a mean of 0 and a standard deviation of 0.01. To get a better idea, let us generate the dataset.

```
In [2]: num_inputs = 2
        num_examples = 1000
        true_w = nd.array([2, -3.4])
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = nd.dot(features, true_w) + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

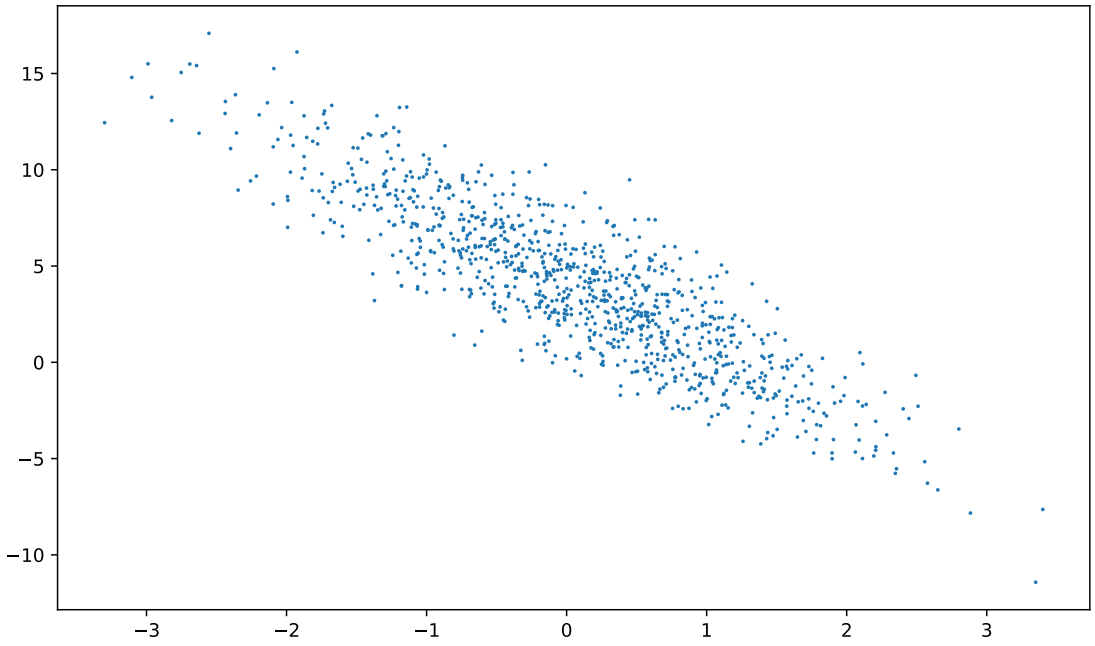
Note that each row in `features` consists of a 2-dimensional data point and that each row in `labels` consists of a 1-dimensional target value (a scalar).

```
In [3]: features[0], labels[0]
```

```
Out[3]: (
         [2.2122064 0.7740038]
         <NDArray 2 @cpu(0)>,
         [6.000587]
         <NDArray 1 @cpu(0)>)
```

By generating a scatter plot using the second `features[:, 1]` and `labels`, we can clearly observe the linear correlation between the two.

```
In [4]: def use_svg_display():
            # Display in vector graphics
            display.set_matplotlib_formats('svg')

        def set_figsize(figsize=(3.5, 2.5)):
            use_svg_display()
            # Set the size of the graph to be plotted
            plt.rcParams['figure.figsize'] = figsize

        set_figsize()
        plt.figure(figsize=(10, 6))
        plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```



The plotting function `plt` as well as the `use_svg_display` and `set_figsize` functions are defined in the `d2l` package. We will call `d2l.plt` directly for future plotting. To print the vector diagram and set its size, we only need to call `d2l.set_figsize()` before plotting, because `plt` is a global variable in the `d2l` package.

6.2.2 Reading Data

We need to iterate over the entire data set and continuously examine mini-batches of data examples when training the model. Here we define a function. Its purpose is to return the features and tags of random `batch_size` (batch size) examples every time it's called. One might wonder why we are not reading one observation at a time but rather construct an iterator which returns a few observations at a time. This has mostly to do with efficiency when optimizing. Recall that when we processed one dimension at a time the algorithm was quite slow. The same thing happens when processing single observations vs. an entire 'batch' of them, which can be represented as a matrix rather than just a vector. In particular, GPUs are much faster when it comes to dealing with matrices, up to an order of magnitude. This is one of the reasons why deep learning usually operates on mini-batches rather than singletons.

```
In [5]: # This function has been saved in the d2l package for future use
        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):
            j = nd.array(indices[i: min(i + batch_size, num_examples)])
            yield features.take(j), labels.take(j)
            # The “take” function will then return the corresponding element based
            # on the indices
```

Let's read and print the first small batch of data examples. The shape of the features in each batch corresponds to the batch size and the number of input dimensions. Likewise, we obtain as many labels as requested by the batch size.

```
In [6]: batch_size = 10

        for X, y in data_iter(batch_size, features, labels):
            print(X, y)
            break
```

```
[[ 0.23113538  1.0828242 ]
 [ 0.32212737 -1.5355504 ]
 [-1.7401326   1.0360596 ]
 [-0.61431247 -0.069661  ]
 [-0.30014795  1.0723637 ]
 [ 0.7508367   0.64649147]
 [-1.2878888   1.5034332 ]
 [ 0.09329322  1.9133487 ]
 [ 0.5510752   1.6094043 ]
 [-0.92666906  0.33800116]]
<NDArray 10x2 @cpu(0)>
[ 0.95886403 10.066744   -2.8146596   3.2289903  -0.02559855  3.5131779
 -3.4834092  -2.1157448  -0.16637745  1.1790081 ]
<NDArray 10 @cpu(0)>
```

Clearly, if we run the iterator again, we obtain a different minibatch until all the data has been exhausted (try this). Note that the iterator described above is a bit inefficient (it requires that we load all data in memory and that we perform a lot of random memory access). The built-in iterators are more efficient and they can deal with data stored on file (or being fed via a data stream).

6.2.3 Initialize Model Parameters

Weights are initialized to normal random numbers using a mean of 0 and a standard deviation of 0.01, with the bias b set to zero.

```
In [7]: w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
        b = nd.zeros(shape=(1,))
```

In the succeeding cells, we're going to update these parameters to better fit our data. This will involve taking the gradient (a multi-dimensional derivative) of some loss function with respect to the parameters. We'll update each parameter in the direction that reduces the loss. In order for `autograd` to know that it needs to set up the appropriate data structures, track changes, etc., we need to attach gradients explicitly.

```
In [8]: w.attach_grad()
        b.attach_grad()
```

6.2.4 Define the Model

Next we'll want to define our model. In this case, we'll be working with linear models, the simplest possible useful neural network. To calculate the output of the linear model, we simply multiply a given input with the model's weights w, and add the offset b.

```
In [9]: # This function has been saved in the d2l package for future use
        def linreg(X, w, b):
            return nd.dot(X, w) + b
```

6.2.5 Define the Loss Function

We will use the squared loss function described in the previous section to define the linear regression loss. In the implementation, we need to transform the true value `y` into the predicted value's shape `y_hat`. The result returned by the following function will also be the same as the `y_hat` shape.

```
In [10]: # This function has been saved in the d2l package for future use
         def squared_loss(y_hat, y):
             return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

6.2.6 Define the Optimization Algorithm

Linear regression actually has a closed-form solution. However, most interesting models that we'll care about cannot be solved analytically. So we'll solve this problem by stochastic gradient descent `sgd`. At each step, we'll estimate the gradient of the loss with respect to our weights, using one batch randomly drawn from our dataset. Then, we'll update our parameters a small amount in the direction that reduces the loss. Here, the gradient calculated by the automatic differentiation module is the gradient sum of a batch of examples. We divide it by the batch size to obtain the average. The size of the step is determined by the learning rate `lr`.

```
In [11]: # This function has been saved in the d2l package for future use
         def sgd(params, lr, batch_size):
             for param in params:
                 param[:] = param - lr * param.grad / batch_size
```

6.2.7 Training

In training, we will iterate over the data to improve the model parameters. In each iteration, the mini-batch stochastic gradient is calculated by first calling the inverse function `backward` depending on the currently read mini-batch data examples (feature `X` and label `y`), and then

calling the optimization algorithm `sgd` to iterate the model parameters. Since we previously set the batch size `batch_size` to 10, the loss shape `l` for each small batch is (10, 1).

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Since nobody wants to compute gradients explicitly (this is tedious and error prone) we use automatic differentiation to compute g. See section "Automatic Gradient" for more details. Since the loss `l` is not a scalar variable, running `l.backward()` will add together the elements in `l` to obtain the new variable, and then calculate the variable model parameters' gradient.

In an epoch (a pass through the data), we will iterate through the `data_iter` function once and use it for all the examples in the training data set (assuming the number of examples is divisible by the batch size). The number of epochs `num_epochs` and the learning rate `lr` are both hyper-parameters and are set to 3 and 0.03, respectively. Unfortunately in practice, the majority of the hyper-parameters will require some adjustment by trial and error. For instance, the model might actually become more accurate by training longer (but this increases computational cost). Likewise, we might want to change the learning rate on the fly. We will discuss this later in the chapter on "Optimization Algorithms".

```
In [12]: lr = 0.03  # Learning rate
         num_epochs = 3  # Number of iterations
         net = linreg  # Our fancy linear model
         loss = squared_loss  # 0.5 (y-y')^2

         for epoch in range(num_epochs):
             # Assuming the number of examples can be divided by the batch size, all
             # the examples in the training data set are used once in one epoch
             # iteration. The features and tags of mini-batch examples are given by X
             # and y respectively
             for X, y in data_iter(batch_size, features, labels):
                 with autograd.record():
                     l = loss(net(X, w, b), y)  # Minibatch loss in X and y
                 l.backward()  # Compute gradient on l with respect to [w,b]
                 sgd([w, b], lr, batch_size)  # Update parameters using their gradient
             train_l = loss(net(features, w, b), labels)
             print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

```
epoch 1, loss 0.040566
epoch 2, loss 0.000151
epoch 3, loss 0.000051
```

To evaluate the trained model, we can compare the actual parameters used with the parameters we have learned after the training has been completed. They are very close to each other.

```
In [13]: print('Error in estimating w', true_w - w.reshape(true_w.shape))
         print('Error in estimating b', true_b - b)
```

```
Error in estimating w
[ 0.00028121 -0.00040436]
```

```
<NDArray 2 @cpu(0)>
Error in estimating b
[0.00080729]
<NDArray 1 @cpu(0)>
```

Note that we should not take it for granted that we are able to reover the parameters accurately. This only happens for a special category problems: strongly convex optimization problems with 'enough' data to ensure that the noisy samples allow us to recover the underlying dependency correctly. In most cases this is *not* the case. In fact, the parameters of a deep network are rarely the same (or even close) between two different runs, unless everything is kept identically, including the order in which the data is traversed. Nonetheless this can lead to very good solutions, mostly due to the fact that quite often there are many sets of parameters that work well.

6.2.8 Summary

We saw how a deep network can be implemented and optimized from scratch, using just NDArray and `autograd` without any need for defining layers, fancy optimizers, etc. This only scratches the surface of what is possible. In the following sections, we will describe additional deep learning models based on what we have just learned and you will learn how to implement them using more concisely.

6.2.9 Problems

1. What would happen if we were to initialize the weights $\mathbf{w} = 0$. Would the algorithm still work?
2. Assume that you're Georg Simon Ohm trying to come up with a model between voltage and current. Can you use `autograd` to learn the parameters of your model.
3. Can you use Planck's Law to determine the temperature of an object using spectral energy density.
4. What are the problems you might encounter if you wanted to extend `autograd` to second derivatives? How would you fix them?
5. Why is the `reshape` function needed in the `squared_loss` function?
6. Experiment using different learning rates to find out how fast the loss function value drops.
7. If the number of examples cannot be divided by the batch size, what happens to the `data_iter` function's behavior?

6.2.10 Scan the QR Code to Discuss

6.3 Concise Implementation of Linear Regression

With the development of deep learning frameworks, it has become increasingly easy to develop deep learning applications. In practice, we can usually implement the same model, but much more concisely how we introduce it in the previous section. In this section, we will introduce how to use the Gluon interface provided by MXNet.

6.3.1 Generating Data Sets

We will generate the same data set as that used in the previous section.

```
In [1]: from mxnet import autograd, nd

        num_inputs = 2
        num_examples = 1000
        true_w = nd.array([2, -3.4])
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = nd.dot(features, true_w) + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

6.3.2 Reading Data

Gluon provides the `data` module to read data. Since `data` is often used as a variable name, we will replace it with the pseudonym `gdata` (adding the first letter of Gluon) when referring to the imported `data` module. In each iteration, we will randomly read a mini-batch containing 10 data instances.

```
In [2]: from mxnet.gluon import data as gdata

        batch_size = 10
        # Combine the features and labels of the training data
        dataset = gdata.ArrayDataset(features, labels)
        # Randomly reading mini-batches
        data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
```

The use of `data_iter` here is the same as in the previous section. Now, we can read and print the first mini-batch of instances.

```
In [3]: for X, y in data_iter:
            print(X, y)
            break
```

```
[[-0.10079116  0.48418596]
 [-0.4844843   2.3336477 ]
 [ 1.3684256  -0.74242383]
 [ 0.20988831 -1.0934474 ]
 [ 1.7879504  -0.20021723]
 [-1.1318913   0.33254048]
 [-0.81052285 -1.1052948 ]
 [ 0.9676651   0.5122743 ]
 [-1.8979539  -0.15962839]
 [-0.64271027  0.58239543]]
<NDArray 10x2 @cpu(0)>
[ 2.3647711  -4.7084603   9.457492    8.336128    8.462152    0.79540896
  6.322081    4.405501    0.9294544   0.919952  ]
<NDArray 10 @cpu(0)>
```

6.3.3 Define the Model

When we implemented the linear regression model from scratch in the previous section, we needed to define the model parameters and use them to describe step by step how the model is evaluated. This can become complicated as we build complex models. Gluon provides a large number of predefined layers, which allow us to focus especially on the layers used to construct the model rather than having to focus on the implementation.

To define a linear model, first import the module `nn`. `nn` is an abbreviation for neural networks. As the name implies, this module defines a large number of neural network layers. We will first define a model variable `net`, which is a `Sequential` instance. In Gluon, a `Sequential` instance can be regarded as a container that concatenates the various layers in sequence. When constructing the model, we will add the layers in their order of occurrence in the container. When input data is given, each layer in the container will be calculated in order, and the output of one layer will be the input of the next layer.

```
In [4]: from mxnet.gluon import nn
        net = nn.Sequential()
```

Recall the architecture of a single layer network. The layer is fully connected since it connects all inputs with all outputs by means of a matrix-vector multiplication. In Gluon, the fully connected layer is referred to as a `Dense` instance. Since we only want to generate a single scalar output, we set that number to 1.

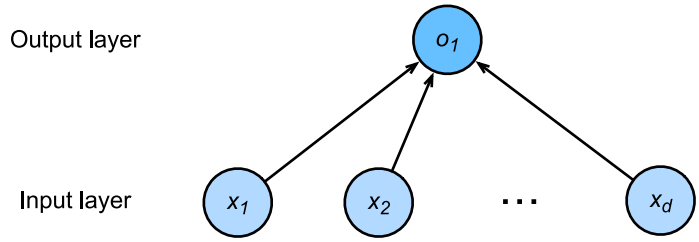


Fig. 6.4: Linear regression is a single-layer neural network.

```
In [5]: net.add(nn.Dense(1))
```

It is worth noting that, in Gluon, we do not need to specify the input shape for each layer, such as the number of linear regression inputs. When the model sees the data, for example, when the `net(X)` is executed later, the model will automatically infer the number of inputs in each layer. We will describe this mechanism in detail in the chapter "Deep Learning Computation". Gluon introduces this design to make model development more convenient.

6.3.4 Initialize Model Parameters

Before using `net`, we need to initialize the model parameters, such as the weights and biases in the linear regression model. We will import the `initializer` module from MXNet. This module provides various methods for model parameter initialization. The `init` here is the abbreviation of `initializer`. By`init.Normal(sigma=0.01)` we specify that each weight parameter element is to be randomly sampled at initialization with a normal distribution with a mean of 0 and standard deviation of 0.01. The bias parameter will be initialized to zero by default.

```
In [6]: from mxnet import init
        net.initialize(init.Normal(sigma=0.01))
```

The code above looks pretty straightforward but in reality something quite strange is happening here. We are initializing parameters for a networks where we haven't told Gluon yet how many dimensions the input will have. It might be 2 as in our example or 2,000, so we couldn't just preallocate enough space to make it work. What happens behind the scenes is that the updates are deferred until the first time that data is sent through the networks. In doing so, we prime all settings (and the user doesn't even need to worry about it). The only cautionary notice is that since the parameters have not been initialized yet, we would not be able to manipulate them yet.

6.3.5 Define the Loss Function

In Gluon, the module `loss` defines various loss functions. We will replace the imported module `loss` with the pseudonym `gloss`, and directly use the squared loss it provides as a loss function for the model.

```
In [7]: from mxnet.gluon import loss as gloss
        loss = gloss.L2Loss()  # The squared loss is also known as the L2 norm loss
```

6.3.6 Define the Optimization Algorithm

Again, we do not need to implement mini-batch stochastic gradient descent. After importing Gluon, we now create a `Trainer` instance and specify a mini-batch stochastic gradient descent with a learning rate of 0.03 (`sgd`) as the optimization algorithm. This optimization algorithm will be used to iterate through all the parameters contained in the `net` instance's nested layers through the `add` function. These parameters can be obtained by the `collect_params` function.

```
In [8]: from mxnet import gluon
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

6.3.7 Training

You might have noticed that it was a bit more concise to express our model in Gluon. For example, we didn't have to individually allocate parameters, define our loss function, or implement stochastic gradient descent. The benefits of relying on Gluon's abstractions will grow substantially once we start working with much more complex models. But once we have all the basic pieces in place, the training loop itself is quite similar to what we would do if implementing everything from scratch.

To refresh your memory. For some number of epochs, we'll make a complete pass over the dataset (train_data), grabbing one mini-batch of inputs and the corresponding ground-truth labels at a time. Then, for each batch, we'll go through the following ritual.

- Generate predictions `net(X)` and the loss `l` by executing a forward pass through the network.
- Calculate gradients by making a backwards pass through the network via `l.backward()`.
- Update the model parameters by invoking our SGD optimizer (note that we need not tell trainer.step about which parameters but rather just the amount of data, since we already performed that in the initialization of trainer).

For good measure we compute the loss on the features after each epoch and print it to monitor progress.

```
In [9]: num_epochs = 3
        for epoch in range(1, num_epochs + 1):
            for X, y in data_iter:
                with autograd.record():
                    l = loss(net(X), y)
                l.backward()
                trainer.step(batch_size)
            l = loss(net(features), labels)
            print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
```

```
epoch 1, loss: 0.040543
epoch 2, loss: 0.000149
epoch 3, loss: 0.000051
```

The model parameters we have learned and the actual model parameters are compared as below. We get the layer we need from the `net` and access its weight (`weight`) and bias (`bias`). The parameters we have learned and the actual parameters are very close.

```
In [10]: w = net[0].weight.data()
         print('Error in estimating w', true_w.reshape(w.shape) - w)
         b = net[0].bias.data()
         print('Error in estimating b', true_b - b)
```

```
Error in estimating w
[[ 0.00010455 -0.00061774]]
<NDArray 1x2 @cpu(0)>
Error in estimating b
[0.00046301]
<NDArray 1 @cpu(0)>
```

6.3.8 Summary

- Using Gluon, we can implement the model more succinctly.
- In Gluon, the module `data` provides tools for data processing, the module `nn` defines a large number of neural network layers, and the module `loss` defines various loss functions.
- MXNet's module `initializer` provides various methods for model parameter initialization.
- Dimensionality and storage are automagically inferred (but caution if you want to access parameters before they've been initialized).

6.3.9 Problems

1. If we replace `l = loss(output, y)` with `l = loss(output, y).mean()`, we need to change `trainer.step(batch_size)` to `trainer.step(1)` accordingly. Why?
2. Review the MXNet documentation to see what loss functions and initialization methods are provided in the modules `gluon.loss` and `init`. Replace the loss by Huber's loss.
3. How do you access the gradient of `dense.weight`?

6.3.10 Scan the QR Code to Discuss

6.4 Softmax Regression

Over the last two sections we worked through how to implement a linear regression model, both *from scratch* and *using Gluon* to automate most of the repetitive work like allocating and initializing parameters, defining loss functions, and implementing optimizers.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (the *price*) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you're probably looking for a regression model.

In reality, we're more often interested in making categorical assignments.

- Does this email belong in the spam folder or the inbox*?
- How likely is this customer to sign up for subscription service?*
- What is the object in the image (donkey, dog, cat, rooster, etc.)?
- Which object is a customer most likely to purchase?

When we're interested in either assigning datapoints to categories or assessing the *probability* that a category applies, we call this task *classification*. The issue with the models that we studied so far is that they cannot be applied to problems of probability estimation.

6.4.1 Classification Problems

Let's start with an admittedly somewhat contrived image problem where the input image has a height and width of 2 pixels and the color is grayscale. Thus, each pixel value can be represented by a scalar. We record the four pixels in the image as x_1, x_2, x_3, x_4. We assume that the actual labels of the images in the training data set are "cat", "chicken"or "dog"(assuming that the three animals can be represented by 4 pixels).

To represent these labels we have two choices. Either we set $y \in \{1, 2, 3\}$, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer. It would also lend itself rather neatly to regression, but the ordering of outcomes imposes some quite unnatural ordering. In our toy case, this would presume that cats are more

similar to chickens than to dogs, at least mathematically. It doesn't work so well in practice either, which is why statisticians invented an alternative approach: one hot encoding via

$$y \in \{(1,0,0),(0,1,0),(0,0,1)\}$$

That is, y is viewed as a three-dimensional vector, where $(1,0,0)$ corresponds to "cat", $(0,1,0)$ to "chicken"and $(0,0,1)$ to "dog".

