

ESE 546: Principles of Deep Learning

Fall 2023

Instructor

Pratik Chaudhari pratikac@seas.upenn.edu

Teaching Assistants

Siming He (siminghe@seas)

Xijie Jiao (xijiej@seas)

Beiming Li (beimingl@seas)

Yankai Li (liyankai@seas)

Amila Mujkic (amilam@wharton)

Vaibhav Sahu (vaibhavs@seas)

Haoyu Wang (why16gzl@seas)

Ruiming Wu (wuru@seas)

December 12, 2023

Contents

1	What is intelligence?	5
1.1	Key components of intelligence	6
1.2	Intelligence: The Beginning (1942-50)	8
1.2.1	Representation Learning	9
1.3	Intelligence: Reloaded (1960-2000)	11
1.4	Intelligence: Revolutions (2006-)	13
1.5	A summary of our goals in this course	14
2	Linear Regression, Perceptron, Stochastic Gradient Descent	15
2.1	Problem setup for machine learning	15
2.1.1	Generalization	16
2.2	Linear regression	17
2.2.1	Maximum Likelihood Estimation	18
2.3	Perceptron	20
2.3.1	Surrogate Losses	21
2.4	Stochastic Gradient Descent	21
2.4.1	The general form of SGD	23
3	Kernels, Beginning of neural networks	24
3.1	Digging deeper into the perceptron	24
3.1.1	Convergence rate	24
3.1.2	Dual representation	25
3.2	Creating nonlinear classifiers from linear ones	26
3.3	Kernels	27
3.3.1	Kernel perceptron	28
3.3.2	Mercer's theorem	29
3.4	Learning the feature vector	30
3.4.1	Random features	31
3.4.2	Learning the feature matrix as well	32
4	Deep fully-connected networks, Backpropagation	34
4.1	Deep fully-connected networks	34
4.1.1	Some deep learning jargon	36
4.1.2	Weights	38
4.2	The backpropagation algorithm	38
4.2.1	One hidden layer with one neuron	39
4.2.2	Implementation of backpropagation	42

5	Convolutional Architectures	43
5.1	Basics of the convolution operation	44
5.1.1	Convolutions of 2D images	46
5.1.2	Some examples	47
5.2	How are convolutions implemented?	49
5.3	Convolutions for multi-channel images in a deep network	50
5.4	Translational equivariance using convolutions	51
5.5	Pooling to build translational invariance	52
6	Data augmentation, Loss functions	55
6.1	Data augmentation	55
6.1.1	Some basic data augmentation techniques	56
6.1.2	How does augmentation help?	56
6.1.3	What kind of augmentation to use when?	57
6.1.4	Data augmentation for text	57
6.2	Loss functions	57
6.2.1	Regression	58
6.2.2	Classification: Cross-Entropy loss	59
6.2.3	Softmax Layer	61
6.2.4	Label smoothing	62
6.2.5	Multiple ground-truth classes	62
7	Bias-Variance Trade-off, Dropout, Batch-Normalization	64
7.1	Bias-Variance Decomposition	64
7.1.1	Cross-Validation	68
7.2	Weight Decay	70
7.2.1	Do not do weight decay on biases	71
7.2.2	Maximum a posteriori (MAP) Estimation	72
7.3	Dropout	74
7.3.1	Bagging classifiers	75
7.3.2	Some insight into how dropout works	76
7.3.3	Implementation details of dropout	78
7.3.4	Using dropout as a heuristic estimate of uncertainty	78
7.4	Batch-Normalization	79
7.4.1	Covariate shift	79
7.4.2	Internal covariate shift	81
7.4.3	Problems with batch-normalization	84
7.4.4	Variants of Batch-Normalization	85
8	Recurrent Architectures and the Attention Mechanism	87
8.1	Recursive updates in a Kalman filter, sufficient statistics	88
8.2	Recurrent Neural Networks (RNNs)	90
8.2.1	Backpropagation in an RNN	92
8.2.2	Handling long-term temporal dependencies	93
8.3	Long Short-Term Memory (LSTM)	96
8.3.1	Gated Recurrent Units (GRUs)	97
8.3.2	LSTMs	97
8.4	Bidirectional architectures	98
8.5	Attention mechanism	99