Network Architecture

If we want to estimate multiple classes, we need multiple outputs, matching the number of categories. This is one of the main differences to regression. Because there are 4 features and 3 output animal categories, the weight contains 12 scalars (w with subscripts) and the bias contains 3 scalars (b with subscripts). We compute these three outputs, o_1, o_2, and o_3, for each input:

$$\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1, \\
o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2, \\
o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.
\end{aligned}$$

The neural network diagram below depicts the calculation above. Like linear regression, softmax regression is also a single-layer neural network. Since the calculation of each output, $o_1, o_2, and o_3$, depends on all inputs, x_1, x_2, x_3, and x_4, the output layer of the softmax regression is also a fully connected layer.

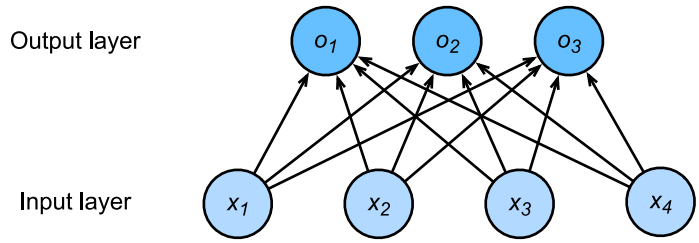


Fig. 6.5: Softmax regression is a single-layer neural network.

Softmax Operation

The chosen notation is somewhat verbose. In vector form we arrive at $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, which is much more compact to write and code. Since the classification problem requires discrete prediction output, we can use a simple approach to treat the output value o_i as the confidence level of the prediction category i. We can choose the class with the largest output value as the predicted output, which is output $\text{argmax}_i\, o_i$. For example, if o_1, o_2, and o_3 are 0.1, 10, and 0.1, respectively, then the prediction category is 2, which represents "chicken".

However, there are two problems with using the output from the output layer directly. On the one hand, because the range of output values from the output layer is uncertain, it is difficult for us to visually judge the meaning of these values. For instance, the output value 10 from the previous example indicates a level of "very confident"that the image category is "chicken". That is because its output value is 100 times that of the other two categories. However, if $o_1 = o_3 = 10^3$, then an output value of 10 means that the chance for the image category to be chicken is very low. On the other hand, since the actual label has discrete values, the error between these discrete values and the output values from an uncertain range is difficult to measure.

We could try forcing the outputs to correspond to probabilities, but there's no guarantee that on new (unseen) data the probabilities would be nonnegative, let alone sum up to 1. For this kind of discrete value prediction problem, statisticians have invented classification models such as (softmax) logistic regression. Unlike linear regression, the output of softmax regression is subjected to a nonlinearity which ensures that the sum over all outcomes always adds up to 1 and that none of the terms is ever negative. The nonlinear transformation works as follows:

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \text{ where } \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

It is easy to see $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \leq \hat{y}_i \leq 1$ for all i. Thus, $\hat{y}$ is a proper probability distribution and the values of o now assume an easily quantifiable meaning. Note that we can still find the most likely class by

$$\hat{\imath}(\mathbf{o}) = \operatorname*{argmax}_i o_i = \operatorname*{argmax}_i \hat{y}_i$$

So, the softmax operation does not change the prediction category output but rather it gives the outputs $\mathbf{o}$ proper meaning. Summarizing it all in vector notation we get $\mathbf{o}^{(i)} = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}$ where $\hat{\mathbf{y}}^{(i)} = \mathrm{softmax}(\mathbf{o}^{(i)})$.

Vectorization for Minibatches

To improve computational efficiency further, we usually carry out vector calculations for mini-batches of data. Assume that we are given a mini-batch $\mathbf{X}$ of examples with dimensionality d and batch size n. Moreover, assume that we have q categories (outputs). Then the minibatch features $\mathbf{X}$ are in $\mathbb{R}^{n \times d}$, weights $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^q$.

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$$
$$\hat{\mathbf{Y}} = \mathrm{softmax}(\mathbf{O})$$

This accelerates the dominant operation: $\mathbf{W}\mathbf{X}$ from a matrix-vector to a matrix-matrix product. The softmax itself can be computed by exponentiating all entries in $\mathbf{O}$ and then normalizing them by the sum appropriately.

6.4.2 Loss Function

Now that we have some mechanism for outputting probabilities, we need to transform this into a measure of how accurate things are, i.e. we need a *loss function*. For this we use the same concept that we already encountered in linear regression, namely likelihood maximization.

Log-Likelihood

The softmax function maps $\mathbf{o}$ into a vector of probabilities corresponding to various outcomes, such as $p(y = \text{cat}|\mathbf{x})$. This allows us to compare the estimates with reality, simply by checking how well it predicted what we observe.

$$p(Y|X) = \prod_{i=1}^{n} p(y^{(i)}|x^{(i)}) \text{ and thus } -\log p(Y|X) = \sum_{i=1}^{n} -\log p(y^{(i)}|x^{(i)})$$

Minimizing $-\log p(Y|X)$ corresponds to predicting things well. This yields the loss function (we dropped the superscript (i) to avoid notation clutter):

$$l = -\log p(y|x) = -\sum_j y_j \log \hat{y}_j$$

Here we used that by construction $\hat{y} = \text{softmax}(\mathbf{o})$ and moreover, that the vector $\mathbf{y}$ consists of all zeroes but for the correct label, such as $(1, 0, 0)$. Hence the the sum over all coordinates j vanishes for all but one term. Since all $\hat{y}_j$ are probabilities, their logarithm is never larger than 0. Consequently, the loss function is minimized if we correctly predict y with *certainty*, i.e. if $p(y|x) = 1$ for the correct label.

Softmax and Derivatives

Since the Softmax and the corresponding loss are so common, it is worth while understanding a bit better how it is computed. Plugging o into the definition of the loss l and using the definition of the softmax we obtain:

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j$$

To understand a bit better what is going on, consider the derivative with respect to o. We get

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = \Pr(y = j|x) - y_j$$

In other words, the gradient is the difference between what the model thinks should happen, as expressed by the probability $p(y|x)$, and what actually happened, as expressed by y. In this sense, it is very similar to what we saw in regression, where the gradient was the difference

between the observation y and estimate $\hat{y}$. This seems too much of a coincidence, and indeed, it isn't. In any exponential family model the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients a lot easier in practice.

Cross-Entropy Loss

Now consider the case where we don't just observe a single outcome but maybe, an entire distribution over outcomes. We can use the same representation as before for y. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l still works out fine, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_j y_j \log \hat{y}_j$$

This loss is called the cross-entropy loss. It is one of the most commonly used ones for multiclass classification. To demystify its name we need some information theory. The following section can be skipped if needed.

6.4.3 Information Theory Basics

Information theory deals with the problem of encoding, decoding, transmitting and manipulating information (aka data), preferentially in as concise form as possible.

Entropy

A key concept is how many bits of information (or randomness) are contained in data. It can be measured as the entropy of a distribution p via

$$H[p] = \sum_j -p(j) \log p(j)$$

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution p we need at least $H[p]$ 'nats' to encode it. If you wonder what a 'nat' is, it is the equivalent of bit but when using a code with base e rather than one with base 2. One nat is $\frac{1}{\log(2)} \approx 1.44$ bit. $H[p]/2$ is often also called the binary entropy.

To make this all a bit more theoretical consider the following: $p(1) = \frac{1}{2}$ whereas $p(2) = p(3) = \frac{1}{4}$. In this case we can easily design an optimal code for data drawn from this distribution, by using `0` to encode `1`, `10` for `2` and `11` for `3`. The expected number of bit is $1.5 = 0.5*1+0.25*2+0.25*2$. It is easy to check that this is the same as the binary entropy $H[p]/\log 2$.

Kullback Leibler Divergence

One way of measuring the difference between two distributions arises directly from the entropy. Since $H[p]$ is the minimum number of bits that we need to encode data drawn from p, we could ask how well it is encoded if we pick the 'wrong' distribution q. The amount of extra bits that we need to encode q gives us some idea of how different these two distributions are. Let us compute this directly - recall that to encode j using an optimal code for q would cost $-\log q(j)$ nats, and we need to use this in $p(j)$ of all cases. Hence we have

$$D(p\|q) = -\sum_j p(j) \log q(j) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)}$$

Note that minimizing $D(p\|q)$ with respect to q is equivalent to minimizing the cross-entropy loss. This can be seen directly by dropping $H[p]$ which doesn't depend on q. We thus showed that softmax regression tries the minimize the surprise (and thus the number of bits) we experience when seeing the true label y rather than our prediction $\hat{y}$.

6.4.4 Model Prediction and Evaluation

After training the softmax regression model, given any example features, we can predict the probability of each output category. Normally, we use the category with the highest predicted probability as the output category. The prediction is correct if it is consistent with the actual category (label). In the next part of the experiment, we will use accuracy to evaluate the model's performance. This is equal to the ratio between the number of correct predictions and the total number of predictions.

6.4.5 Summary

- We introduced the softmax operation which takes a vector maps it into probabilities.
- Softmax regression applies to classification problems. It uses the probability distribution of the output category in the softmax operation.
- Cross entropy is a good measure of the difference between two probability distributions. It measures the number of bits needed to encode the data given our model.

6.4.6 Problems

1. Show that the Kullback-Leibler divergence $D(p\|q)$ is nonnegative for all distributions p and q. Hint - use Jensen's inequality, i.e. use the fact that $-\log x$ is a convex function.
2. Show that $\log \sum_j \exp(o_j)$ is a convex function in o.

3. We can explore the connection between exponential families and the softmax in some more depth
    - Compute the second derivative of the cross entropy loss $l(y, \hat{y})$ for the softmax.
    - Compute the variance of the distribution given by softmax(o) and show that it matches the second derivative computed above.
4. Assume that we three classes which occur with equal probability, i.e. the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    - What is the problem if we try to design a binary code for it? Can we match the entropy lower bound on the number of bits?
    - Can you design a better code. Hint - what happens if we try to encode two independent observations? What if we encode n observations jointly?
5. Softmax is a misnomer for the mapping introduced above (but everyone in deep learning uses it). The real softmax is defined as RealSoftMax$(a, b) = \log(\exp(a) + \exp(b))$.
    - Prove that RealSoftMax$(a, b) > \max(a, b)$.
    - Prove that this holds for λ^{-1}RealSoftMax$(\lambda a, \lambda b)$, provided that $\lambda > 0$.
    - Show that for $\lambda \to \infty$ we have λ^{-1}RealSoftMax$(\lambda a, \lambda b) \to \max(a, b)$.
    - What does the soft-min look like?
    - Extend this to more than two numbers.

6.4.7 Scan the QR Code to Discuss

6.5 Image Classification Data (Fashion-MNIST)

Before introducing the implementation for softmax regression, we need a suitable dataset. To make things more visually compelling we pick one on classification. It will be used multiple times in later chapters to allow us to observe the difference between model accuracy and computational efficiency between comparison algorithms. The most commonly used image classification data set is the MNIST handwritten digit recognition data set. It was proposed by LeCun, Cortes and Burges in the 1990s. However, most models have a classification accuracy of over 95% on MNIST, hence it is hard to spot the difference between different models. In order to

get a better intuition about the difference between algorithms we use a more complex data set. Fashion-MNIST was proposed by Xiao, Rasul and Vollgraf in 2017.

6.5.1 Getting the Data

First, import the packages or modules required in this section.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet.gluon import data as gdata
        import sys
        import time
```

Next, we will download this data set through Gluon's `data` package. The data is automatically retrieved from the Internet the first time it is called. We specify the acquisition of a training data set, or a testing data set by the parameter `train`. The test data set, also called the testing set, is only used to evaluate the performance of the model and is not used to train the model.

```
In [2]: mnist_train = gdata.vision.FashionMNIST(train=True)
        mnist_test = gdata.vision.FashionMNIST(train=False)
```

The number of images for each category in the training set and the testing set is 6,000 and 1,000, respectively. Since there are 10 categories, the number of examples for the training set and the testing set is 60,000 and 10,000, respectively.

```
In [3]: len(mnist_train), len(mnist_test)
```

```
Out[3]: (60000, 10000)
```

We can access any example by square brackets `[]`, and next, we will get the image and label of the first example.

```
In [4]: feature, label = mnist_train[0]
```

The variable `feature` corresponds to an image with a height and width of 28 pixels. Each pixel is an 8-bit unsigned integer (uint8) with values between 0 and 255. It is stored in a 3D NDArray. Its last dimension is the number of channels. Since the data set is a grayscale image, the number of channels is 1. For the sake of simplicity, we will record the shape of the image with the height and width of h and w pixels, respectively, as $h \times w$ or `(h, w)`.

```
In [5]: feature.shape, feature.dtype
```

```
Out[5]: ((28, 28, 1), numpy.uint8)
```

The label of each image is represented as a scalar in NumPy. Its type is a 32-bit integer.

```
In [6]: label, type(label), label.dtype
```

```
Out[6]: (2, numpy.int32, dtype('int32'))
```

There are 10 categories in Fashion-MNIST: t-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. The following function can convert a numeric label into a corresponding text label.

```
In [7]: # This function has been saved in the d2l package for future use
        def get_fashion_mnist_labels(labels):
            text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                           'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
            return [text_labels[int(i)] for i in labels]
```

The following defines a function that can draw multiple images and corresponding labels in a single line.

```
In [8]: # This function has been saved in the d2l package for future use
        def show_fashion_mnist(images, labels):
            d2l.use_svg_display()
            # Here _ means that we ignore (not use) variables
            _, figs = d2l.plt.subplots(1, len(images), figsize=(12, 12))
            for f, img, lbl in zip(figs, images, labels):
                f.imshow(img.reshape((28, 28)).asnumpy())
                f.set_title(lbl)
                f.axes.get_xaxis().set_visible(False)
                f.axes.get_yaxis().set_visible(False)
```

Next, let's take a look at the image contents and text labels for the first nine examples in the training data set.

```
In [9]: X, y = mnist_train[0:9]
        show_fashion_mnist(X, get_fashion_mnist_labels(y))
```

6.5.2 Reading a Minibatch

To make our life easier when reading from the training and test sets we use a `DataLoader` rather than creating one from scratch, as we did in the section on *"Linear Regression Implementation Starting from Scratch"*. The data loader reads a mini-batch of data with an example number of `batch_size` each time.

In practice, data reading is often a performance bottleneck for training, especially when the model is simple or when the computer is fast. A handy feature of Gluon's `DataLoader` is the ability to use multiple processes to speed up data reading (not currently supported on Windows). For instance, we can set aside 4 processes to read the data (via `num_workers`).

In addition, we convert the image data from uint8 to 32-bit floating point numbers using the `ToTensor` class. Beyond that we divide all numbers by 255 so that all pixels have values between 0 and 1. The `ToTensor` class also moves the image channel from the last dimension to the first

dimension to facilitate the convolutional neural network calculations introduced later. Through the `transform_first` function of the data set, we apply the transformation of `ToTensor` to the first element of each data example (image and label), i.e., the image.

```
In [10]: batch_size = 256
         transformer = gdata.vision.transforms.ToTensor()
         if sys.platform.startswith('win'):
             # 0 means no additional processes are needed to speed up the reading of
             # data
             num_workers = 0
         else:
             num_workers = 4

         train_iter = gdata.DataLoader(mnist_train.transform_first(transformer),
                                       batch_size, shuffle=True,
                                       num_workers=num_workers)
         test_iter = gdata.DataLoader(mnist_test.transform_first(transformer),
                                      batch_size, shuffle=False,
                                      num_workers=num_workers)
```

The logic that we will use to obtain and read the Fashion-MNIST data set is encapsulated in the `d2l.load_data_fashion_mnist` function, which we will use in later chapters. This function will return two variables, `train_iter` and `test_iter`. As the content of this book continues to deepen, we will further improve this function. Its full implementation will be described in the section "Deep Convolutional Neural Networks (AlexNet)".

Let's look at the time it takes to read the training data.

```
In [11]: start = time.time()
         for X, y in train_iter:
             continue
         '%.2f sec' % (time.time() - start)
```

```
Out[11]: '1.18 sec'
```

6.5.3 Summary

- Fashion-MNIST is an apparel classification data set containing 10 categories, which we will use to test the performance of different algorithms in later chapters.
- We store the shape of image using height and width of h and w pixels, respectively, as $h \times w$ or `(h, w)`.
- Data iterators are a key component for efficient performance. Use existing ones if available.

6.5.4 Problems

1. Does reducing `batch_size` (for instance, to 1) affect read performance?

2. For non-Windows users, try modifying `num_workers` to see how it affects read performance.
3. Use the MXNet documentation to see which other datasets are available in `mxnet.gluon.data.vision`.
4. Use the MXNet documentation to see which other transformations are available in `mxnet.gluon.data.vision.transforms`.

6.5.5 Scan the QR Code to Discuss

6.6 Implementation of Softmax Regression from Scratch

Just like we learned how to implement linear regression from scratch, it is very instructive to do the same for softmax regression. After that we'll repeat the same procedure using Gluon for comparison. We begin with our regular import ritual.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, nd
```

We use the Fashion-MNIST data set with batch size 256.

```
In [2]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

6.6.1 Initialize Model Parameters

Just as in linear regression, we use vectors to represent examples. Since each example is an image with 28×28 pixels we can store it as a 784 dimensional vector. Moreover, since we have 10 categories, the single layer network has an output dimension of 10. Consequently, the weight and bias parameters of the softmax regression are matrices of size 784×10 and 1×10 respectively. We initialize W with Gaussian noise.

```
In [3]: num_inputs = 784
        num_outputs = 10
```

```
        W = nd.random.normal(scale=0.01, shape=(num_inputs, num_outputs))
        b = nd.zeros(num_outputs)
```

As before, we have to attach a gradient to the model parameters.

```
In [4]: W.attach_grad()
        b.attach_grad()
```

6.6.2 The Softmax

Before defining softmax regression let us briefly review how operators such as `sum` work along specific dimensions in an NDArray. Given a matrix `X` we can sum over all elements (default) or only over elements in the same column (`axis=0`) or the same row (`axis=1`). Moreover, we can retain the same dimensionality rather than collapsing out the dimension that we summed over, if required (`keepdims=True`).

```
In [5]: X = nd.array([[1, 2, 3], [4, 5, 6]])
        X.sum(axis=0, keepdims=True), X.sum(axis=1, keepdims=True)
```

```
Out[5]: (
         [[5. 7. 9.]]
         <NDArray 1x3 @cpu(0)>,
         [[ 6.]
          [15.]]
         <NDArray 2x1 @cpu(0)>)
```

We can now define the softmax function. For that we first exponentiate each term using `exp` and then sum each row to get the normalization constant. Last we divide each row by its normalization constant and return the result. Before looking at the code, let's look at this in equation form:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}$$

The denominator is sometimes called the partition function (and its logarithm the log-partition function). The origins of that name are in statistical physics where a related equation models the distribution over an ensemble of particles). Also note that in the definition below we are somewhat sloppy as we do not take any precautions against numerical overflow or underflow due to large (or very small) elements of the matrix, as we did in Naive Bayes.

```
In [6]: def softmax(X):
            X_exp = X.exp()
            partition = X_exp.sum(axis=1, keepdims=True)
            return X_exp / partition  # The broadcast mechanism is applied here
```

As you can see, for any random input, we turn each element into a non-negative number. Moreover, each row sums up to 1, as is required for a probability.

```
In [7]: X = nd.random.normal(shape=(2, 5))
        X_prob = softmax(X)
        X_prob, X_prob.sum(axis=1)
```

```
Out[7]: (
         [[0.21324193 0.33961776 0.1239742  0.27106097 0.05210521]
          [0.11462264 0.3461234  0.19401033 0.29583326 0.04941036]]
         <NDArray 2x5 @cpu(0)>,
         [1.0000001 1.       ]
         <NDArray 2 @cpu(0)>)
```

6.6.3 The Model

With the softmax operation, we can define the softmax regression model discussed in the last section. We change each original image into a vector with length `num inputs` through the `reshape` function.

```
In [8]: def net(X):
            return softmax(nd.dot(X.reshape((-1, num_inputs)), W) + b)
```

6.6.4 The Loss Function

In the *last section*, we introduced the cross-entropy loss function used by softmax regression. It may be the most common loss function you'll find in all of deep learning. That's because at the moment, classification problems tend to be far more abundant than regression problems.

Recall that it picks the label's predicted probability and takes its logarithm $-\log p(y|x)$. Rather than having to do this using a Python `for` loop (which tends to be inefficient) we have a `pick` function which allows us to select the appropriate terms from the matrix of softmax entries easily. We illustrate this in the case of 3 categories and 2 examples.

```
In [9]: y_hat = nd.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
        y = nd.array([0, 2], dtype='int32')
        nd.pick(y_hat, y)
```

```
Out[9]:
        [0.1 0.5]
        <NDArray 2 @cpu(0)>
```

This yields the cross-entropy loss function.

```
In [10]: def cross_entropy(y_hat, y):
             return - nd.pick(y_hat, y).log()
```

6.6.5 Classification Accuracy

Given a class of predicted probability distributions `y_hat`, we use the one with the highest predicted probability as the output category. If it is consistent with the actual category `y`, then this prediction is correct. The classification accuracy is the ratio between the number of correct predictions and the total number of predictions made.

To demostrate how to compute accuracy, the function `accuracy` is defined as follows: `y_hat.argmax(axis=1)` returns the largest element index to matrix `y_hat`, the result has the same

shape as variable `y`. Now all we need to do is check whether both match. Since the equality operator == is datatype-sensitive (e.g. an `int` and a `float32` are never equal), we also need to convert both to the same type (we pick `float32`). The result is an NDArray containing entries of 0 (false) and 1 (true). Taking the mean yields the desired result.