8.5.1	Weighted regression estimate	100
8.5.2	Attention layer in deep networks	101
8.5.3	Attention in recurrent networks	106
8.6	Some applications of attention-based networks (transformers)	107
8.6.1	Pretraining on natural language	108
8.6.2	Handling multi-modal inputs	111
9	Background on Optimization, Gradient Descent	113
9.1	Convexity	114
9.2	Introduction to Gradient Descent	116
9.2.1	Conditions for optimality	117
9.2.2	Different types of convergence	117
9.3	Convergence rate for gradient descent	119
9.3.1	Some assumptions	119
9.3.2	GD for convex functions	119
9.3.3	Gradient descent for strongly convex functions	123
9.4	Limits on convergence rate of first-order methods	126
10	Accelerated Gradient Descent	127
10.1	Polyak's Heavy Ball method	127
10.1.1	Polyak's method can fail to converge	130
10.2	Nesterov's method	130
10.2.1	A model for understanding Nesterov's updates	131
10.2.2	How to pick the momentum parameter?	133
11	Stochastic Gradient Descent	135
11.1	SGD for least-squares regression	137
11.2	Convergence of SGD	138
11.2.1	Typical assumptions in the analysis of SGD	139
11.2.2	Convergence rate of SGD for strongly-convex functions	141
11.2.3	When should one use SGD in place of GD?	143
11.3	Accelerating SGD using momentum	144
11.3.1	Momentum methods do not accelerate SGD	145
11.4	The Adam optimizer	147
11.5	Understanding SGD as a Markov Chain	150
11.5.1	Gradient flow	151
11.5.2	Markov chains	151
11.5.3	A Markov chain model of SGD	154
11.5.4	The Gibbs distribution	156
11.5.5	Convergence of a Markov chain to its invariant distribution	158
12	Shape of the energy landscape of neural networks	161
12.1	Introduction	162
12.2	Deep Linear Networks	163
12.3	Extending the picture to deep networks	167

13	Generalization performance of machine learning models	169
13.1	The PAC-Learning model	169
13.2	Concentration of Measure	172
13.2.1	Union Bound (or Boole's Inequality)	173
13.2.2	Chernoff Bound	173
13.3	Uniform convergence	174
13.4	Vapnik-Chernovenkis (VC) dimension	177
14	Sloppy Models	181
14.1	Model manifold of nonlinear regression	181
14.2	Understanding optimization for sloppy models	185
14.3	Understanding generalization for sloppy models	188
15	Variational Inference	190
15.1	The model	190
15.2	Some technical basics	192
15.2.1	Variational calculus	192
15.2.2	Laplace approximation	194
15.2.3	Digging deeper into KL-divergence	195
15.3	Evidence Lower Bound (ELBO)	197
15.3.1	Parameterizing ELBO	199
15.4	Gradient of the ELBO	200
15.4.1	The Reparameterization Trick	201
15.4.2	Score-function estimator of the gradient	203
15.4.3	Gradient of the remaining terms in ELBO	204
15.5	Some comments	205
16	Generative Adversarial Networks	207
16.1	Two-sample tests and Discriminators	208
16.2	Building the Discriminator in a GAN	209
16.3	Building the Generator of a GAN	211
16.4	Putting the discriminator and generator together	212
16.5	How to perform validation for a GAN?	214
16.6	The zoo of GANs	216
	Bibliography	217

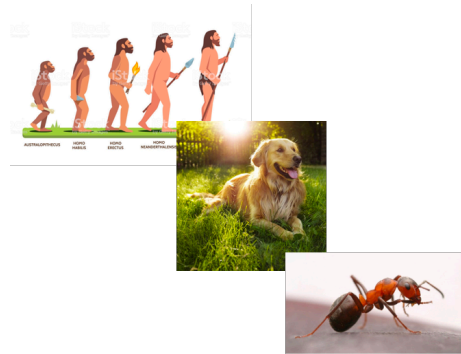
Chapter 1

What is intelligence?

Reading

1. Bishop 1.1-1.5
2. Goodfellow Chapter 1
3. “A logical calculus of the ideas immanent in nervous activity” by Warren McCulloch and Walter Pitts ([McCulloch and Pitts, 1943](#)).
4. “Computing machinery and intelligence” by Alan Turing in 1950 ([Turing, 2009](#)).

What is intelligence? It is hard to define, I don't know a good definition. We certainly know it when we see it. All humans are intelligent. Dogs are plenty intelligent. Most of us would agree that a house fly or an ant is less intelligent than a dog. What are the common features of these species? They all can gather food, search for mates and reproduce, adapt to changing environments and, in general, the ability to survive.



Are plants intelligent? Plants have sensors, they can measure light, temperature, pressure etc. They possess reflexes, e.g., sunflowers follow the sun. This is an indication of “reactive/automatic intelligence”. The mere existence of a sensory and actuation mechanism is not an indicator of

1 intelligence. Plants cannot perform planned movements, e.g., they cannot
2 travel to new places.



Figure 1.1: A Tunicate on the ocean floor

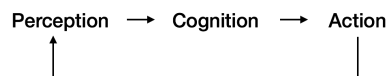
3 A Tunicate in Figure 1.1 is an interesting plant however. Tunicates are
4 invertebrates. When they are young they roam around the ocean floor in
5 search of nutrients, and they also have a nervous system (ganglion cells)
6 at this point of time that helps them do so. Once they find a nutritious
7 rock, they attach themselves to it and then eat and digest their own brain.
8 They do not need it anymore. They are called “tunicates” because they
9 develop a thick covering (shown above) or a “tunic” to protect themselves.