```
In [11]: def accuracy(y_hat, y):
             return (y_hat.argmax(axis=1) == y.astype('float32')).mean().asscalar()
```

We will continue to use the variables `y_hat` and `y` defined in the `pick` function, as the predicted probability distribution and label, respectively. We can see that the first example's prediction category is 2 (the largest element of the row is 0.6 with an index of 2), which is inconsistent with the actual label, 0. The second example's prediction category is 2 (the largest element of the row is 0.5 with an index of 2), which is consistent with the actual label, 2. Therefore, the classification accuracy rate for these two examples is 0.5.

```
In [12]: accuracy(y_hat, y)
```

```
Out[12]: 0.5
```

Similarly, we can evaluate the accuracy for model `net` on the data set `data_iter`.

```
In [13]: # The function will be gradually improved: the complete implementation will be
         # discussed in the "Image Augmentation" section
         def evaluate_accuracy(data_iter, net):
             acc_sum, n = 0.0, 0
             for X, y in data_iter:
                 y = y.astype('float32')
                 acc_sum += (net(X).argmax(axis=1) == y).sum().asscalar()
                 n += y.size
             return acc_sum / n
```

Because we initialized the `net` model with random weights, the accuracy of this model should be close to random guessing, i.e. 0.1 for 10 classes.

```
In [14]: evaluate_accuracy(test_iter, net)
```

```
Out[14]: 0.0925
```

6.6.6 Model Training

The implementation for training softmax regression is very similar to the implementation of linear regression discussed earlier. We still use the mini-batch stochastic gradient descent to optimize the loss function of the model. When training the model, the number of epochs, `num_epochs`, and learning rate `lr` are both adjustable hyper-parameters. By changing their values, we may be able to increase the classification accuracy of the model.

```
In [15]: num_epochs, lr = 5, 0.1

         # This function has been saved in the d2l package for future use
         def train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                       params=None, lr=None, trainer=None):
             for epoch in range(num_epochs):
                 train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
                 for X, y in train_iter:
```

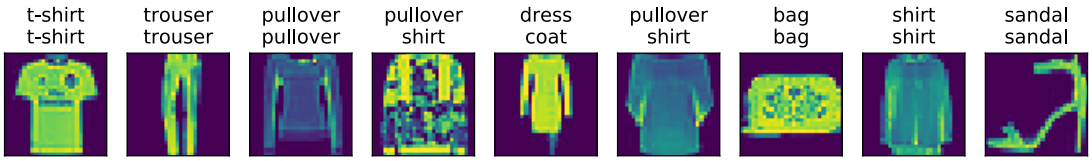
```
                    with autograd.record():
                        y_hat = net(X)
                        l = loss(y_hat, y).sum()
                    l.backward()
                    if trainer is None:
                        d2l.sgd(params, lr, batch_size)
                    else:
                        # This will be illustrated in the next section
                        trainer.step(batch_size)
                    y = y.astype('float32')
                    train_l_sum += l.asscalar()
                    train_acc_sum += (y_hat.argmax(axis=1) == y).sum().asscalar()
                    n += y.size
                test_acc = evaluate_accuracy(test_iter, net)
                print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
                      % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))

        train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs,
                  batch_size, [W, b], lr)
```

```
epoch 1, loss 0.7909, train acc 0.745, test acc 0.805
epoch 2, loss 0.5752, train acc 0.810, test acc 0.820
epoch 3, loss 0.5291, train acc 0.823, test acc 0.830
epoch 4, loss 0.5048, train acc 0.831, test acc 0.837
epoch 5, loss 0.4898, train acc 0.834, test acc 0.841
```

6.6.7 Prediction

Now that training is complete, we can show how to classify the image. Given a series of images, we will compare their actual labels (first line of text output) and the model predictions (second line of text output).

```
In [16]: for X, y in test_iter:
             break

         true_labels = d2l.get_fashion_mnist_labels(y.asnumpy())
         pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1).asnumpy())
         titles = [truelabel + '\n' + predlabel
                   for truelabel, predlabel in zip(true_labels, pred_labels)]

         d2l.show_fashion_mnist(X[0:9], titles[0:9])
```



6.6.8 Summary

We can use softmax regression to carry out multi-category classification. Training is very similar to that of linear regression: retrieve and read data, define models and loss functions, then train models using optimization algorithms. In fact, most common deep learning models have a similar training procedure.

6.6.9 Problems

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. What problems might this cause (hint - try to calculate the size of $\exp(50)$)?
2. The function `cross_entropy` in this section is implemented according to the definition of the cross-entropy loss function. What could be the problem with this implementation (hint - consider the domain of the logarithm)?
3. What solutions you can think of to fix the two problems above?
4. Is it always a good idea to return the most likely label. E.g. would you do this for medical diagnosis?
5. Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary?

6.6.10 Scan the QR Code to Discuss

6.7 Concise Implementation of Softmax Regression

We already saw that it is much more convenient to use Gluon in the context of *linear regression*. Now we will see how this applies to classification, too. We being with our import ritual.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import gluon, init
        from mxnet.gluon import loss as gloss, nn
```

We still use the Fashion-MNIST data set and the batch size set from the last section.

```
In [2]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

6.7.1 Initialize Model Parameters

As *mentioned previously*, the output layer of softmax regression is a fully connected layer. Therefore, we are adding a fully connected layer with 10 outputs. We initialize the weights at random with zero mean and standard deviation 0.01.

```
In [3]: net = nn.Sequential()
        net.add(nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

6.7.2 The Softmax

In the previous example, we calculated our model's output and then ran this output through the cross-entropy loss. At its heart it uses `-nd.pick(y_hat, y).log()`. Mathematically, that's a perfectly reasonable thing to do. However, computationally, things can get hairy, as we've already alluded to a few times (e.g. in the context of Naive Bayes and in the problem set of the previous chapter). Recall that the softmax function calculates $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}$, where $\hat{y}_j$ is the j-th element of `yhat` and z_j is the j-th element of the input `y_linear` variable, as computed by the softmax.

If some of the z_i are very large (i.e. very positive), e^{z_i} might be larger than the largest number we can have for certain types of `float` (i.e. overflow). This would make the denominator (and/or numerator) `inf` and we get zero, or `inf`, or `nan` for $\hat{y}_j$. In any case, we won't get a well-defined return value for `cross_entropy`. This is the reason we subtract $\max(z_i)$ from all z_i first in `softmax` function. You can verify that this shifting in z_i will not change the return value of `softmax`.

After the above subtraction/ normalization step, it is possible that z_j is very negative. Thus, e^{z_j} will be very close to zero and might be rounded to zero due to finite precision (i.e underflow), which makes $\hat{y}_j$ zero and we get `-inf` for $\log(\hat{y}_j)$. A few steps down the road in backpropagation, we start to get horrific not-a-number (`nan`) results printed to screen.

Our salvation is that even though we're computing these exponential functions, we ultimately plan to take their log in the cross-entropy functions. It turns out that by combining these two operators `softmax` and `cross_entropy` together, we can elude the numerical stability issues that might otherwise plague us during backpropagation. As shown in the equation below, we

avoided calculating e^{z_j} but directly used z_j due to $\log(\exp(\cdot))$.

$$\begin{aligned}\log\left(\hat{y}_j\right) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right) \\ &= \log\left(e^{z_j}\right) - \log\left(\sum_{i=1}^{n} e^{z_i}\right) \\ &= z_j - \log\left(\sum_{i=1}^{n} e^{z_i}\right)\end{aligned}$$

We'll want to keep the conventional softmax function handy in case we ever want to evaluate the probabilities output by our model. But instead of passing softmax probabilities into our new loss function, we'll just pass $\hat{y}$ and compute the softmax and its log all at once inside the softmax_cross_entropy loss function, which does smart things like the log-sum-exp trick (see on Wikipedia).

```
In [4]: loss = gloss.SoftmaxCrossEntropyLoss()
```

6.7.3 Optimization Algorithm

We use the mini-batch random gradient descent with a learning rate of 0.1 as the optimization algorithm. Note that this is the same choice as for linear regression and it illustrates the portability of the optimizers.

```
In [5]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

6.7.4 Training

Next, we use the training functions defined in the last section to train a model.

```
In [6]: num_epochs = 5
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)
```

```
epoch 1, loss 0.7922, train acc 0.745, test acc 0.797
epoch 2, loss 0.5731, train acc 0.812, test acc 0.826
epoch 3, loss 0.5287, train acc 0.824, test acc 0.832
epoch 4, loss 0.5047, train acc 0.830, test acc 0.831
epoch 5, loss 0.4884, train acc 0.835, test acc 0.841
```

Just as before, this algorithm converges to a fairly decent accuracy of 83.7%, albeit this time with a lot fewer lines of code than before. Note that in many cases Gluon takes specific precautions beyond what one would naively do to ensure numerical stability. This takes care of many common pitfalls when coding a model from scratch.

6.7.5 Problems

1. Try adjusting the hyper-parameters, such as batch size, epoch, and learning rate, to see what the results are.
2. Why might the test accuracy decrease again after a while? How could we fix this?

6.7.6 Scan the QR Code to Discuss

6.8 Multilayer Perceptron

In the previous chapters we showed how you could implement multiclass logistic regression (also called softmax regression) for classifying images of clothing into the 10 possible categories. This is where things start to get fun. We understand how to wrangle data, coerce our outputs into a valid probability distribution (via `softmax`), how to apply an appropriate loss function, and how to optimize over our parameters. Now that we've covered these preliminaries, we can extend our toolbox to include deep neural networks.

6.8.1 Hidden Layers

Recall that before, we mapped our inputs directly onto our outputs through a single linear transformation via

$$\hat{\mathbf{o}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

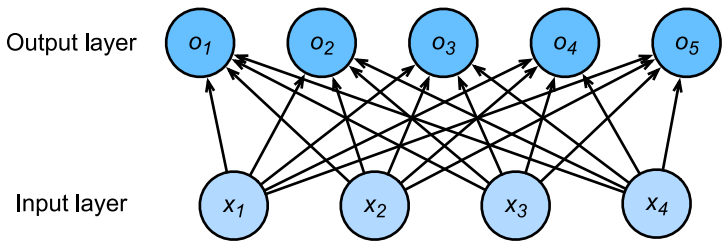


Fig. 6.6: Single layer perceptron with 5 output units.

If our labels really were related to our input data by an approximately linear function, then this approach might be adequate. But linearity is a *strong assumption*. Linearity means that given an output of interest, for each input, increasing the value of the input should either drive the value of the output up or drive it down, irrespective of the value of the other inputs.

From one to many

Imagine the case of classifying cats and dogs based on black and white images. That's like saying that for each pixel, increasing its value either increases the probability that it depicts a dog or decreases it. That's not reasonable. After all, the world contains both black dogs and black cats, and both white dogs and white cats.

Teasing out what is depicted in an image generally requires allowing more complex relationships between our inputs and outputs, considering the possibility that our pattern might be characterized by interactions among the many features. In these cases, linear models will have low accuracy. We can model a more general class of functions by incorporating one or more hidden layers. The easiest way to do this is to stack a bunch of layers of neurons on top of each other. Each layer feeds into the layer above it, until we generate an output. This architecture is commonly called a "multilayer perceptron". With an MLP, we stack a bunch of layers on top of each other. Here's an example:

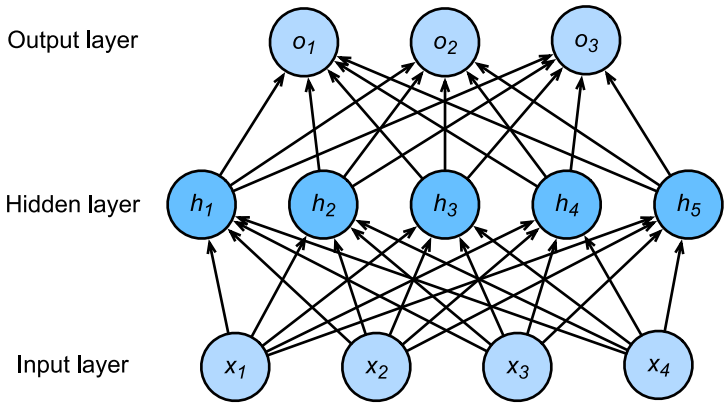


Fig. 6.7: Multilayer perceptron with hidden layers. This example contains a hidden layer with 5 hidden units in it.

In the multilayer perceptron above, the number of inputs and outputs is 4 and 3 respectively, and the hidden layer in the middle contains 5 hidden units. Since the input layer does not involve any calculations, there are a total of 2 layers in the multilayer perceptron. The neurons in the hidden layer are fully connected to the inputs within the input layer. The neurons in the output layer and the neurons in the hidden layer are also fully connected. Therefore, both the hidden layer and the output layer in the multilayer perceptron are fully connected layers.

From linear to nonlinear

Let us write out what is happening mathematically in the picture above, e.g. for multiclass classification.

$$\begin{aligned}\mathbf{h} &= \mathbf{W}_1\mathbf{x} + \mathbf{b}_1\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2\\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{o})\end{aligned}$$

The problem with the approach above is that we have gained nothing over a simple single layer perceptron since we can collapse out the hidden layer by an equivalently parametrized single layer perceptron using $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$ and $\mathbf{b} = \mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2$.

$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

To fix this we need another key ingredient - a nonlinearity σ such as $\max(x, 0)$ after each layer. Once we do this, it becomes impossible to merge layers. This yields

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2\\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{o})\end{aligned}$$

Clearly we could continue stacking such hidden layers, e.g. $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ and $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$ on top of each other to obtain a true multilayer perceptron.

Multilayer perceptrons can account for complex interactions in the inputs because the hidden neurons depend on the values of each of the inputs. It's easy to design a hidden node that that does arbitrary computation, such as, for instance, logical operations on its inputs. And it's even widely known that multilayer perceptrons are universal approximators. That means that even for a single-hidden-layer neural network, with enough nodes, and the right set of weights, it could model any function at all! Actually learning that function is the hard part. And it turns out that we can approximate functions much more compactly if we use deeper (vs wider) neural networks. We'll get more into the math in a subsequent chapter, but for now let's actually build an MLP. In this example, we'll implement a multilayer perceptron with two hidden layers and one output layer.

Vectorization and mini-batch

When given a mini-batch of samples we can use vectorization to gain better efficiency in implementation. In a nutshell, we replace vectors by matrices. As before, denote by $\mathbf{X}$ the matrix of

inputs from a minibatch. Then an MLP with two hidden layers can be expressed as

$$\begin{aligned}\mathbf{H}_1 &= \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1)\\ \mathbf{H}_2 &= \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2)\\ \mathbf{O} &= \mathrm{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3)\end{aligned}$$

This is easy to implement and easy to optimize. With some abuse of notation we define the nonlinearity σ to apply to its inputs on a row-wise fashion, i.e. one observation at a time, often one coordinate at a time. This is true for most activation functions (the batch normalization is a notable exception from that rule).

6.8.2 Activation Functions

Let us look a bit more at examples of activation functions. After all, it is this alternation between linear and nonlinear terms that makes deep networks work. A rather popular choice, due to its simplicity of implementation and its efficacy is the ReLu function.

ReLU Function

The ReLU (rectified linear unit) function provides a very simple nonlinear transformation. Given the element x, the function is defined as

$$\mathrm{ReLU}(x) = \max(x, 0).$$

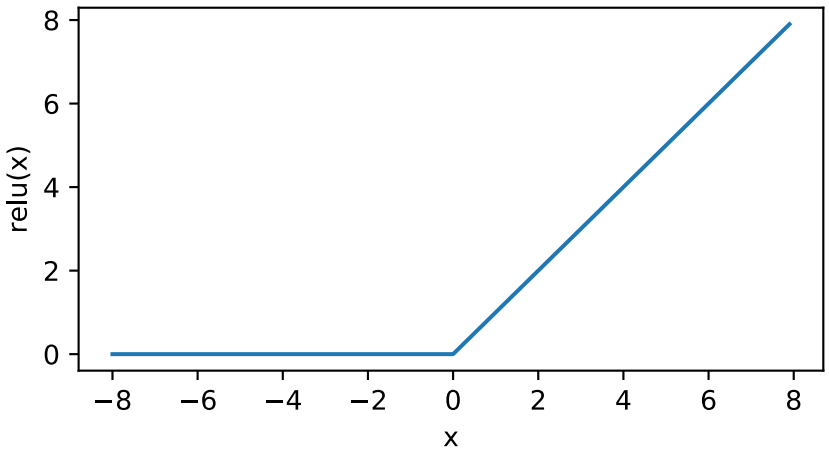
It can be understood that the ReLU function retains only positive elements and discards negative elements. To get a better idea of what it looks like it helps to plot it. For convenience we define a plotting function `xyplot` to take care of the gruntwork.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, nd

        def xyplot(x_vals, y_vals, name):
            d2l.set_figsize(figsize=(5, 2.5))
            d2l.plt.plot(x_vals.asnumpy(), y_vals.asnumpy())
            d2l.plt.xlabel('x')
            d2l.plt.ylabel(name + '(x)')
```

Then, we can plot the ReLU function using the `relu` function provided by NDArray. As you can see, the activation function is a two-stage linear function.

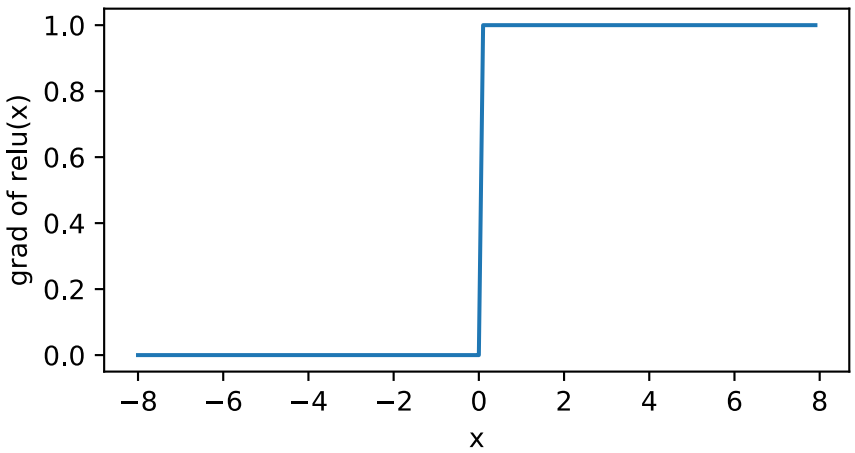
```
In [2]: x = nd.arange(-8.0, 8.0, 0.1)
        x.attach_grad()
        with autograd.record():
```

```
    y = x.relu()
xyplot(x, y, 'relu')
```



Obviously, when the input is negative, the derivative of ReLU function is 0; when the input is positive, the derivative of ReLU function is 1. Note that the ReLU function is not differentiable when the input is 0. Instead, we pick its left-hand-side (LHS) derivative 0 at location 0. The derivative of the ReLU function is plotted below.

```
In [3]: y.backward()
        xyplot(x, x.grad, 'grad of relu')
```



Note that there are many variants to the ReLU function, such as the parameterized ReLU (pReLU) of He et al., 2015. Effectively it adds a linear term to the ReLU, so some information

still gets through, even when the argument is negative.

$$\text{pReLU}(x) = \max(0, x) - \alpha x$$

The reason for using the ReLU is that its derivatives are particularly well behaved - either they vanish or they just let the argument through. This makes optimization better behaved and it reduces the issue of the vanishing gradient problem (more on this later).

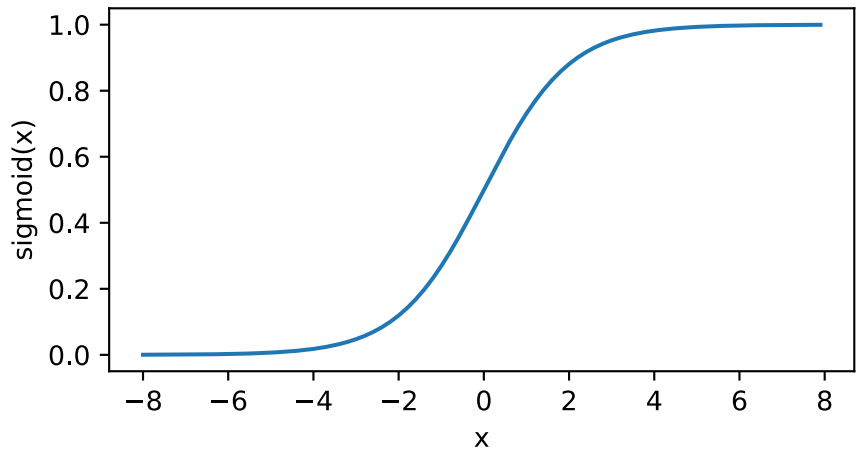
Sigmoid Function

The Sigmoid function can transform the value of an element in $\mathbb{R}$ to the interval $(0, 1)$.

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

The Sigmoid function was commonly used in early neural networks, but is currently being replaced by the simpler ReLU function. In the "Recurrent Neural Network"chapter, we will describe how to utilize the function's ability to control the flow of information in a neural network thanks to its capacity to transform the value range between 0 and 1. The derivative of the Sigmoid function is plotted below. When the input is close to 0, the Sigmoid function approaches a linear transformation.

```
In [4]: with autograd.record():
            y = x.sigmoid()
        xyplot(x, y, 'sigmoid')
```

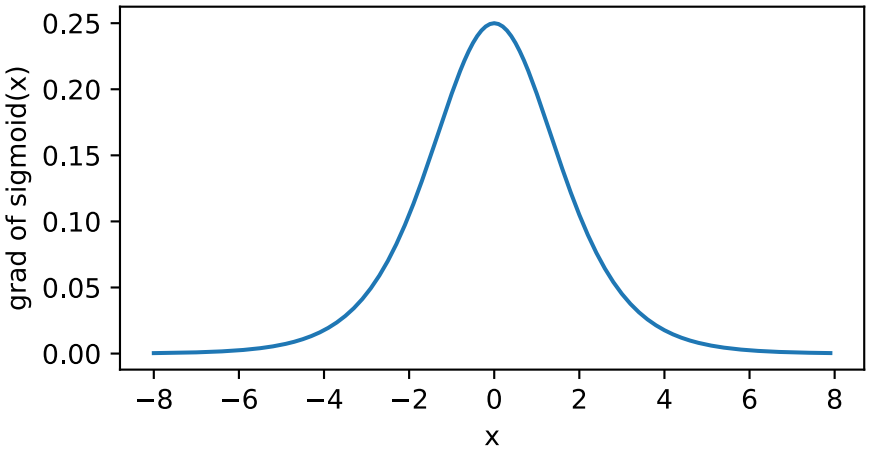


The derivative of Sigmoid function is as follows:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)\,(1 - \text{sigmoid}(x))\,.$$

The derivative of Sigmoid function is plotted below. When the input is 0, the derivative of the Sigmoid function reaches a maximum of 0.25; as the input deviates further from 0, the derivative of Sigmoid function approaches 0.

```
In [5]: y.backward()
        xyplot(x, x.grad, 'grad of sigmoid')
```



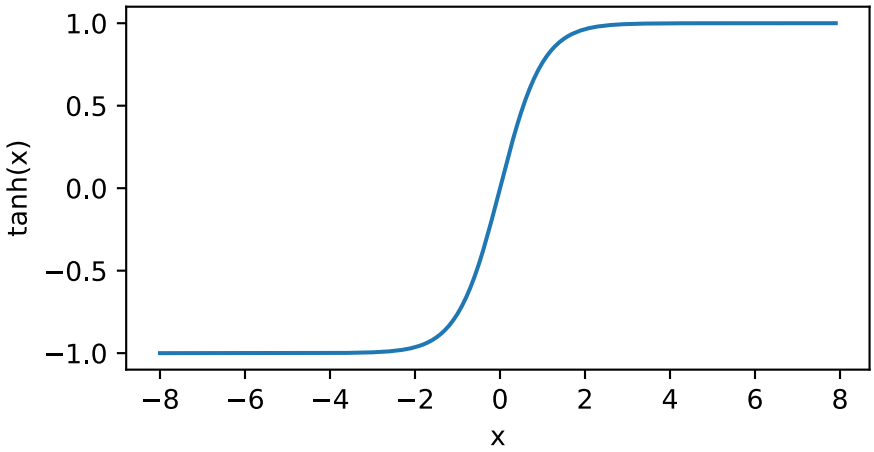
Tanh Function

The Tanh (Hyperbolic Tangent) function transforms the value of an element to the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

We can then plot the Tanh function. As the input nears 0, the Tanh function approaches linear transformation. Although the shape of the function is similar to that of the Sigmoid function, the Tanh function is symmetric at the origin of the coordinate system.

```
In [6]: with autograd.record():
            y = x.tanh()
        xyplot(x, y, 'tanh')
```

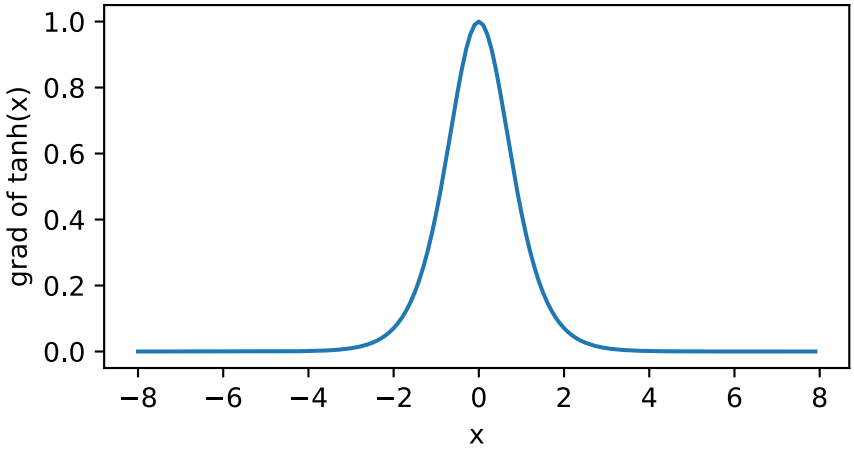


The derivative of the Tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

The derivative of Tanh function is plotted below. As the input nears 0, the derivative of the Tanh function approaches a maximum of 1; as the input deviates away from 0, the derivative of the Tanh function approaches 0.

```
In [7]: y.backward()
        xyplot(x, x.grad, 'grad of tanh')
```



In summary, we have a range of nonlinearities and now know how to layer them to build quite powerful network architectures. As a side note, we have now pretty much reached the state of the art in deep learning, anno 1990. The main difference is that we have a powerful deep learn-

ing framework which lets us build models in a few lines of code where previously thousands of lines of C and Fortran would have been needed.

6.8.3 Summary

- The multilayer perceptron adds one or multiple fully connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.
- Commonly used activation functions include the ReLU function, the Sigmoid function, and the Tanh function.

6.8.4 Problems

1. Compute the derivative of the Tanh and the pReLU activation function.
2. Show that a multilayer perceptron using only ReLU (or pReLU) constructs a continuous piecewise linear function.
3. Show that $\tanh(x) + 1 = 2\text{sigmoid}(2x)$.
4. Assume we have a multilayer perceptron *without* nonlinearities between the layers. In particular, assume that we have d input dimensions, d output dimensions and that one of the layers had only $d/2$ dimensions. Show that this network is less expressive (powerful) than a single layer perceptron.
5. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems to you expect this to cause?

6.8.5 Scan the QR Code to Discuss

6.9 Implementation of Multilayer Perceptron from Scratch

Now that we learned how multilayer perceptrons (MLPs) work in theory, let's implement them. First, import the required packages or modules.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import nd
        from mxnet.gluon import loss as gloss
```

We continue to use the Fashion-MNIST data set. We will use the Multilayer Perceptron for image classification

```
In [2]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

6.9.1 Initialize Model Parameters

We know that the dataset contains 10 classes and that the images are of $28 \times 28 = 784$ pixel resolution. Thus the number of inputs is 784 and the number of outputs is 10. Moreover, we use an MLP with one hidden layer and we set the number of hidden units to 256, but we could have picked some other value for this *hyperparameter*, too. Typically one uses powers of 2 since things align more nicely in memory.

```
In [3]: num_inputs, num_outputs, num_hiddens = 784, 10, 256

        W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens))
        b1 = nd.zeros(num_hiddens)
        W2 = nd.random.normal(scale=0.01, shape=(num_hiddens, num_outputs))
        b2 = nd.zeros(num_outputs)
        params = [W1, b1, W2, b2]

        for param in params:
            param.attach_grad()
```

6.9.2 Activation Function

Here, we use the underlying `maximum` function to implement the ReLU, instead of invoking `ReLU` directly.

```
In [4]: def relu(X):
            return nd.maximum(X, 0)
```

6.9.3 The model

As in softmax regression, using `reshape` we change each original image to a length vector of `num_inputs`. We then implement implement the MLP just as discussed previously.

```
In [5]: def net(X):
            X = X.reshape((-1, num_inputs))
            H = relu(nd.dot(X, W1) + b1)
            return nd.dot(H, W2) + b2
```

6.9.4 The Loss Function

For better numerical stability, we use Gluon's functions, including softmax calculation and cross-entropy loss calculation. We discussed the intricacies of that in the *previous section*. This is simply to avoid lots of fairly detailed and specific code (the interested reader is welcome to look at the source code for more details, something that is useful for implementing other related functions).

```
In [6]: loss = gloss.SoftmaxCrossEntropyLoss()
```

6.9.5 Training

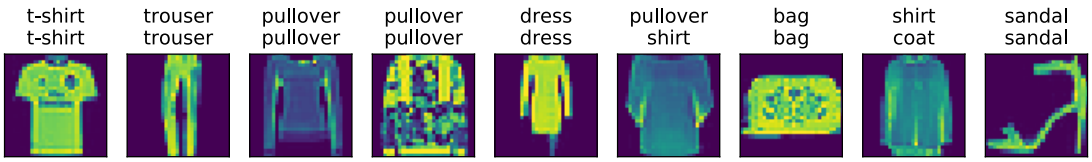
Steps for training the Multilayer Perceptron are no different from Softmax Regression training steps. In the `d2l` package, we directly call the `train_ch3` function, whose implementation was introduced *here*. We set the number of epochs to 10 and the learning rate to 0.5.

```
In [7]: num_epochs, lr = 10, 0.5
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                      params, lr)
```

```
epoch 1, loss 0.7906, train acc 0.704, test acc 0.789
epoch 2, loss 0.4930, train acc 0.817, test acc 0.832
epoch 3, loss 0.4354, train acc 0.839, test acc 0.858
epoch 4, loss 0.3988, train acc 0.852, test acc 0.853
epoch 5, loss 0.3713, train acc 0.863, test acc 0.866
epoch 6, loss 0.3570, train acc 0.868, test acc 0.876
epoch 7, loss 0.3377, train acc 0.876, test acc 0.879
epoch 8, loss 0.3271, train acc 0.879, test acc 0.880
epoch 9, loss 0.3174, train acc 0.882, test acc 0.878
epoch 10, loss 0.3073, train acc 0.886, test acc 0.876
```

To see how well we did, let's apply the model to some test data. If you're interested, compare the result to corresponding *linear model*.

```
In [8]: for X, y in test_iter:
            break

        true_labels = d2l.get_fashion_mnist_labels(y.asnumpy())
        pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1).asnumpy())
        titles = [truelabel + '\n' + predlabel
                  for truelabel, predlabel in zip(true_labels, pred_labels)]

        d2l.show_fashion_mnist(X[0:9], titles[0:9])
```



This looks slightly better than before, a clear sign that we're on to something good here.

6.9.6 Summary

We saw that implementing a simple MLP is quite easy, when done manually. That said, for a large number of layers this can get quite complicated (e.g. naming the model parameters, etc).

6.9.7 Problems

1. Change the value of the hyper-parameter `num_hiddens` in order to see the result effects.
2. Try adding a new hidden layer to see how it affects the results.
3. How does changing the learning rate change the result.
4. What is the best result you can get by optimizing over all the parameters (learning rate, iterations, number of hidden layers, number of hidden units per layer)?

6.9.8 Scan the QR Code to Discuss

6.10 Concise Implementation of Multilayer Perceptron

Now that we learned how multilayer perceptrons (MLPs) work in theory, let's implement them. We begin, as always, by importing modules.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init
        from mxnet.gluon import loss as gloss, nn
```

6.10.1 The Model

The only difference from the softmax regression is the addition of a fully connected layer as a hidden layer. It has 256 hidden units and uses ReLU as the activation function.

```
In [2]: net = nn.Sequential()
        net.add(nn.Dense(256, activation='relu'))
        net.add(nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

One minor detail is of note when invoking `net.add()`. It adds one or more layers to the network. That is, an equivalent to the above lines would be `net.add(nn.Dense(256, activation='relu'), nn.Dense(10))`. Also note that Gluon automagically infers the missing parameteters, such as the fact that the second layer needs a matrix of size 256×10. This happens the first time the network is invoked.

We use almost the same steps for softmax regression training as we do for reading and training the model.

```
In [3]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)

        loss = gloss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
        num_epochs = 10
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)
```

```
epoch 1, loss 0.8046, train acc 0.699, test acc 0.825
epoch 2, loss 0.4917, train acc 0.818, test acc 0.847
epoch 3, loss 0.4262, train acc 0.842, test acc 0.853
epoch 4, loss 0.3962, train acc 0.854, test acc 0.869
epoch 5, loss 0.3762, train acc 0.862, test acc 0.875
epoch 6, loss 0.3580, train acc 0.867, test acc 0.877
epoch 7, loss 0.3409, train acc 0.875, test acc 0.875
epoch 8, loss 0.3293, train acc 0.878, test acc 0.878
epoch 9, loss 0.3165, train acc 0.884, test acc 0.882
epoch 10, loss 0.3115, train acc 0.885, test acc 0.878
```

6.10.2 Problems

1. Try adding a few more hidden layers to see how the result changes.
2. Try out different activation functions. Which ones work best?
3. Try out different initializations of the weights.

6.10.3 Scan the QR Code to Discuss

6.11 Model Selection, Underfitting and Overfitting

In machine learning, our goal is to discover general patterns. For example, we might want to learn an association between genetic markers and the development of dementia in adulthood. Our hope would be to uncover a pattern that could be applied successfully to assess risk for the entire population.

However, when we train models, we don't have access to the entire population (or current or potential humans). Instead, we can access only a small, finite sample. Even in a large hospital system, we might get hundreds of thousands of medical records. Given such a finite sample size, it's possible to uncover spurious associations that don't hold up for unseen data.

Let's consider an extreme pathological case. Imagine that you want to learn to predict which people will repay their loans. A lender hires you as a data scientist to investigate the case and gives you complete files on 100 applicants, of which 5 defaulted on their loans within 3 years. The files might include hundreds of features including income, occupation, credit score, length of employment etcetera. Imagine that they additionally give you video footage of their interview with a lending agent. That might seem like a lot of data!

Now suppose that after generating an enormous set of features, you discover that of the 5 applicants who defaults, all 5 were wearing blue shirts during their interviews, while only 40% of general population wore blue shirts. There's a good chance that any model you train would pick up on this signal and use it as an important part of its learned pattern.

Even if defaulters are no more likely to wear blue shirts, there's a 1% chance that we'll observe all five defaulters wearing blue shirts. And keeping the sample size low while we have hundreds or thousands of features, we may observe a large number of spurious correlations. Given trillions of training examples, these false associations might disappear. But we seldom have that luxury.

The phenomena of fitting our training distribution more closely than the real distribution is called overfitting, and the techniques used to combat overfitting are called regularization. More to the point, in the previous sections we observed this effect on the Fashion-MNIST dataset. If you altered the model structure or the hyper-parameters during the experiment, you may have found that for some choices the model might not have been as accurate when using the testing data set compared to when the training data set was used.

6.11.1 Training Error and Generalization Error

Before we can explain this phenomenon, we need to differentiate between training and a generalization error. In layman's terms, training error refers to the error exhibited by the model during its use of the training data set and generalization error refers to any expected error when applying the model to an imaginary stream of additional data drawn from the underlying data distribution. The latter is often estimated by applying the model to the test set. In the previously discussed loss functions, for example, the squared loss function used for linear regression or the cross-entropy loss function used for softmax regression, can be used to calculate training and generalization error rates.

The following three thought experiments will help illustrate this situation better. Consider a college student trying to prepare for his final exam. A diligent student will strive to practice well and test his abilities using exams from previous years. Nonetheless, doing well on past exams is no guarantee that he will excel when it matters. For instance, the student might try to prepare by rote learning the answers to the exam questions. This requires the student to memorize many things. He might even remember the answers for past exams perfectly. Another student might prepare by trying to understand the reasons for giving certain answers. In most cases, the latter student will do much better.

Likewise, we would expect that a model that simply performs table lookup to answer questions. If the inputs are discrete, this might very well work after seeing *many* examples. Nonetheless, such a model is unlikely to work well in practice, as data is often real-valued and more scarce than we would like. Furthermore, we only have a finite amount of RAM to store our model in.

Lastly, consider a trivial classification problem. Our training data consists of labels only, namely 0 for heads and 1 for tails, obtained by tossing a fair coin. No matter what we do, the generalization error will always be $\frac{1}{2}$. Yet the training error may be quite a bit less than that, depending on the luck of the draw. E.g. for the dataset {0, 1, 1, 1, 0, 1} we will 'predict' class 1 and incur an error of $\frac{1}{3}$, which is considerably better than what it should be. We can also see that as we increase the amount of data, the probability for large deviations from $\frac{1}{2}$ will diminish and the training error will be close to it, too. This is because our model 'overfit' to the data and since things will 'average out' as we increase the amount of data.

Statistical Learning Theory

There is a formal theory of this phenomenon. Glivenko and Cantelli derived in their eponymous theorem the rate at which the training error converges to the generalization error. In a series of seminal papers Vapnik and Chervonenkis extended this to much more general function classes. This laid the foundations of Statistical Learning Theory.

Unless stated otherwise, we assume that both the training set and the test set are drawn independently and identically drawn from the same distribution. This means that in drawing from the distribution there is no memory between draws. Moreover, it means that we use the same distribution in both cases. Obvious cases where this might be violated is if we want to build a face recognition by training it on elementary students and then want to deploy it in the general population. This is unlikely to work since, well, the students tend to look quite from the general population. By training we try to find a function that does particularly well on the training data. If the function is very fleible such as to be able to adapt well to any details in the training data, it might do a bit too well. This is precisely what we want to avoid (or at least control). Instead we want to find a model that reduces the generalization error. A lot of tuning in deep learning is devoted to making sure that this does not happen.

Model Complexity

When we have simple models and abundant data, we expect the generalization error to resemble the training error. When we work with more complex models and fewer examples, we expect the training error to go down but the generalization gap to grow. What precisely constitutes model complexity is a complex matter. Many factors govern whether a model will generalize well. For example a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. Often with neural networks, we think of a model that takes more training steps as more complex, and one subject to early stopping as less complex.

It can be difficult to compare the complexity among members of very different model classes (say decision trees versus neural networks). For now a simple rule of thumb is quite useful: A model that can readily explain arbitrary facts is what statisticians view as complex, whereas one that has only a limited expressive power but still manages to explain the data well is probably closer to the truth. In philosophy this is closely related to Popper's criterion of falsifiability of a scientific theory: a theory is good if it fits data and if there are specific tests which can be used to disprove it. This is important since all statistical estimation is post hoc, i.e. we estimate after we observe the facts, hence vulnerable to the associated fallacy. Ok, enough of philosophy, let's get to more tangible issues. To give you some intuition in this chapter, we'll focus on a few factors that tend to influence the generalizability of a model class:

1. The number of tunable parameters. When the number of tunable parameters, sometimes denoted as the number of degrees of freedom, is large, models tend to be more susceptible to overfitting.
2. The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to over fitting.
3. The number of training examples. It's trivially easy to overfit a dataset containing only one or two examples even if your model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

6.11.2 Model Selection

In machine learning we usually select our model based on an evaluation of the performance of several candidate models. This process is called model selection. The candidate models can be similar models using different hyper-parameters. Using the multilayer perceptron as an example, we can select the number of hidden layers as well as the number of hidden units, and activation functions in each hidden layer. A significant effort in model selection is usually required in order to end up with an effective model. In the following section we will be describing the validation data set often used in model selection.

Validation Data Set

Strictly speaking, the test set can only be used after all the hyper-parameters and model parameters have been selected. In particular, the test data must not be used in model selection process, such as in the tuning of hyper-parameters. We should not rely solely on the training data during model selection, since the generalization error rate cannot be estimated from the training error rate. Bearing this in mind, we can reserve a portion of data outside of the training and testing data sets to be used in model selection. This reserved data is known as the validation data set, or validation set. For example, a small, randomly selected portion from a given training set can be used as a validation set, with the remainder used as the true training set.

However, in practical applications the test data is rarely discarded after one use since it's not easily obtainable. Therefore, in practice, there may be unclear boundaries between validation and testing data sets. Unless explicitly stated, the test data sets used in the experiments provided in this book should be considered as validation sets, and the test accuracy in the experiment report are for validation accuracy. The good news is that we don't need too much data in the validation set. The uncertainty in our estimates can be shown to be of the order of $O(n^{-\frac{1}{2}})$.

K-Fold Cross-Validation

When there is not enough training data, it is considered excessive to reserve a large amount of validation data, since the validation data set does not play a part in model training. A solution to this is the K-fold cross-validation method. In K-fold cross-validation, the original training data set is split into K non-coincident sub-data sets. Next, the model training and validation process is repeated K times. Every time the validation process is repeated, we validate the model using a sub-data set and use the $K-1$ sub-data set to train the model. The sub-data set used to validate the model is continuously changed throughout this K training and validation process. Finally, the average over K training and validation error rates are calculated respectively.

6.11.3 Underfitting and Overfitting

Next, we will look into two common problems that occur during model training. One type of problem occurs when the model is unable to reduce training errors since the model is too simplistic. This phenomenon is known as underfitting. As discussed, another type of problem is when the number of model training errors is significantly less than that of the testing data set, also known as overfitting. In practice, both underfitting and overfitting should be dealt with simultaneously whenever possible. Although many factors could cause the above two fitting problems, for the time being we'll be focusing primarily on two factors: model complexity and training data set size.

Model Complexity

We use polynomials as a way to illustrate the issue. Given training data consisting of the scalar data feature x and the corresponding scalar label y, we try to find a polynomial of degree d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i$$

to estimate y. Here w_i refers to the model's weight parameter. The bias is implicit in w_0 since $x^0 = 1$. Similar to linear regression we also use a squared loss for simplicity (note that $d = 1$ we recover linear regression).

A higher order polynomial function is more complex than a lower order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider. Therefore, using the same training data set, higher order polynomial functions should be able to achieve a lower training error rate (relative to lower degree polynomials). Bearing in mind the given training data set, the typical relationship between model complexity and error is shown in the diagram below. If the model is too simple for the dataset, we are likely to see underfitting, whereas if we pick an overly complex model we see overfitting. Choosing an appropriately complex model for the data set is one way to avoid underfitting and overfitting.

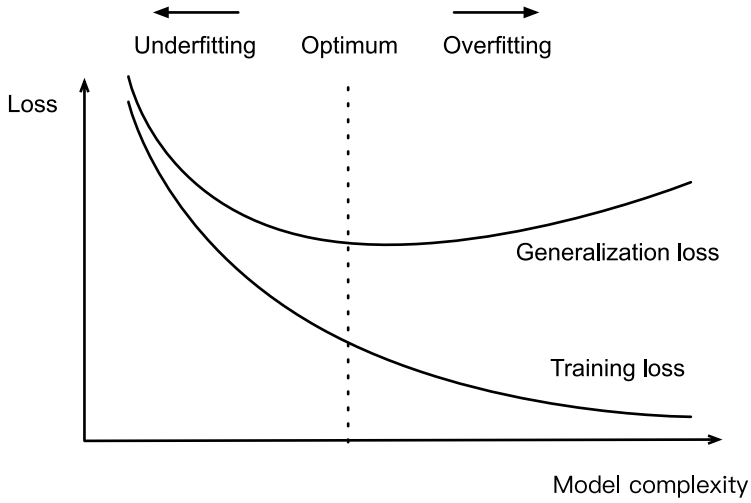


Fig. 6.8: Influence of Model Complexity on Underfitting and Overfitting

Data Set Size

Another influence is the amount of training data. Typically, if there are not enough samples in the training data set, especially if the number of samples is less than the number of model

parameters (count by element), overfitting is more likely to occur. Additionally, as we increase the amount of training data, the generalization error typically decreases. This means that more data never hurts. Moreover, it also means that we should typically only use complex models (e.g. many layers) if we have sufficient data.

6.11.4 Polynomial Regression

Let us try how this works in practice by fitting polynomials to data. As before we start by importing some modules.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
```

Generating Data Sets

First we need data. Given x we will use the following cubic polynomial to generate the labels on training and test data:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1)$$

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. The number of samples for both the training and the testing data sets is set to 100.

```
In [2]: maxdegree = 20  # Maximum degree of the polynomial
        n_train, n_test = 100, 1000  # Training and test data set sizes
        true_w = nd.zeros(maxdegree)  # Allocate lots of empty space
        true_w[0:4] = nd.array([5, 1.2, -3.4, 5.6])

        features = nd.random.normal(shape=(n_train + n_test, 1))
        features = nd.random.shuffle(features)
        poly_features = nd.power(features, nd.arange(maxdegree).reshape((1, -1)))
        poly_features = poly_features / (
            nd.gamma(nd.arange(maxdegree) + 1).reshape((1, -1)))
        labels = nd.dot(poly_features, true_w)
        labels += nd.random.normal(scale=0.1, shape=labels.shape)
```

For optimization we typically want to avoid very large values of gradients, losses, etc.; This is why the monomials stored in `poly_features` are rescaled from x^i to $\frac{1}{i!}x^i$. It allows us to avoid very large values for large exponents i. Factorials are implemented in Gluon using the Gamma function, where $n! = \Gamma(n+1)$.

Take a look at the first 2 samples from the generated data set. The value 1 is technically a feature, namely the constant feature corresponding to the bias.

```
In [3]: features[:2], poly_features[:2], labels[:2]
```

```
Out[3]: (
         [[-0.5095612 ]
          [ 0.34202248]]
         <NDArray 2x1 @cpu(0)>,
         [[ 1.00000000e+00 -5.09561181e-01  1.29826277e-01 -2.20514797e-02
            2.80914456e-03 -2.86286173e-04  2.43133891e-05 -1.76987987e-06
            1.12732764e-07 -6.38269260e-09  3.25237282e-10 -1.50662070e-11
            6.39762874e-13 -2.50767950e-14  9.12725858e-16 -3.10059752e-17
            9.87465261e-19 -2.95984643e-20  8.37901598e-22 -2.24716890e-23]
          [ 1.00000000e+00  3.42022479e-01  5.84896803e-02  6.66826218e-03
            5.70173899e-04  3.90024616e-05  2.22328640e-06  1.08630559e-07
            4.64426186e-09  1.76493528e-10  6.03647601e-12  1.87691835e-13
            5.34956872e-15  1.40744067e-16  3.43840286e-18  7.84007342e-20
            1.67592618e-21  3.37178999e-23  6.40682210e-25  1.15330363e-26]]
         <NDArray 2x20 @cpu(0)>,
         [3.8980482 5.3267784]
         <NDArray 2 @cpu(0)>)
```

Defining, Training and Testing Model

We first define the plotting function `semilogy`, where the y axis makes use of the logarithmic scale.

```
In [4]: # This function has been saved in the d2l package for future use
        def semilogy(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
                     legend=None, figsize=(3.5, 2.5)):
            d2l.set_figsize(figsize)
            d2l.plt.xlabel(x_label)
            d2l.plt.ylabel(y_label)
            d2l.plt.semilogy(x_vals, y_vals)
            if x2_vals and y2_vals:
                d2l.plt.semilogy(x2_vals, y2_vals, linestyle=':')
                d2l.plt.legend(legend)
```

Similar to linear regression, polynomial function fitting also makes use of a squared loss function. Since we will be attempting to fit the generated data set using models of varying complexity, we insert the model definition into the `fit_and_plot` function. The training and testing steps involved in polynomial function fitting are similar to those previously described in softmax regression.

```
In [5]: num_epochs, loss = 200, gloss.L2Loss()

        def fit_and_plot(train_features, test_features, train_labels, test_labels):
            net = nn.Sequential()
            # Switch off the bias since we already catered for it in the polynomial
            # features
            net.add(nn.Dense(1, use_bias=False))
            net.initialize()
            batch_size = min(10, train_labels.shape[0])
            train_iter = gdata.DataLoader(gdata.ArrayDataset(
                train_features, train_labels), batch_size, shuffle=True)
            trainer = gluon.Trainer(net.collect_params(), 'sgd',
```

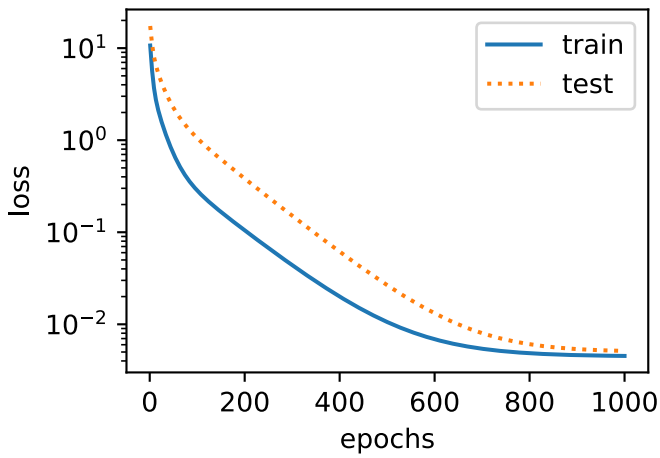
```
                            {'learning_rate': 0.01})
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(loss(net(train_features),
                             train_labels).mean().asscalar())
        test_ls.append(loss(net(test_features),
                            test_labels).mean().asscalar())
    print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
    semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
             range(1, num_epochs + 1), test_ls, ['train', 'test'])
    print('weight:', net[0].weight.data().asnumpy())
```

Third-order Polynomial Function Fitting (Normal)

We will begin by first using a third-order polynomial function with the same order as the data generation function. The results show that this model's training error rate when using the testing data set is low. The trained model parameters are also close to the true values $w = [5, 1.2, -3.4, 5.6]$.

```
In [6]: num_epochs = 1000
        # Pick the first four dimensions, i.e. 1, x, x^2, x^3 from the polynomial
        # features
        fit_and_plot(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
                     labels[:n_train], labels[n_train:])
```

```
final epoch: train loss 0.0045331516 test loss 0.005139292
weight: [[ 4.995281   1.2209135 -3.3927279  5.5615673]]
```

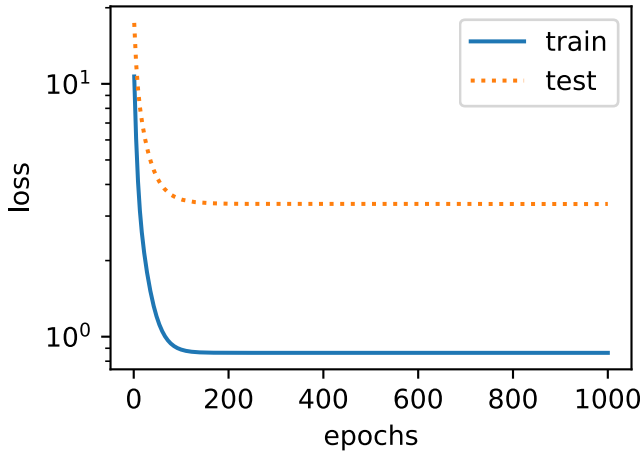


Linear Function Fitting (Underfitting)

Let's take another look at linear function fitting. Naturally, after the decline in the early epoch, it's difficult to further decrease this model's training error rate. After the last epoch iteration has been completed, the training error rate is still high. When used in data sets generated by non-linear models (like the third-order polynomial function) linear models are susceptible to underfitting.

```
In [7]: num_epochs = 1000
        # Pick the first four dimensions, i.e. 1, x from the polynomial features
        fit_and_plot(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
                     labels[:n_train], labels[n_train:])
```

```
final epoch: train loss 0.8632305 test loss 3.3497398
weight: [[ 4.9264355  3.378852  -2.6734383]]
```



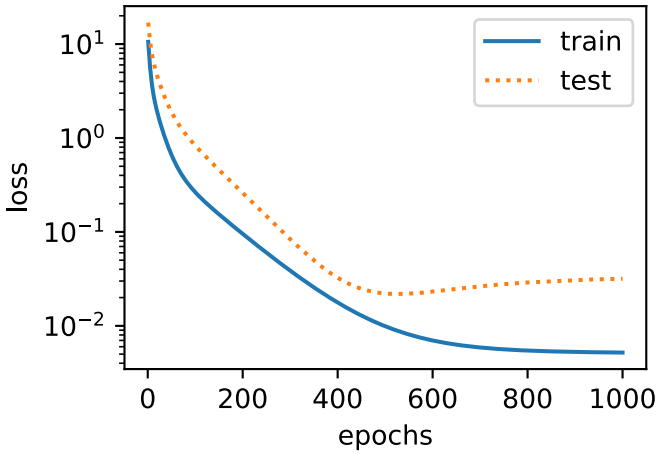
Insufficient Training (Overfitting)

In practice, if the model hasn't been trained sufficiently, it is still easy to overfit even if a third-order polynomial function with the same order as the data generation model is used. Let's train the model using a polynomial of too high degree. There is insufficient data to pin down the fact that all higher degree coefficients are close to zero. This will result in a model that's too complex to be easily influenced by noise in the training data. Even if the training error rate is low, the testing error data rate will still be high.

Try out different model complexities (`n_degree`) and training set sizes (`n_subset`) to gain some intuition of what is happening.

```
In [8]: num_epochs = 1000
        n_subset = 100  # Subset of data to train on
        n_degree = 20   # Degree of polynomials
        fit_and_plot(poly_features[1:n_subset, 0:n_degree],
```

```
                    poly_features[n_train:, 0:n_degree], labels[1:n_subset],
                    labels[n_train:])
final epoch: train loss 0.0051957457 test loss 0.031716917
weight: [[ 4.9595509e+00  1.2579050e+00 -3.2018919e+00  5.2782149e+00
  -6.3598323e-01  1.3321426e+00 -1.6340528e-02  2.0418510e-01
  -6.1858200e-02  5.8488246e-02 -3.7162870e-02 -6.7995586e-02
   3.7113789e-02 -1.9592594e-02  6.2177788e-02  3.2198682e-02
   3.4999892e-02 -4.5971844e-02 -2.2483468e-02  2.9451251e-03]]
```



Further along in later chapters, we will continue discussing overfitting problems and methods for dealing with them, such as weight decay and dropout.

6.11.5 Summary

- Since the generalization error rate cannot be estimated based on the training error rate, simply minimizing the training error rate will not necessarily mean a reduction in the generalization error rate. Machine learning models need to be careful to safeguard against overfitting such as to minimize the generalization error.
- A validation set can be used for model selection (provided that it isn't used too liberally).
- Underfitting means that the model is not able to reduce the training error rate while overfitting is a result of the model training error rate being much lower than the testing data set rate.
- We should choose an appropriately complex model and avoid using insufficient training samples.

6.11.6 Problems

1. Can you solve the polynomial regression problem exactly? Hint - use linear algebra.
2. Model selection for polynomials
   - Plot the training error vs. model complexity (degree of the polynomial). What do you observe?
   - Plot the test error in this case.
   - Generate the same graph as a function of the amount of data?
3. What happens if you drop the normalization of the polynomial features x^i by $1/i!$. Can you fix this in some other way?
4. What degree of polynomial do you need to reduce the training error to 0?
5. Can you ever expect to see 0 generalization error?

6.11.7 Scan the QR Code to Discuss

6.12 Weight Decay

In the previous section, we encountered overfitting and the need for capacity control. While increasing the training data set may mitigate overfitting, obtaining additional training data is often costly, hence it is preferable to control the complexity of the functions we use. In particular, we saw that we could control the complexity of a polynomial by adjusting its degree. While this might be a fine strategy for problems on one-dimensional data, this quickly becomes difficult to manage and too coarse. For instance, for vectors of dimensionality D the number of monomials of a given degree d is $\binom{D-1+d}{D-1}$. Hence, instead of controlling for the number of functions we need a more fine-grained tool for adjusting function complexity.

6.12.1 Squared Norm Regularization

One of the most commonly used techniques is weight decay. It relies on the notion that among all functions f the function $f = 0$ is the simplest of all. Hence we can measure functions by their proximity to zero. There are many ways of doing this. In fact there exist entire branches

of mathematics, e.g. in functional analysis and the theory of Banach spaces which are devoted to answering this issue.

For our purpose something much simpler will suffice: A linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ can be considered simple if its weight vector is small. We can measure this via $\|\mathbf{w}\|^2$. One way of keeping the weight vector small is to add its value as a penalty to the problem of minimizing the loss. This way if the weight vector becomes too large, the learning algorithm will prioritize minimizing $\mathbf{w}$ over minimizing the training error. That's exactly what we want. To illustrate things in code, consider the previous section on *"Linear Regression"*. There the loss is given by

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

Recall that $\mathbf{x}^{(i)}$ are the observations, $y^{(i)}$ are labels, and $(\mathbf{w}, b)$ are the weight and bias parameters respectively. To arrive at the new loss function which penalizes the size of the weight vector we need to add $\|\mathbf{w}\|^2$, but how much should we add? This is where the regularization constant (hyperparameter) λ comes in:

$$l(\mathbf{w}, b) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

$\lambda \geq 0$ governs the amount of regularization. For $\lambda = 0$ we recover the previous loss function, whereas for $\lambda > 0$ we ensure that $\mathbf{w}$ cannot grow too large. The astute reader might wonder why we are squaring the weight vector. This is done both for computational convenience since it leads to easy to compute derivatives, and for statistical performance, as it penalizes large weight vectors a lot more than small ones. The stochastic gradient descent updates look as follows:

$$w \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right)\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right),$$

As before, we update $\mathbf{w}$ in accordance to the amount to which our estimate differs from the observation. However, we also shrink the size of $\mathbf{w}$ towards 0, i.e. the weight 'decays'. This is much more convenient than having to pick the number of parameters as we did for polynomials. In particular, we now have a continuous mechanism for adjusting the complexity of f. Small values of λ correspond to fairly unconstrained $\mathbf{w}$ whereas large values of λ constrain $\mathbf{w}$ considerably. Since we don't want to have large bias terms either, we often add b^2 as penalty, too.

6.12.2 High-dimensional Linear Regression

For high-dimensional regression it is difficult to pick the 'right' dimensions to omit. Weight-decay regularization is a much more convenient alternative. We will illustrate this below. But

first we need to generate some data via

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01)$$

That is, we have additive Gaussian noise with zero mean and variance 0.01. In order to observe overfitting more easily we pick a high-dimensional problem with $d = 200$ and a deliberatly low number of training examples, e.g. 20. As before we begin with our import ritual (and data generation).

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn

        n_train, n_test, num_inputs = 20, 100, 200
        true_w, true_b = nd.ones((num_inputs, 1)) * 0.01, 0.05

        features = nd.random.normal(shape=(n_train + n_test, num_inputs))
        labels = nd.dot(features, true_w) + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)
        train_features, test_features = features[:n_train, :], features[n_train:, :]
        train_labels, test_labels = labels[:n_train], labels[n_train:]
```

6.12.3 Implementation from Scratch

Next, we will show how to implement weight decay from scratch. For this we simply add the ℓ_2 penalty as an additional loss term after the target function. The squared norm penalty derives its name from the fact that we are adding the second power $\sum_i x_i^2$. There are many other related penalties. In particular, the ℓ_p norm is defined as

$$\|\mathbf{x}\|_p^p := \sum_{i=1}^{d} |x_i|^p$$

Initialize Model Parameters

First, define a function that randomly initializes model parameters. This function attaches a gradient to each parameter.

```
In [2]: def init_params():
            w = nd.random.normal(scale=1, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            w.attach_grad()
            b.attach_grad()
            return [w, b]
```

Define ℓ_2 Norm Penalty

A convenient way of defining this penalty is by squaring all terms in place and summing them up. We divide by 2 to keep the math looking nice and simple.

```
In [3]: def l2_penalty(w):
            return (w**2).sum() / 2
```

Define Training and Testing

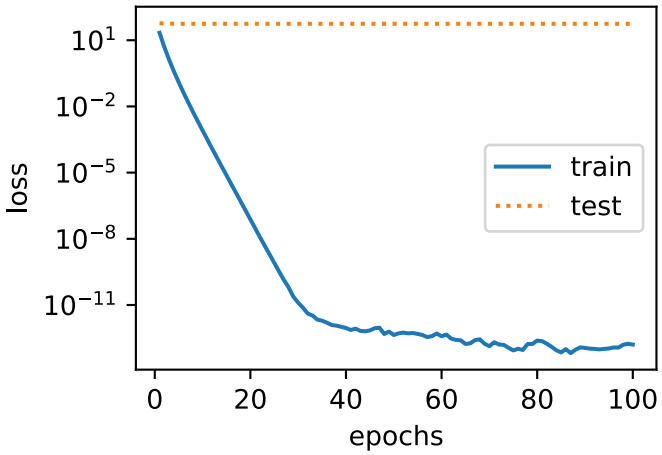
The following defines how to train and test the model separately on the training data set and the test data set. Unlike the previous sections, here, the ℓ_2 norm penalty term is added when calculating the final loss function. The linear network and the squared loss are as before and thus imported via `d2l.linreg` and `d2l.squared_loss` respectively.

```
In [4]: batch_size, num_epochs, lr = 1, 100, 0.003
        net, loss = d2l.linreg, d2l.squared_loss
        train_iter = gdata.DataLoader(gdata.ArrayDataset(
            train_features, train_labels), batch_size, shuffle=True)

        def fit_and_plot(lambd):
            w, b = init_params()
            train_ls, test_ls = [], []
            for _ in range(num_epochs):
                for X, y in train_iter:
                    with autograd.record():
                        # The L2 norm penalty term has been added
                        l = loss(net(X, w, b), y) + lambd * l2_penalty(w)
                    l.backward()
                    d2l.sgd([w, b], lr, batch_size)
                train_ls.append(loss(net(train_features, w, b),
                                     train_labels).mean().asscalar())
                test_ls.append(loss(net(test_features, w, b),
                                    test_labels).mean().asscalar())
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                         range(1, num_epochs + 1), test_ls, ['train', 'test'])
            print('l2 norm of w:', w.norm().asscalar())
```

Training without Regularization

Next, let's train and test the high-dimensional linear regression model. When `lambd = 0` we do not use weight decay. As a result, while the training error decreases, the test error does not. This is a perfect example of overfitting.

```
In [5]: fit_and_plot(lambd=0)
```

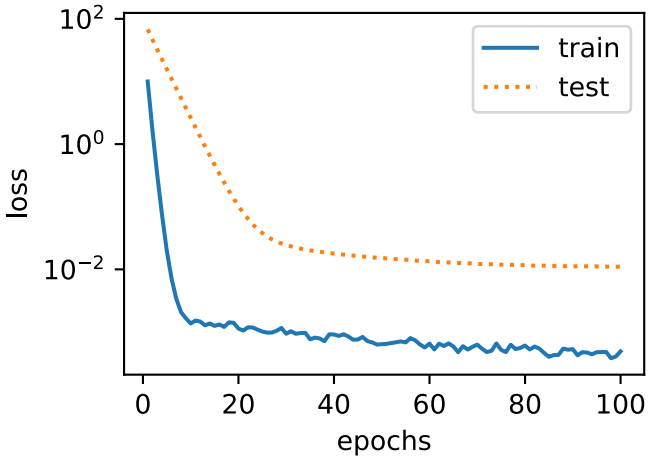



```
l2 norm of w: 11.611941
```

Using Weight Decay

The example below shows that even though the training error increased, the error on the test set decreased. This is precisely the improvement that we expect from using weight decay. While not perfect, overfitting has been mitigated to some extent. In addition, the ℓ_2 norm of the weight $\mathbf{w}$ is smaller than without using weight decay.

```
In [6]: fit_and_plot(lambd=3)
```




```
l2 norm of w: 0.04024435
```

6.12.4 Concise Implementation

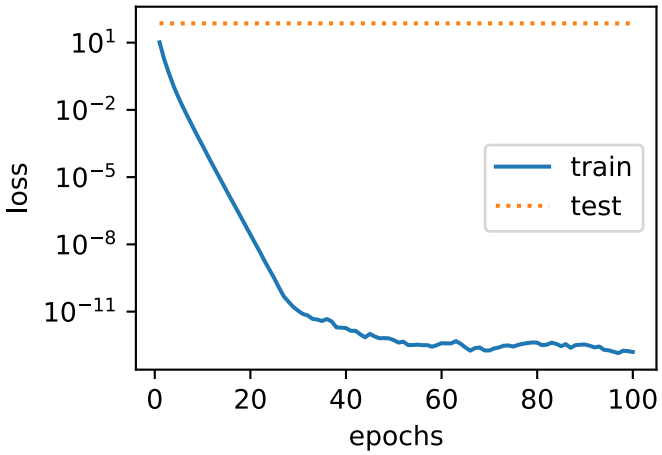
Weight decay in Gluon is quite convenient (and also a bit special) insofar as it is typically integrated with the optimization algorithm itself. The reason for this is that it is much faster (in terms of runtime) for the optimizer to take care of weight decay and related things right inside the optimization algorithm itself, since the optimizer itself needs to touch all parameters anyway.

Here, we directly specify the weight decay hyper-parameter through the `wd` parameter when constructing the `Trainer` instance. By default, Gluon decays weight and bias simultaneously. Note that we can have *different* optimizers for different sets of parameters. For instance, we can have a `Trainer` with weight decay and one without to take care of $\mathbf{w}$ and b respectively.

```
In [7]: def fit_and_plot_gluon(wd):
            net = nn.Sequential()
            net.add(nn.Dense(1))
            net.initialize(init.Normal(sigma=1))
            # The weight parameter has been decayed. Weight names generally end with
            # "weight".
            trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                                      {'learning_rate': lr, 'wd': wd})
            # The bias parameter has not decayed. Bias names generally end with "bias"
            trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                                      {'learning_rate': lr})
            train_ls, test_ls = [], []
            for _ in range(num_epochs):
                for X, y in train_iter:
                    with autograd.record():
                        l = loss(net(X), y)
                    l.backward()
                    # Call the step function on each of the two Trainer instances to
                    # update the weight and bias separately
                    trainer_w.step(batch_size)
                    trainer_b.step(batch_size)
                train_ls.append(loss(net(train_features),
                                     train_labels).mean().asscalar())
                test_ls.append(loss(net(test_features),
                                    test_labels).mean().asscalar())
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                         range(1, num_epochs + 1), test_ls, ['train', 'test'])
            print('L2 norm of w:', net[0].weight.data().norm().asscalar())
```

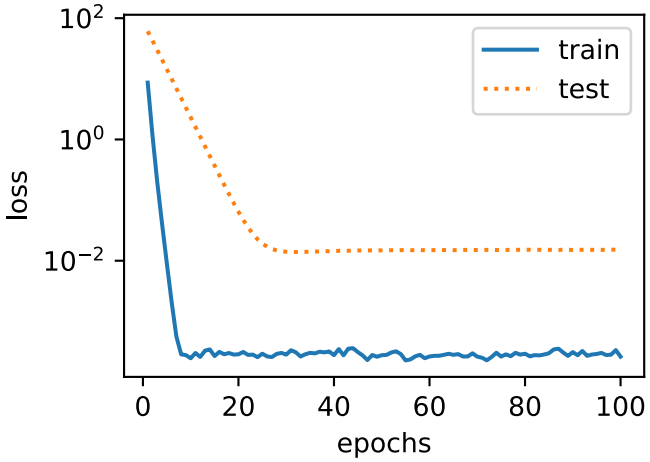
The plots look just the same as when we implemented weight decay from scratch (but they run a bit faster and are a bit easier to implement, in particular for large problems).

```
In [8]: fit_and_plot_gluon(0)
```




```
L2 norm of w: 13.311798
```

```
In [9]: fit_and_plot_gluon(3)
```




```
L2 norm of w: 0.033074923
```

So far we only touched upon what constitutes a simple *linear* function. For nonlinear functions answering this question is way more complex. For instance, there exist Reproducing Kernel Hilbert Spaces which allow one to use many of the tools introduced for linear functions in a nonlinear context. Unfortunately, algorithms using them do not always scale well to very large amounts of data. For all intents and purposes of this book we limit ourselves to simply summing over the weights for different layers, e.g. via $\sum_l \|\mathbf{w}_l\|^2$, which is equivalent to weight decay applied to all layers.

6.12.5 Summary

- Regularization is a common method for dealing with overfitting. It adds a penalty term to the loss function on the training set to reduce the complexity of the learned model.
- One particular choice for keeping the model simple is weight decay using an ℓ_2 penalty. This leads to weight decay in the update steps of the learning algorithm.
- Gluon provides automatic weight decay functionality in the optimizer by setting the hyperparameter `wd`.
- You can have different optimizers within the same training loop, e.g. for different sets of parameters.

6.12.6 Problems

1. Experiment with the value of λ in the estimation problem in this page. Plot training and test accuracy as a function of λ. What do you observe?
2. Use a validation set to find the optimal value of λ. Is it really the optimal value? Does this matter?
3. What would the update equations look like if instead of $\|\mathbf{w}\|^2$ we used $\sum_i |w_i|$ as our penalty of choice (this is called ℓ_1 regularization).
4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (mathematicians call this the Frobenius norm)?
5. Review the relationship between training error and generalization error. In addition to weight decay, increased training, and the use of a model of suitable complexity, what other ways can you think of to deal with overfitting?
6. In Bayesian statistics we use the product of prior and likelihood to arrive at a posterior via $p(w|x) \propto p(x|w)p(w)$. How can you identify $p(w)$ with regularization?

6.12.7 Scan the QR Code to Discuss

6.13 Dropout

In the previous chapter, we introduced one classical approach to regularize statistical models. We penalized the size (the ℓ_2 norm) of the weights, coercing them to take smaller values. In probabilistic terms we might say that this imposes a Gaussian prior on the value of the weights. But in more intuitive, functional terms, we can say that this encourages the model to spread out its weights among many features and not to depend too much on a small number of potentially spurious associations.

6.13.1 Overfitting Revisited

With great flexibility comes overfitting liability. Given many more features than examples, linear models can overfit. But when there are many more examples than features, linear models can usually be counted on not to overfit. Unfortunately this propensity to generalize well comes at a cost. For every feature, a linear model has to assign it either positive or negative weight. Linear models can't take into account nuanced interactions between features. In more formal texts, you'll see this phenomena discussed as the bias-variance tradeoff. Linear models have high bias, (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data).

Deep neural networks, however, occupy the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, they can learn complex interactions among groups of features. For example, they might infer that "Nigeria"and "Western Union"appearing together in an email indicates spam but that "Nigeria"without "Western Union"does not connote spam.

Even for a small number of features, deep neural networks are capable of overfitting. As one demonstration of the incredible flexibility of neural networks, researchers showed that neural networks perfectly classify randomly labeled data. Let's think about what means. If the labels are assigned uniformly at random, and there are 10 classes, then no classifier can get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

6.13.2 Robustness through Perturbations

Let's think briefly about what we expect from a good statistical model. Obviously we want it to do well on unseen test data. One way we can accomplish this is by asking for what amounts to a 'simple' model. Simplicity can come in the form of a small number of dimensions, which is what we did when discussing fitting a function with monomial basis functions. Simplicity can also come in the form of a small norm for the basis funtions. This is what led to weight decay and ℓ_2 regularization. Yet a third way to impose some notion of simplicity is that the function should be robust under modest changes in the input. For instance, when we classify images, we would expect that alterations of a few pixels are mostly harmless.

In fact, this notion was formalized by Bishop in 1995, when he proved that Training with Input Noise is Equivalent to Tikhonov Regularization. That is, he connected the notion of having a smooth (and thus simple) function with one that is resilient to perturbations in the input. Fast forward to 2014. Given the complexity of deep networks with many layers, enforcing smoothness just on the input misses out on what is happening in subsequent layers. The ingenious idea of Srivastava et al., 2014 was to apply Bishop's idea to the *internal* layers of the network, too, namely to inject noise into the computational path of the network while it's training.

A key challenge in this context is how to add noise without introducing undue bias. In terms of inputs $\mathbf{x}$, this is relatively easy to accomplish: simply add some noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to it and use this data during training via $\mathbf{x}' = \mathbf{x} + \epsilon$. A key property is that in expectation $\mathbf{E}[\mathbf{x}'] = \mathbf{x}$. For intermediate layers, though, this might not be quite so desirable since the scale of the noise might not be appropriate. The alternative is to perturb coordinates as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

By design, the expectation remains unchanged, i.e. $\mathbf{E}[h'] = h$. This idea is at the heart of dropout where intermediate activations h are replaced by a random variable h' with matching expectation. The name 'dropout' arises from the notion that some neurons 'drop out' of the computation for the purpose of computing the final result. During training we replace intermediate activations with random variables

6.13.3 Dropout in Practice

Recall the *multilayer perceptron* with a hidden layer and 5 hidden units. Its architecture is given by

$$\begin{aligned} h &= \sigma(W_1 x + b_1) \\ o &= W_2 h + b_2 \\ \hat{y} &= \text{softmax}(o) \end{aligned}$$

When we apply dropout to the hidden layer, it amounts to removing hidden units with probability p since their output is set to 0 with that probability. A possible result is the network shown below. Here h_2 and h_5 are removed. Consequently the calculation of y no longer depends on h_2 and h_5 and their respective gradient also vanishes when performing backprop. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$. This is exactly what we want for regularization purposes to cope with overfitting. At test time we typically do not use dropout to obtain more conclusive results.

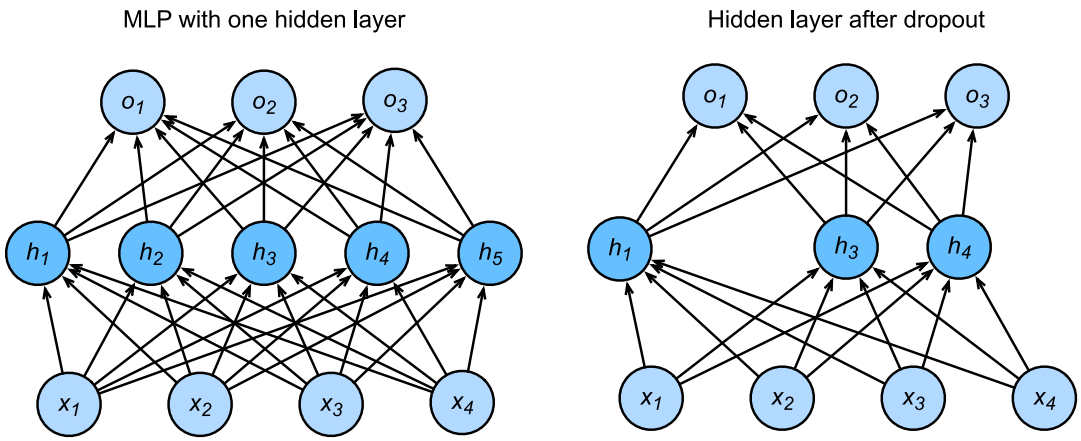


Fig. 6.9: MLP before and after dropout

6.13.4 Implementation from Scratch

To implement the dropout function we have to draw as many random variables as the input has dimensions from the uniform distribution $U[0, 1]$. According to the definition of dropout, we can implement it easily. The following `dropout` function will drop out the elements in the NDArray input `X` with the probability of `drop_prob`.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn

        def dropout(X, drop_prob):
            assert 0 <= drop_prob <= 1
            # In this case, all elements are dropped out
            if drop_prob == 1:
                return X.zeros_like()
            mask = nd.random.uniform(0, 1, X.shape) > drop_prob
            return mask * X / (1.0-drop_prob)
```

Let us test how it works in a few examples. The dropout probability is 0, 0.5, and 1, respectively.

```
In [2]: X = nd.arange(16).reshape((2, 8))
        print(dropout(X, 0))
        print(dropout(X, 0.5))
        print(dropout(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
<NDArray 2x8 @cpu(0)>
```

```
[[ 0.  0.  0.  0.  8. 10. 12.  0.]
 [16.  0. 20. 22.  0.  0.  0. 30.]]
<NDArray 2x8 @cpu(0)>

[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
<NDArray 2x8 @cpu(0)>
```

Defining Model Parameters

Let's use the same dataset as used previously, namely Fashion-MNIST, described in the section "*Softmax Regression - Starting From Scratch*". We will define a multilayer perceptron with two hidden layers. The two hidden layers both have 256 outputs.

```
In [3]: num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

        W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens1))
        b1 = nd.zeros(num_hiddens1)
        W2 = nd.random.normal(scale=0.01, shape=(num_hiddens1, num_hiddens2))
        b2 = nd.zeros(num_hiddens2)
        W3 = nd.random.normal(scale=0.01, shape=(num_hiddens2, num_outputs))
        b3 = nd.zeros(num_outputs)

        params = [W1, b1, W2, b2, W3, b3]
        for param in params:
            param.attach_grad()
```

Define the Model

The model defined below concatenates the fully connected layer and the activation function ReLU, using dropout for the output of each activation function. We can set the dropout probability of each layer separately. It is generally recommended to set a lower dropout probability closer to the input layer. Below we set it to 0.2 and 0.5 for the first and second hidden layer respectively. By using the `is_training` function described in the "Autograd" section we can ensure that dropout is only active during training.

```
In [4]: drop_prob1, drop_prob2 = 0.2, 0.5

        def net(X):
            X = X.reshape((-1, num_inputs))
            H1 = (nd.dot(X, W1) + b1).relu()
            # Use dropout only when training the model
            if autograd.is_training():
                # Add a dropout layer after the first fully connected layer
                H1 = dropout(H1, drop_prob1)
            H2 = (nd.dot(H1, W2) + b2).relu()
            if autograd.is_training():
                # Add a dropout layer after the second fully connected layer
                H2 = dropout(H2, drop_prob2)
            return nd.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of multilayer perceptrons described previously.

```
In [5]: num_epochs, lr, batch_size = 10, 0.5, 256
        loss = gloss.SoftmaxCrossEntropyLoss()
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                      params, lr)
```

```
epoch 1, loss 1.1203, train acc 0.572, test acc 0.790
epoch 2, loss 0.5875, train acc 0.783, test acc 0.835
epoch 3, loss 0.4874, train acc 0.824, test acc 0.849
epoch 4, loss 0.4442, train acc 0.838, test acc 0.854
epoch 5, loss 0.4175, train acc 0.848, test acc 0.859
epoch 6, loss 0.3956, train acc 0.857, test acc 0.866
epoch 7, loss 0.3802, train acc 0.861, test acc 0.866
epoch 8, loss 0.3697, train acc 0.865, test acc 0.876
epoch 9, loss 0.3537, train acc 0.870, test acc 0.874
epoch 10, loss 0.3410, train acc 0.875, test acc 0.881
```

6.13.5 Concise Implementation

In Gluon, we only need to add the `Dropout` layer after the fully connected layer and specify the dropout probability. When training the model, the `Dropout` layer will randomly drop out the output elements of the previous layer at the specified dropout probability; the `Dropout` layer simply passes the data through during testing.

```
In [6]: net = nn.Sequential()
        net.add(nn.Dense(256, activation="relu"),
                # Add a dropout layer after the first fully connected layer
                nn.Dropout(drop_prob1),
                nn.Dense(256, activation="relu"),
                # Add a dropout layer after the second fully connected layer
                nn.Dropout(drop_prob2),
                nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

Next, we will train and test the model.

```
In [7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)
```

```
epoch 1, loss 1.2600, train acc 0.521, test acc 0.750
epoch 2, loss 0.6061, train acc 0.776, test acc 0.827
epoch 3, loss 0.5165, train acc 0.813, test acc 0.846
epoch 4, loss 0.4673, train acc 0.832, test acc 0.847
epoch 5, loss 0.4371, train acc 0.839, test acc 0.859
epoch 6, loss 0.4138, train acc 0.851, test acc 0.865
epoch 7, loss 0.3959, train acc 0.854, test acc 0.869
epoch 8, loss 0.3852, train acc 0.859, test acc 0.876
epoch 9, loss 0.3713, train acc 0.864, test acc 0.870
epoch 10, loss 0.3598, train acc 0.870, test acc 0.879
```

6.13.6 Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often all three are used jointly.
- Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p.
- Dropout is only used during training.

6.13.7 Problems

1. Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. Compute the variance of the the activation random variables after applying dropout.
4. Why should you typically not using dropout?
5. If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?
6. Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?
7. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
8. Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?

6.13.8 References

[1] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). JMLR

6.13.9 Scan the QR Code to Discuss

6.14 Forward Propagation, Back Propagation, and Computational Graphs

In the previous sections we used a mini-batch stochastic gradient descent optimization algorithm to train the model. During the implementation of the algorithm, we only calculated the forward propagation of the model, which is to say, we calculated the model output for the input, then called the auto-generated `backward` function to then finally calculate the gradient through the `autograd` module. The automatic gradient calculation, when based on back-propagation, significantly simplifies the implementation of the deep learning model training algorithm. In this section we will use both mathematical and computational graphs to describe forward and back propagation. More specifically, we will explain forward and back propagation through a sample model with a single hidden layer perceptron with ℓ_2 norm regularization. This section will help understand a bit better what goes on behind the scenes when we invoke a deep network.

6.14.1 Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network within the models in the order from input layer to output layer. In the following we work in detail through the example of a deep network with one hidden layer step by step. This is a bit tedious but it will serve us well when discussing what really goes on when we call `backward`.

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and there is no bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$$

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After entering the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ into the activation function ϕ operated by the basic elements, we will obtain a hidden layer variable with the vector length of h,

$$\mathbf{h} = \phi(\mathbf{z}).$$

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming the parameters of the output layer only possess a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector length of q,

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

Assuming the loss function is l and the example label is y, we can then calculate the loss term

for a single data example,

$$L = l(\mathbf{o}, y).$$

According to the definition of ℓ_2 norm regularization, given the hyper-parameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right),$$

where the Frobenius norm of the matrix is equivalent to the calculation of the L_2 norm after flattening the matrix to a vector. Finally, the model's regularized loss on a given data example is

$$J = L + s.$$

We refer to J as the objective function of a given data example and refer to it as the 'objective function' in the following discussion.

6.14.2 Computational Graph of Forward Propagation

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. The figure below contains the graph associated with the simple network described above. The lower left corner signifies the input and the upper right corner the output. Notice that the direction of the arrows (which illustrate data flow) are primarily rightward and upward.

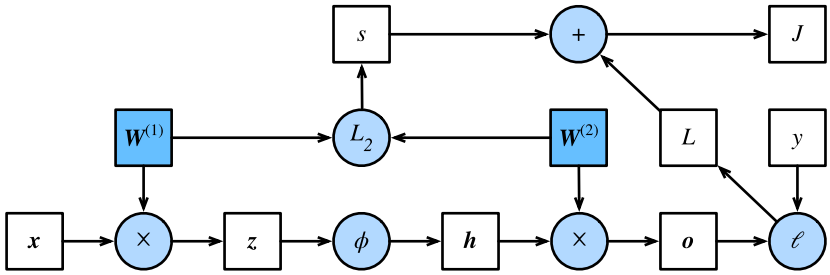


Fig. 6.10: Compute Graph

6.14.3 Back Propagation

Back propagation refers to the method of calculating the gradient of neural network parameters. In general, back propagation calculates and stores the intermediate variables of an objective function related to each layer of the neural network and the gradient of the parameters in the order of the output layer to the input layer according to the 'chain rule' in calculus. Assume

that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y}) = g \circ f(\mathsf{X})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule we can compute the derivative of Z wrt. X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right).$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions have been carried out. For vectors this is straightforward: it is simply matrix-matrix multiplication and for higher dimensional tensors we use the appropriate counterpart. The operator prod hides all the notation overhead.

The parameters of the simple network with one hidden layer are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of back propagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accompish this we will apply the chain rule and calculate in turn the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the compute graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1$$

Next we compute the gradient of the objective function with respect to variable of the output layer $\mathbf{o}$ according to the chain rule.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$$

Next we calculate the gradients of the regularization term with respect to both parameters.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$$

Now we are able calculate the gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue back propagation along the output layer to the hidden layer. The gradient $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ of the hidden layer variable is

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}.$$

Since the activation function ϕ applies element-wise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}$ requires the use of the element-wise multiplication operator. We

denote it by $\odot$.

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right).$$

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

6.14.4 Training a Model

When training networks, forward and backward propagation depend on each other. In particular, for forward propagation we traverse the compute graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why backpropagation requires significantly more memory than plain 'inference' - we end up computing tensors as gradients and need to retain all the intermediate variables to invoke the chain rule. Another reason is that we typically train with minibatches containing more than one variable, thus more intermediate activations need to be stored.

6.14.5 Summary

- Forward propagation sequentially calculates and stores intermediate variables within the compute graph defined by the neural network. It proceeds from input to output layer.
- Back propagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- When training deep learning models, forward propagation and back propagation are interdependent.
- Training requires significantly more memory and storage.

6.14.6 Problems

1. Assume that the inputs $\mathbf{x}$ are matrices. What is the dimensionality of the gradients?
2. Add a bias to the hidden layer of the model described in this chapter.
    - Draw the corresponding compute graph.
    - Derive the forward and backward propagation equations.

3. Compute the memory footprint for training and inference in model described in the current chapter.
4. Assume that you want to compute *second* derivatives. What happens to the compute graph? Is this a good idea?
5. Assume that the compute graph is too large for your GPU.
   - Can you partition it over more than one GPU?
   - What are the advantages and disadvantages over training on a smaller minibatch?

6.14.7 Scan the QR Code to Discuss

6.15 Numerical Stability and Initialization

So far we covered the tools needed to implement multilayer perceptrons, how to solve regression and classification problems, and how to control capacity. However, we took initialization of the parameters for granted, or rather simply assumed that they would not be particularly relevant. In the following we will look at them in more detail and discuss some useful heuristics.

Secondly, we were not particularly concerned with the choice of activation. Indeed, for shallow networks this is not very relevant. For deep networks, however, design choices of nonlinearity and initialization play a crucial role in making the optimization algorithm converge relatively rapidly. Failure to be mindful of these issues can lead to either exploding or vanishing gradients.

6.15.1 Vanishing and Exploding Gradients

Consider a deep network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$. Each layer satisfies:

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{ and thus } \mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$$

If all activations and inputs are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ associated with the function f_t at layer t simply as

$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:=\mathbf{v}_t}.$$

In other words, it is the product of $d-t$ matrices $\mathbf{M}_d \cdot \ldots \cdot \mathbf{M}_t$ and the gradient vector $\mathbf{v}_t$. What happens is quite similar to the situation when we experienced numerical underflow when multiplying too many probabilities. At the time we were able to mitigate the problem by switching from into log-space, i.e. by shifting the problem from the mantissa to the exponent of the numerical representation. Unfortunately the problem outlined in the equation above is much more serious: initially the matrices M_t may well have a wide variety of eigenvalues. They might be small, they might be large, and in particular, their product might well be *very large* or *very small*. This is not (only) a problem of numerical representation but it means that the optimization algorithm is bound to fail. It either receives gradients with excessively large or excessively small steps. In the former case, the parameters explode and in the latter case we end up with vanishing gradients and no meaningful progress.

Exploding Gradients

To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scaling that we picked, the matrix product explodes. If this were to happen to us with a deep network, we would have no meaningful chance of making the algorithm converge.

```
In [1]: %matplotlib inline
        import mxnet as mx
        from mxnet import nd, autograd
        from matplotlib import pyplot as plt

        M = nd.random.normal(shape=(4,4))
        print('A single matrix', M)
        for i in range(100):
            M = nd.dot(M, nd.random.normal(shape=(4,4)))

        print('After multiplying 100 matrices', M)
```

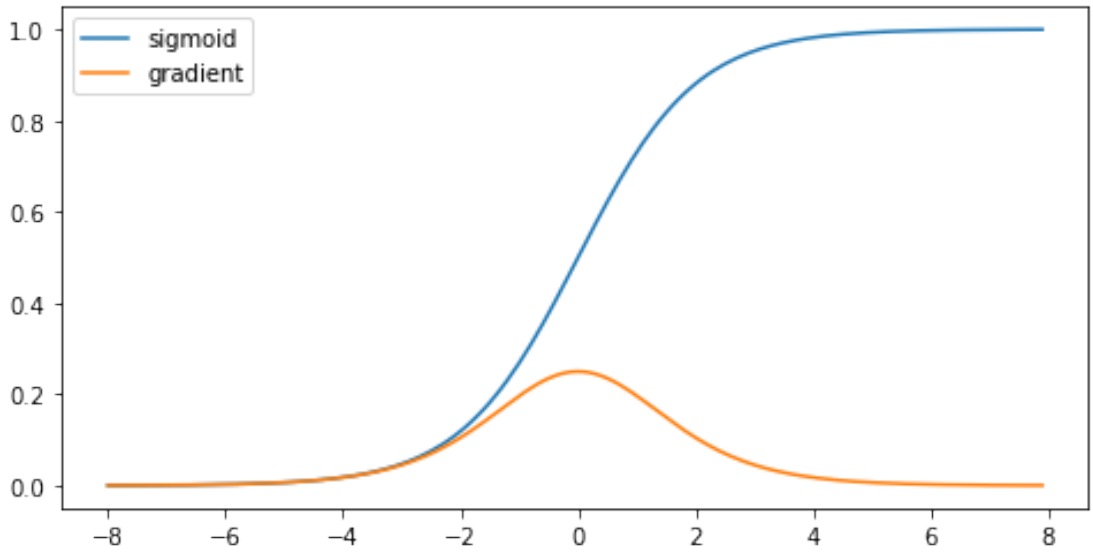
```
A single matrix
[[ 2.2122064   0.7740038   1.0434405   1.1839255 ]
 [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
<NDArray 4x4 @cpu(0)>
After multiplying 100 matrices
[[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
 [-5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20]
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19]]
<NDArray 4x4 @cpu(0)>
```

Vanishing Gradients

The converse problem of vanishing gradients is just as bad. One of the major culprits in this context are the activation functions σ that are interleaved with the linear operations in each layer. Historically, a popular activation used to be the sigmoid function $(1+\exp(-x))$ that was

introduced in the section discussing *Multilayer Perceptrons*. Let us briefly review the function to see why picking it as nonlinear activation function might be problematic.

```
In [2]: x = nd.arange(-8.0, 8.0, 0.1)
        x.attach_grad()
        with autograd.record():
            y = x.sigmoid()
        y.backward()

        plt.figure(figsize=(8, 4))
        plt.plot(x.asnumpy(), y.asnumpy())
        plt.plot(x.asnumpy(), x.grad.asnumpy())
        plt.legend(['sigmoid', 'gradient'])
        plt.show()
```



As we can see, the gradient of the sigmoid vanishes for very large or very small arguments. Due to the chain rule, this means that unless we are in the Goldilocks zone where the activations are in the range of, say $[-4, 4]$, the gradients of the overall product may vanish. When we have many layers this is likely to happen for *some* layer. Before ReLu $\max(0, x)$ was proposed, this problem used to be the bane of deep network training. As a consequence ReLu has become the default choice when designing activation functions in deep networks.

Symmetry

A last problem in deep network design is the symmetry inherent in their parametrization. Assume that we have a deep network with one hidden layer with two units, say h_1 and h_2. In this case, we could flip the weights $\mathbf{W}_1$ of the first layer and likewise the outputs of the second layer and we would obtain the same function. More generally, we have permutation symmetry between the hidden units of each layer. This is more than just a theoretical nuisance. Assume

that we initialize the parameters of some layer as $\mathbf{W}_l = 0$ or even just assume that all entries of $\mathbf{W}_l$ are identical. In this case the gradients for all dimensions are identical and we will never be able to use the expressive power inherent in a given layer. In fact, the hidden layer behaves as if it had only a single unit.

6.15.2 Parameter Initialization

One way of addressing, or at least mitigating the issues raised above is through careful initialization of the weight vectors. This way we can ensure that at least initially the gradients do not vanish and that they are within a reasonable scale where the network weights do not diverge. Additional care during optimization and suitable regularization ensures that things never get too bad. Let's get started.

Default Initialization

In the previous sections, e.g. in *"Concise Implementation of Linear Regression"*, we used `net.initialize(init.Normal(sigma=0.01))` as a way to pick normally distributed random numbers as initial values for the weights. If the initialization method is not specified, such as `net.initialize()`, MXNet will use the default random initialization method: each element of the weight parameter is randomly sampled with an uniform distribution $U[-0.07, 0.07]$ and the bias parameters are all set to 0. Both choices tend to work quite well in practice for moderate problem sizes.

Xavier Initialization

Let's look at the scale distribution of the activations of the hidden units h_i for some layer. They are given by

$$h_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j$$

The weights W_{ij} are all drawn independently from the same distribution. Let's furthermore assume that this distribution has zero mean and variance σ^2 (this doesn't mean that the distribution has to be Gaussian, just that mean and variance need to exist). We don't really have much control over the inputs into the layer x_j but let's proceed with the somewhat unrealistic assumption that they also have zero mean and variance γ^2 and that they're independent of $\mathbf{W}$.

In this case we can compute mean and variance of h_i as follows:

$$\begin{aligned}
\mathbf{E}[h_i] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij} x_j] = 0 \\
\mathbf{E}[h_i^2] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2 x_j^2] \\
&= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2] \mathbf{E}[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2
\end{aligned}$$

One way to keep the variance fixed is to set $n_{\text{in}}\sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the top layers. That is, instead of $\mathbf{Ww}$ we need to deal with $\mathbf{W}^\top \mathbf{g}$, where $\mathbf{g}$ is the incoming gradient from the layer above. Using the same reasoning as for forward propagation we see that the gradients' variance can blow up unless $n_{\text{out}}\sigma^2 = 1$. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to satisfy

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

This is the reasoning underlying the eponymous Xavier initialization, proposed by Xavier Glorot and Yoshua Bengio in 2010. It works well enough in practice. For Gaussian random variables the Xavier initialization picks a normal distribution with zero mean and variance $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$. For uniformly distributed random variables $U[-a, a]$ note that their variance is given by $a^2/3$. Plugging $a^2/3$ into the condition on σ^2 yields that we should initialize uniformly with $U\left[-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})}\right]$.

Beyond

The reasoning above barely scratches the surface. In fact, MXNet has an entire `mxnet.initializer` module with over a dozen different heuristics. They can be used, e.g. when parameters are tied (i.e. when parameters of in different parts the network are shared), for superresolution, sequence models, and related problems. We recommend the reader to review what is offered as part of this module.

6.15.3 Summary

- Vanishing and exploding gradients are common issues in very deep networks, unless great care is taking to ensure that gradients and parameters remain well controlled.

- Initialization heuristics are needed to ensure that at least the initial gradients are neither too large nor too small.
- The ReLu addresses one of the vanishing gradient problems, namely that gradients vanish for very large inputs. This can accelerate convergence significantly.
- Random initialization is key to ensure that symmetry is broken before optimization.

6.15.4 Problems

1. Can you design other cases of symmetry breaking besides the permutation symmetry?
2. Can we initialize all weight parameters in linear regression or in softmax regression to the same value?
3. Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?
4. If we know that some terms diverge, can we fix this after the fact? Look at the paper on LARS by You, Gitman and Ginsburg, 2017 for inspiration.

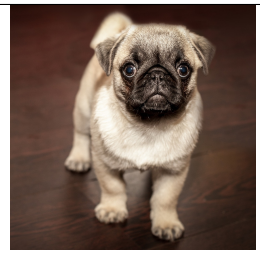
6.15.5 Scan the QR Code to Discuss

6.16 Environment

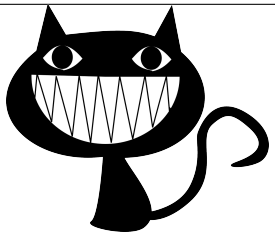
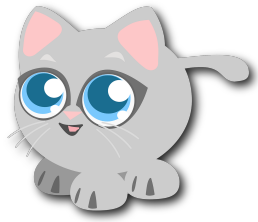
So far we did not worry very much about where the data came from and how the models that we build get deployed. Not caring about it can be problematic. Many failed machine learning deployments can be traced back to this situation. This chapter is meant to help with detecting such situations early and points out how to mitigate them. Depending on the case this might be rather simple (ask for the 'right' data) or really difficult (implement a reinforcement learning system).

6.16.1 Covariate Shift

At its heart is a problem that is easy to understand but also equally easy to miss. Consider being given the challenge of distinguishing cats and dogs. Our training data consists of images of the following kind:

cat	cat	dog	dog
			

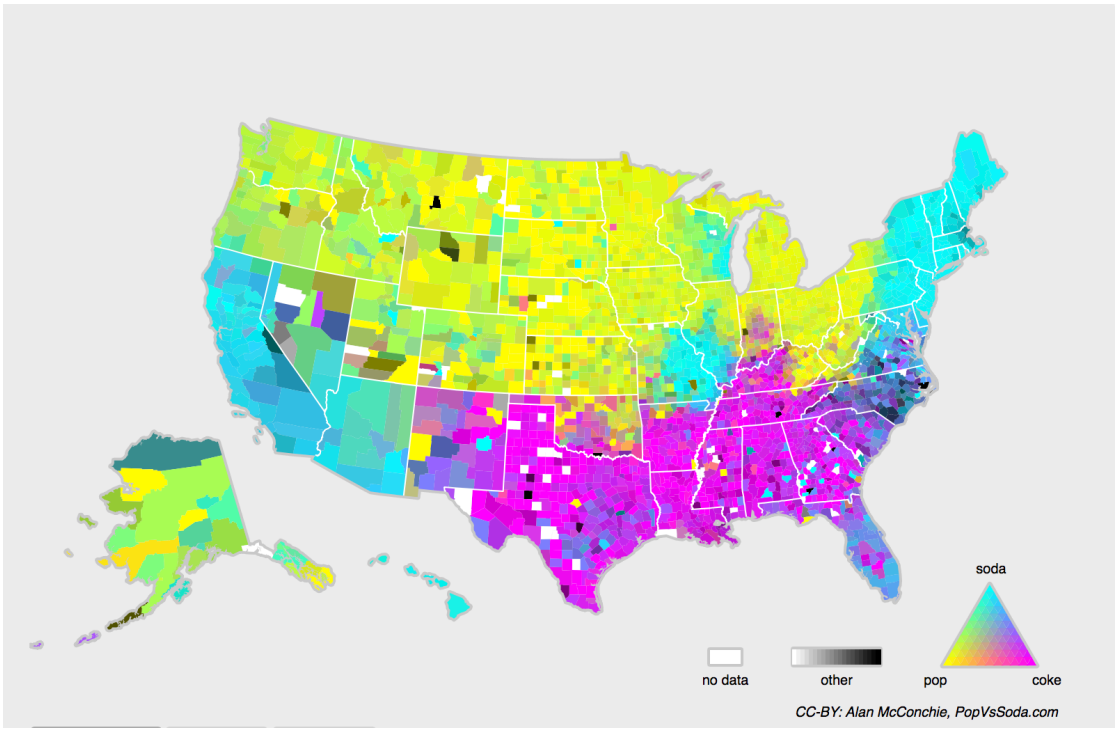
At test time we are asked to classify the following images:

cat	cat	dog	dog
			

Obviously this is unlikely to work well. The training set consists of photos, while the test set contains only cartoons. The colors aren't even accurate. Training on a dataset that looks substantially different from the test set without some plan for how to adapt to the new domain is a bad idea. Unfortunately, this is a very common pitfall. Statisticians call this **Covariate Shift**, i.e. the situation where the distribution over the covariates (aka training data) is shifted on test data relative to the training case. Mathematically speaking, we are referring the case where $p(x)$ changes but $p(y|x)$ remains unchanged.

6.16.2 Concept Shift

A related problem is that of concept shift. This is the situation where the labels change. This sounds weird - after all, a cat is a cat. Well, cats maybe but not soft drinks. There is considerable concept shift throughout the USA, even for such a simple term:



If we were to build a machine translation system, the distribution $p(y|x)$ would be different, e.g. depending on our location. This problem can be quite tricky to spot. A saving grace is that quite often the $p(y|x)$ only shifts gradually (e.g. the click-through rate for NOKIA phone ads). Before we go into further details, let us discuss a number of situations where covariate and concept shift are not quite as blatantly obvious.

6.16.3 Examples

Medical Diagnostics

Imagine you want to design some algorithm to detect cancer. You get data of healthy and sick people; you train your algorithm; it works fine, giving you high accuracy and you conclude that you're ready for a successful career in medical diagnostics. Not so fast ···

Many things could go wrong. In particular, the distributions that you work with for training and those in the wild might differ considerably. This happened to an unfortunate startup I had the opportunity to consult for many years ago. They were developing a blood test for a disease that affects mainly older men and they'd managed to obtain a fair amount of blood samples from patients. It is considerably more difficult, though, to obtain blood samples from healthy men (mainly for ethical reasons). To compensate for that, they asked a large number of students on campus to donate blood and they performed their test. Then they asked me whether

I could help them build a classifier to detect the disease. I told them that it would be very easy to distinguish between both datasets with probably near perfect accuracy. After all, the test subjects differed in age, hormone level, physical activity, diet, alcohol consumption, and many more factors unrelated to the disease. This was unlikely to be the case with real patients: Their sampling procedure had caused an extreme case of covariate shift that couldn't be corrected by conventional means. In other words, training and test data were so different that nothing useful could be done and they had wasted significant amounts of money.

Self Driving Cars

A company wanted to build a machine learning system for self-driving cars. One of the key components is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on 'test data' drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this 'feature' very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The so-trained classifier worked 'perfectly'. Unfortunately, all it had learned was to distinguish trees with shadows from trees without shadows - the first set of pictures was taken in the early morning, the second one at noon.

Nonstationary distributions

A much more subtle situation is where the distribution changes slowly and the model is not updated adequately. Here are a number of typical cases:

- We train a computational advertising model and then fail to update it frequently (e.g. we forget to incorporate that an obscure new device called an iPad was just launched).
- We build a spam filter. It works well at detecting all spam that we've seen so far. But then the spammers wisen up and craft new messages that look quite unlike anything we've seen before.
- We build a product recommendation system. It works well for the winter. But then it keeps on recommending Santa hats after Christmas.

More Anecdotes

- We build a classifier for "Not suitable/safe for work"(NSFW) images. To make our life easy, we scrape a few seedy Subreddits. Unfortunately the accuracy on real life data is lacking (the pictures posted on Reddit are mostly 'remarkable' in some way, e.g. being

taken by skilled photographers, whereas most real NSFW images are fairly unremarkable ···). Quite unsurprisingly the accuracy is not very high on real data.

- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data - the offending examples are close-ups where the face fills the entire image (no such data was in the training set).
- We build a web search engine for the USA market and want to deploy it in the UK.

In short, there are many cases where training and test distribution $p(x)$ are different. In some cases, we get lucky and the models work despite the covariate shift. We now discuss principled solution strategies. Warning - this will require some math and statistics.

6.16.4 Covariate Shift Correction

Assume that we want to estimate some dependency $p(y|x)$ for which we have labeled data (x_i, y_i). Alas, the observations x_i are drawn from some distribution $q(x)$ rather than the 'proper' distribution $p(x)$. To make progress, we need to reflect about what exactly is happening during training: we iterate over training data and associated labels $\{(x_1, y_1), \ldots (y_n, y_n)\}$ and update the weight vectors of the model after every minibatch.

Depending on the situation we also apply some penalty to the parameters, such as weight decay, dropout, zoneout, or anything similar. This means that we largely minimize the loss on the training.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(x_i, y_i, f(x_i)) + \text{some penalty}(w)$$

Statisticians call the first term an *empirical average*, that is an average computed over the data drawn from $p(x)p(y|x)$. If the data is drawn from the 'wrong' distribution q, we can correct for that by using the following simple identity:

$$\begin{aligned} \int p(x)f(x)dx &= \int p(x)f(x)\frac{q(x)}{p(x)}dx \\ &= \int q(x)f(x)\frac{p(x)}{q(x)}dx \end{aligned}$$

In other words, we need to re-weight each instance by the ratio of probabilities that it would have been drawn from the correct distribution $\beta(x) := p(x)/q(x)$. Alas, we do not know that ratio, so before we can do anything useful we need to estimate it. Many methods are available, e.g. some rather fancy operator theoretic ones which try to recalibrate the expectation operator directly using a minimum-norm or a maximum entropy principle. Note that for any such approach, we need samples drawn from both distributions - the 'true' p, e.g. by access to training data, and the one used for generating the training set q (the latter is trivially available).

In this case there exists a very effective approach that will give almost as good results: logistic

regression. This is all that is needed to compute estimate probability ratios. We learn a classifier to distinguish between data drawn from $p(x)$ and data drawn from $q(x)$. If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of the two distributions. On the other hand, any instances that can be well discriminated should be significantly over/underweighted accordingly. For simplicity's sake assume that we have an equal number of instances from both distributions, denoted by $x_i \sim p(x)$ and $x_i' \sim q(x)$ respectively. Now denote by z_i labels which are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$p(z=1|x) = \frac{p(x)}{p(x)+q(x)} \text{ and hence } \frac{p(z=1|x)}{p(z=-1|x)} = \frac{p(x)}{q(x)}$$

Hence, if we use a logistic regression approach where $p(z=1|x) = \frac{1}{1+\exp(-f(x))}$ it follows that

$$\beta(x) = \frac{1/(1+\exp(-f(x)))}{\exp(-f(x)/(1+\exp(-f(x))))} = \exp(f(x))$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a reweighted minimization problem where we weigh terms by β, e.g. via the head gradients. Here's a prototypical algorithm for that purpose which uses an unlabeled training set X and test set Z:

1. Generate training set with $\{(x_i, -1)...(z_j, 1)\}$
2. Train binary classifier using logistic regression to get function f
3. Weigh training data using $\beta_i = \exp(f(x_i))$ or better $\beta_i = \min(\exp(f(x_i)), c)$
4. Use weights β_i for training on X with labels Y

Generative Adversarial Networks use the very idea described above to engineer a *data generator* such that it cannot be distinguished from a reference dataset. For this, we use one network, say f to distinguish real and fake data and a second network g that tries to fool the discriminator f into accepting fake data as real. We will discuss this in much more detail later.

6.16.5 Concept Shift Correction

Concept shift is much harder to fix in a principled manner. For instance, in a situation where suddenly the problem changes from distinguishing cats from dogs to one of distinguishing white from black animals, it will be unreasonable to assume that we can do much better than just training from scratch using the new labels. Fortunately, in practice, such extreme shifts almost never happen. Instead, what usually happens is that the task keeps on changing slowly. To make things more concrete, here are some examples:

- In computational advertising, new products are launched, old products become less popular. This means that the distribution over ads and their popularity changes gradually and any click-through rate predictor needs to change gradually with it.

- Traffic cameras lenses degrade gradually due to environmental wear, affecting image quality progressively.
- News content changes gradually (i.e. most of the news remains unchanged but new stories appear).

In such cases, we can use the same approach that we used for training networks to make them adapt to the change in the data. In other words, we use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

6.16.6 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in $p(x)$ and in $p(y|x)$, let us consider a number of problems that we can solve using machine learning.

- **Batch Learning.** Here we have access to training data and labels $\{(x_1, y_1), \ldots (x_n, y_n)\}$, which we use to train a network $f(x, w)$. Later on, we deploy this network to score new data (x, y) drawn from the same distribution. This is the default assumption for any of the problems that we discuss here. For instance, we might train a cat detector based on lots of pictures of cats and dogs. Once we trained it, we ship it as part of a smart catdoor computer vision system that lets only cats in. This is then installed in a customer's home and is never updated again (barring extreme circumstances).
- **Online Learning.** Now imagine that the data (x_i, y_i) arrives one sample at a time. More specifically, assume that we first observe x_i, then we need to come up with an estimate $f(x_i, w)$ and only once we've done this, we observe y_i and with it, we receive a reward (or incur a loss), given our decision. Many real problems fall into this category. E.g. we need to predict tomorrow's stock price, this allows us to trade based on that estimate and at the end of the day we find out whether our estimate allowed us to make a profit. In other words, we have the following cycle where we are continuously improving our model given new observations.

model $f_t \longrightarrow$ data $x_t \longrightarrow$ estimate $f_t(x_t) \longrightarrow$ observation $y_t \longrightarrow$ loss $l(y_t, f_t(x_t)) \longrightarrow$ model f_{t+1}

- **Bandits.** They are a *special case* of the problem above. While in most learning problems we have a continuously parametrized function f where we want to learn its parameters (e.g. a deep network), in a bandit problem we only have a finite number of arms that we can pull (i.e. a finite number of actions that we can take). It is not very surprising that for this simpler problem stronger theoretical guarantees in terms of optimality can be obtained. We list it mainly since this problem is often (confusingly) treated as if it were a distinct learning setting.
- **Control (and nonadversarial Reinforcement Learning).** In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it'll just remember and the response will depend on what happened before. E.g. a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional integral derivative) controller algorithms are a popular choice

there. Likewise, a user's behavior on a news site will depend on what we showed him previously (e.g. he will read most news only once). Many such algorithms form a model of the environment in which they act such as to make their decisions appear less random (i.e. to reduce variance).

- **Reinforcement Learning.** In the more general case of an environment with memory, we may encounter situations where the environment is trying to *cooperate* with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to *win*. Chess, Go, Backgammon or StarCraft are some of the cases. Likewise, we might want to build a good controller for autonomous cars. The other cars are likely to respond to the autonomous car's driving style in nontrivial ways, e.g. trying to avoid it, trying to cause an accident, trying to cooperate with it, etc.

One key distinction between the different situations above is that the same strategy that might have worked throughout in the case of a stationary environment, might not work throughout when the environment can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once he starts exploiting it. The speed and manner at which the environment changes determines to a large extent the type of algorithms that we can bring to bear. For instance, if we *know* that things may only change slowly, we can force any estimate to change only slowly, too. If we know that the environment might change instantaneously, but only very infrequently, we can make allowances for that. These types of knowledge are crucial for the aspiring data scientist to deal with concept shift, i.e. when the problem that he is trying to solve changes over time.

6.16.7 Summary

- In many cases training and test set do not come from the same distribution. This is called covariate shift.
- Covariate shift can be detected and corrected if the shift isn't too severe. Failure to do so leads to nasty surprises at test time.
- In some cases the environment *remembers* what we did and will respond in unexpected ways. We need to account for that when building models.

6.16.8 Problems

1. What could happen when we change the behavior of a search engine? What might the users do? What about the advertisers?
2. Implement a covariate shift detector. Hint - build a classifier.
3. Implement a covariate shift corrector.
4. What could go wrong if training and test set are very different? What would happen to the sample weights?

6.16.9 Scan the QR Code to Discuss

6.17 Predicting House Prices on Kaggle

The previous chapters introduced a number of basic tools to build deep networks and to perform capacity control using dimensionality, weight decay and dropout. It's time to put our knowledge to good use by participating in a Kaggle competition. Predicting house prices is the perfect start, since its data is fairly generic and doesn't have much regular structure in the way text of images do. Note that the dataset is *not* the Boston housing dataset of Harrison and Rubinfeld, 1978. Instead, it consists of the larger and more fully-featured dataset of house prices in Ames, IA covering 2006-2010. It was collected by Bart de Cock in 2011. Due to its larger size it presents a slightly more interesting estimation problem.

In this chapter we will apply what we've learned so far. In particular, we will walk you through details of data preprocessing, model design, hyperparameter selection and tuning. We hope that through a hands-on approach you will be able to observe the effects of capacity control, feature extraction, etc. in practice. Such experience is vital if you want to become an experienced data scientist.

6.17.1 Kaggle

Kaggle is a popular platform for machine learning competitions. It combines data, code and users in a way to allow for both collaboration and competition. For instance, you can see the code that (some) competitors submitted and you can see how well you're doing relative to everyone else. If you want to participate in one of the competitions, you need to register for an account. So it's best to do this now.

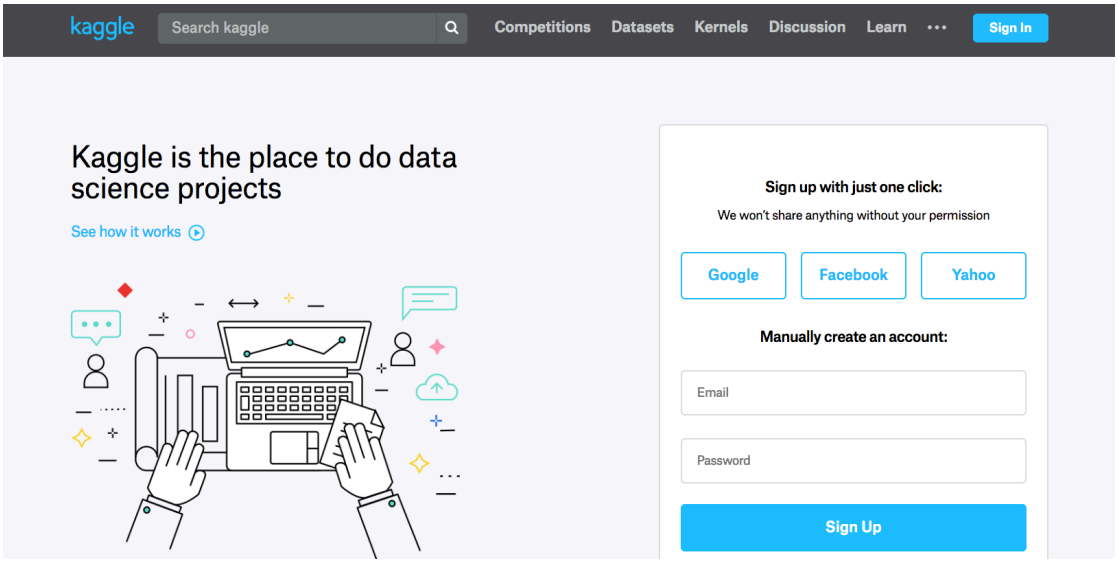


Fig. 6.11: Kaggle website

On the House Prices Prediction page you can find the data set (under the data tab), submit predictions, see your ranking, etc.; You can find it at the URL below:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

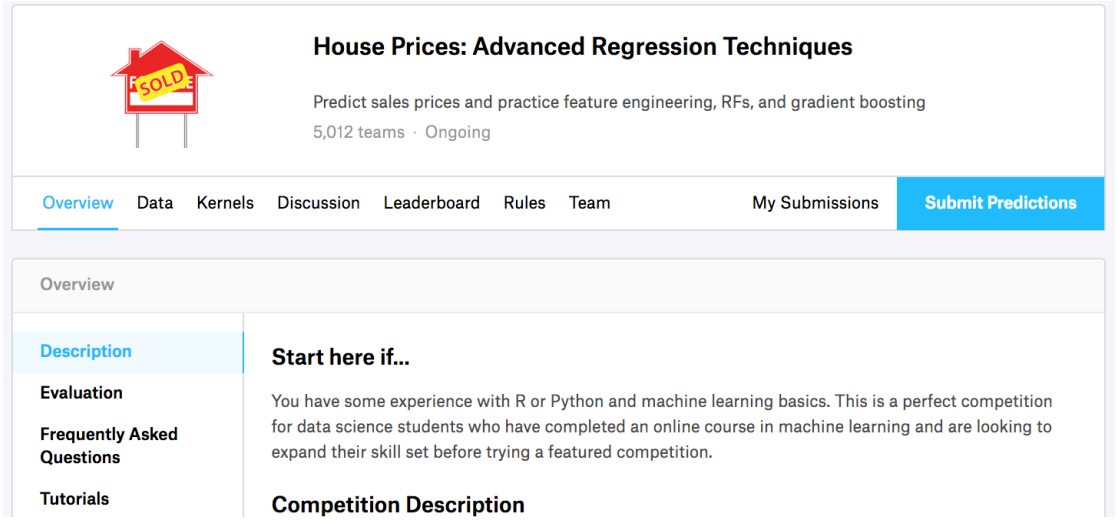


Fig. 6.12: House Price Prediction

6.17.2 Accessing and Reading Data Sets

The competition data is separated into training and test sets. Each record includes the property values of the house and attributes such as street type, year of construction, roof type, basement condition. The data includes multiple datatypes, including integers (year of construction), discrete labels (roof type), floating point numbers, etc.; Some data is missing and is thus labeled 'na'. The price of each house, namely the label, is only included in the training data set (it's a competition after all). The 'Data' tab on the competition tab has links to download the data.

We will read and process the data using `pandas`, an efficient data analysis toolkit. Make sure you have `pandas` installed for the experiments in this section.

```
In [1]: # If pandas is not installed, please uncomment the following line:
        # !pip install pandas

        import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
        import numpy as np
        import pandas as pd
```

For convenience we already downloaded the data and stored it in the `../data` directory. To load the two CSV (Comma Separated Values) files containing training and test data respectively we use Pandas.

```
In [2]: train_data = pd.read_csv('../data/kaggle_house_pred_train.csv')
        test_data = pd.read_csv('../data/kaggle_house_pred_test.csv')
```

The training data set includes 1,460 examples, 80 features, and 1 label., the test data contains 1,459 examples and 80 features.

```
In [3]: print(train_data.shape)
        print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

Let's take a look at the first 4 and last 2 features as well as the label (SalePrice) from the first 4 examples:

```
In [4]: train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]]
```

Out[4]:

	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

We can see that in each example, the first feature is the ID. This helps the model identify each training example. While this is convenient, it doesn't carry any information for prediction purposes. Hence we remove it from the dataset before feeding the data into the network.

```
In [5]: all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

6.17.3 Data Preprocessing

As stated above, we have a wide variety of datatypes. Before we feed it into a deep network we need to perform some amount of processing. Let's start with the numerical features. We begin by replacing missing values with the mean. This is a reasonable strategy if features are missing at random. To adjust them to a common scale we rescale them to zero mean and unit variance. This is accomplished as follows:

$$x \leftarrow \frac{x - \mu}{\sigma}$$

To check that this transforms x to data with zero mean and unit variance simply calculate $\mathbf{E}[(x - \mu)/\sigma] = (\mu - \mu)/\sigma = 0$. To check the variance we use $\mathbf{E}[(x - \mu)^2] = \sigma^2$ and thus the transformed variable has unit variance. The reason for 'normalizing' the data is that it brings all features to the same order of magnitude. After all, we do not know *a priori* which features are likely to be relevant. Hence it makes sense to treat them equally.

```
In [6]: 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 = all_features.fillna(0)
```

Next we deal with discrete values. This includes variables such as 'MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, 'MSZoning' assumes the values 'RL' and 'RM'. They map into vectors $(1, 0)$ and $(0, 1)$ respectively. Pandas does this automatically for us.

```
In [7]: # Dummy_na=True refers to a missing value being a legal eigenvalue, and
        # creates an indicative feature for it
        all_features = pd.get_dummies(all_features, dummy_na=True)
        all_features.shape
```

```
Out[7]: (2919, 354)
```

You can see that this conversion increases the number of features from 79 to 331. Finally, via the `values` attribute we can extract the NumPy format from the Pandas dataframe and convert it into MXNet's native representation - NDArray for training.

```
In [8]: n_train = train_data.shape[0]
        train_features = nd.array(all_features[:n_train].values)
        test_features = nd.array(all_features[n_train:].values)
        train_labels = nd.array(train_data.SalePrice.values).reshape((-1, 1))
```

6.17.4 Training

To get started we train a linear model with squared loss. This will obviously not lead to a competition winning submission but it provides a sanity check to see whether there's meaningful information in the data. It also amounts to a minimum baseline of how well we should expect any 'fancy' model to work.

```
In [9]: loss = gloss.L2Loss()

        def get_net():
            net = nn.Sequential()
            net.add(nn.Dense(1))
            net.initialize()
            return net
```

House prices, like shares, are relative. That is, we probably care more about the relative error $\frac{y-\hat{y}}{y}$ than about the absolute error. For instance, getting a house price wrong by USD 100,000 is terrible in Rural Ohio, where the value of the house is USD 125,000. On the other hand, if we err by this amount in Los Altos Hills, California, we can be proud of the accuracy of our model (the median house price there exceeds 4 million).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the error that is being used to measure the quality in this competition. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{y}_i)^2}$$

```
In [10]: 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 = nd.clip(net(features), 1, float('inf'))
             rmse = nd.sqrt(2 * loss(clipped_preds.log(), labels.log()).mean())
             return rmse.asscalar()
```

Unlike in the previous sections, the following training functions use the Adam optimization algorithm. Compared to the previously used mini-batch stochastic gradient descent, the Adam optimization algorithm is relatively less sensitive to learning rates. This will be covered in further detail later on when we discuss the details on Optimization Algorithms in a separate chapter.

```
In [11]: 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 = gdata.DataLoader(gdata.ArrayDataset(
                 train_features, train_labels), batch_size, shuffle=True)
             # The Adam optimization algorithm is used here
             trainer = gluon.Trainer(net.collect_params(), 'adam', {
                 'learning_rate': learning_rate, 'wd': weight_decay})
             for epoch in range(num_epochs):
```

```
            for X, y in train_iter:
                with autograd.record():
                    l = loss(net(X), y)
                l.backward()
                trainer.step(batch_size)
            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
```

6.17.5 k-Fold Cross-Validation

The k-fold cross-validation was introduced in the section where we discussed how to deal with "*Model Selection, Underfitting and Overfitting*". We will put this to good use to select the model design and to adjust the hyperparameters. We first need a function that returns the i-th fold of the data in a k-fold cros-validation procedure. It proceeds by slicing out the i-th segment as validation data and returning the rest as training data. Note - this is not the most efficient way of handling data and we would use something much smarter if the amount of data was considerably larger. But this would obscure the function of the code considerably and we thus omit it.

```
In [12]: 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 = nd.concat(X_train, X_part, dim=0)
                     y_train = nd.concat(y_train, y_part, dim=0)
             return X_train, y_train, X_valid, y_valid
```

The training and verification error averages are returned when we train k times in the k-fold cross-validation.

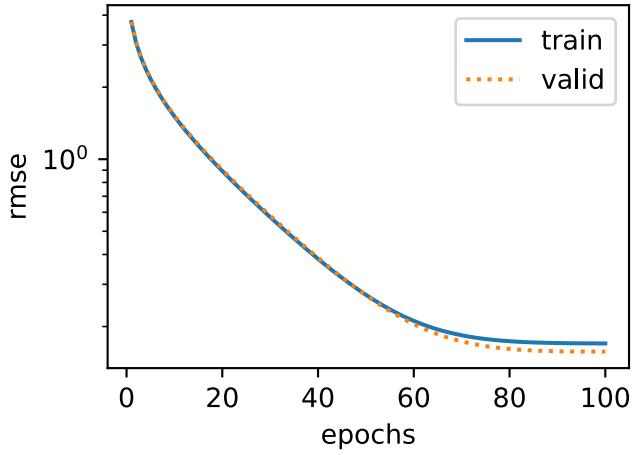
```
In [13]: def k_fold(k, X_train, y_train, num_epochs,
                    learning_rate, weight_decay, batch_size):
             train_l_sum, valid_l_sum = 0, 0
             for i in range(k):
                 data = get_k_fold_data(k, i, X_train, y_train)
                 net = get_net()
                 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.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse',
                                  range(1, num_epochs + 1), valid_ls,
```

```
                                ['train', 'valid'])
            print('fold %d, train rmse: %f, valid rmse: %f' % (
                i, train_ls[-1], valid_ls[-1]))
        return train_l_sum / k, valid_l_sum / k
```

6.17.6 Model Selection

We pick a rather un-tuned set of hyperparameters and leave it up to the reader to improve the model considerably. Finding a good choice can take quite some time, depending on how many things one wants to optimize over. Within reason the k-fold crossvalidation approach is resilient against multiple testing. However, if we were to try out an unreasonably large number of options it might fail since we might just get lucky on the validation split with a particular set of hyperparameters.

```
In [14]: 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('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
               % (k, train_l, valid_l))
```




```
fold 0, train rmse: 0.169922, valid rmse: 0.157187
fold 1, train rmse: 0.162306, valid rmse: 0.191155
fold 2, train rmse: 0.163406, valid rmse: 0.167690
fold 3, train rmse: 0.167915, valid rmse: 0.154745
fold 4, train rmse: 0.163010, valid rmse: 0.182964
5-fold validation: avg train rmse: 0.165312, avg valid rmse: 0.170748
```

You will notice that sometimes the number of training errors for a set of hyper-parameters can be very low, while the number of errors for the K-fold cross validation may be higher. This is most likely a consequence of overfitting. Therefore, when we reduce the amount of training errors, we need to check whether the amount of errors in the k-fold cross-validation have also been reduced accordingly.

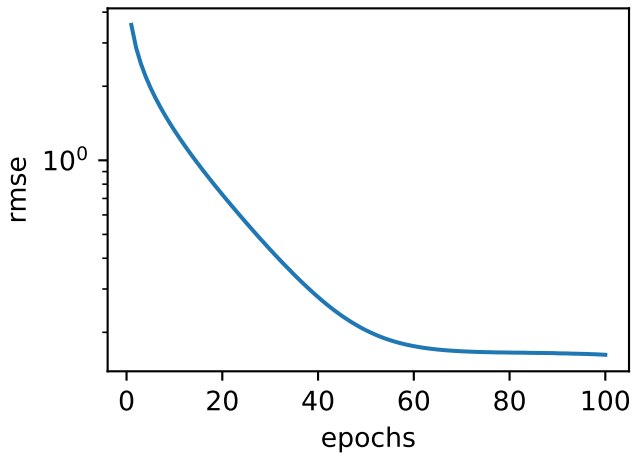
6.17.7 Predict and Submit

Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the crossvalidation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a CSV file will simplify uploading the results to Kaggle.

```
In [15]: 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.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse')
             print('train rmse %f' % train_ls[-1])
             # Apply the network to the test set
             preds = net(test_features).asnumpy()
             # Reformat it for 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)
```

Let's invoke the model. A good sanity check is to see whether the predictions on the test set resemble those of the k-fold crossvalication process. If they do, it's time to upload them to Kaggle.

```
In [16]: train_and_pred(train_features, test_features, train_labels, test_data,
                        num_epochs, lr, weight_decay, batch_size)
```




```
train rmse 0.162080
```

A file, `submission.csv` will be generated by the code above (CSV is one of the file formats accepted by Kaggle). Next, we can submit our predictions on Kaggle and compare them to the actual house price (label) on the testing data set, checking for errors. The steps are quite simple:

- Log in to the Kaggle website and visit the House Price Prediction Competition page.

- Click the “Submit Predictions”or “Late Submission”button on the right.
- Click the “Upload Submission File”button in the dashed box at the bottom of the page and select the prediction file you wish to upload.
- Click the “Make Submission”button at the bottom of the page to view your results.

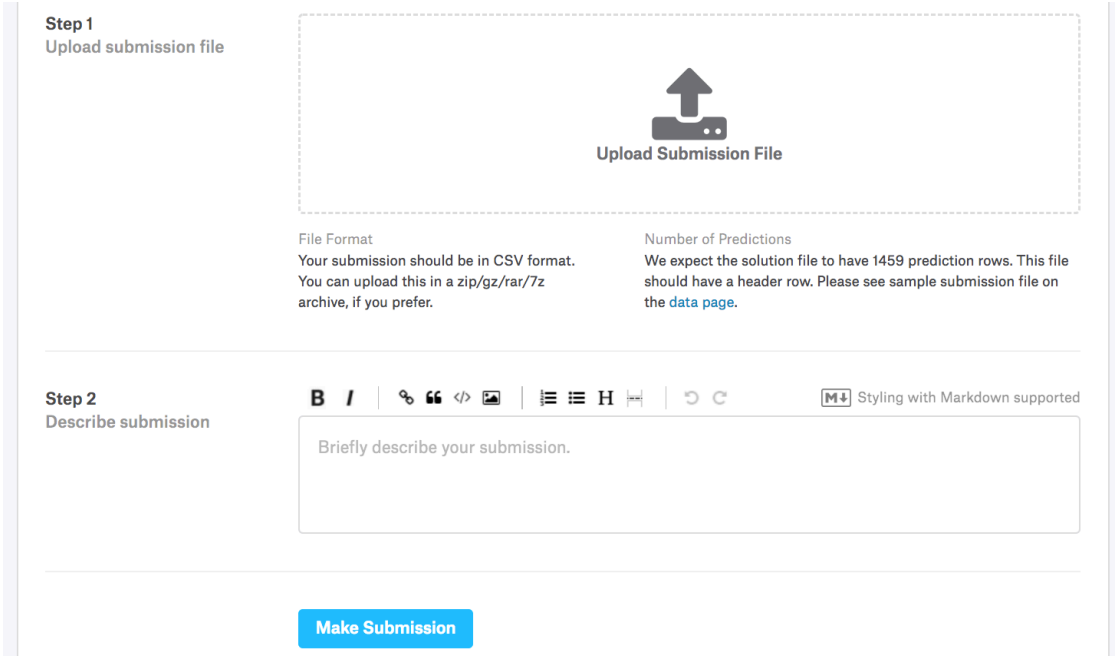


Fig. 6.13: Submitting data to Kaggle

6.17.8 Summary

- Real data often contains a mix of different datatypes and needs to be preprocessed.
- Rescaling real-valued data to zero mean and unit variance is a good default. So is replacing missing values with their mean.
- Transforming categorical variables into indicator variables allows us to treat them like vectors.
- We can use k-fold cross validation to select the model and adjust the hyper-parameters.
- Logarithms are useful for relative loss.

6.17.9 Problems

1. Submit your predictions for this tutorial to Kaggle. How good are your predictions?
2. Can you improve your model by minimizing the log-price directly? What happens if you try to predict the log price rather than the price?
3. Is it always a good idea to replace missing values by their mean? Hint - can you construct a situation where the values are not missing at random?
4. Find a better representation to deal with missing values. Hint - What happens if you add an indicator variable?
5. Improve the score on Kaggle by tuning the hyperparameters through k-fold crossvalidation.
6. Improve the score by improving the model (layers, regularization, dropout).
7. What happens if we do not standardize the continuous numerical features like we have done in this section?

6.17.10 Scan the QR Code to Discuss