10 Is a program like AlphaGo intelligent? There is a very nice movie on
11 Netflix on the development of AlphaGo and here’s an excerpt from the
12 movie (<https://youtu.be/YrTRKh4FPio>). The commentator in this video
13 is wondering how Lee Se-dol, who was one of the most accomplished
14 Go players in the world then, might defeat this very powerful program;
15 this was I believe after AlphaGo was up 3-0 in the match already. The
16 commentator says so very nonchalantly: if you want to defeat AlphaGo
17 all you have to do is pull the plug.

18 A key indicator of intelligence (and this is just my opinion) is the
19 ability to take actions upon the world. With this comes the ability to affect
20 your environment, preempt antagonistic agents in the environment and
21 take actions that achieve your desired outcomes. You should not think
22 of intelligence (artificial or otherwise) as something that takes a dataset
23 and learns how to make predictions using this dataset. For example, if
24 I dropped my keys at the back of the class, I cannot possibly find them
25 without moving around, using priors of where keys typically hide (which
26 is akin to learning from a dataset) only helps us search more efficiently.

27 1.1 Key components of intelligence

28 With this definition, we can write down the three key parts that an
29 intelligent, autonomous agent possesses as follows.



30 Perception refers to the sensory mechanisms to gain information about
31 the environment (eyes, ears, smell, tactile input etc.). Action refers to

1 your hands, legs, or motors/engines in machines that help you move on
 2 the basis of this information. Cognition is kind of the glue in between.
 3 It is in charge of crunching the information of your sensors, creating
 4 a good “representation” of the world around you and then undertaking
 5 actions based on this representation. The three facets of intelligence are
 6 not sequential and intelligence is not merely a feed-forward process. Your
 7 sensory inputs depend on the previous action you took. While searching
 8 for something you take actions that are explicitly designed to give you
 9 different sensory inputs than what you are getting at the moment.

10 This class will focus on learning. It is a component, not the entirety,
 11 of cognition.

Learning is in charge of looking at past data and predicting what future data may look like.

12 Cognition also involves handling situations when the current data does
 13 not match past data, etc. To give you an example, arithmetic problems
 14 you solved in elementary school are akin to learning whereas figuring out
 15 that taking a standard deduction when you file your income tax versus
 16 itemized deduction is like cognition...

17 Some other classes at Penn that address these various aspects of
 18 intelligence are:

- 19 • Perception: CIS 580, CIS 581, CIS 680, MEAM 620, ESE 650
- 20 • Learning/Cognition: CIS 519, CIS 520, CIS 521, CIS 522, CIS
 21 620, ESE 542, ESE 545, CIS 700
- 22 • Control: ESE 650, MEAM 520/620, ESE 500/505, ESE 615, ESE
 23 618, MEAM 517

24 **The objective of the learning process is really to crunch** 25 **past data and learn a prior**

26 Imagine a supreme agent which is infinitely fast, clever, and can interpret
 27 its sensory data and compute the best actions for any task, say driving.
 28 Learning from past data is not essential for such an agent; effectively the
 29 supreme agent can simulate every physical process around it quickly and
 30 decide upon the best action it should take. Past data helps if you are not as
 31 fast as the supreme agent or if you want to save some compute time/energy
 32 during decision making.

A deep network or a machine learning model is not a mechanism

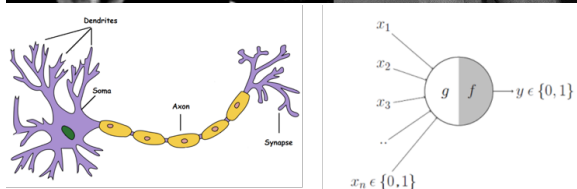
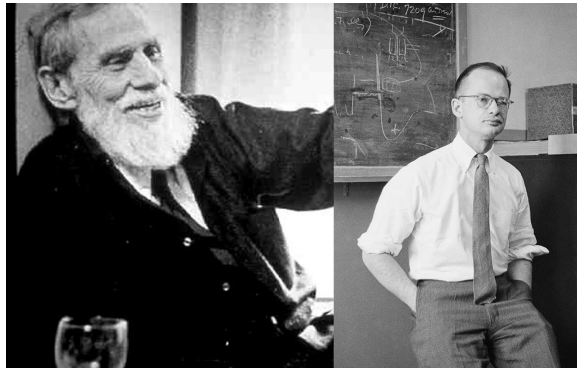
🔍 There are also situations when you do not have enough information to make a decision, e.g., you do not precisely know the future location of the car in front of you while driving. Will the supreme agent benefit from seeing historical data in this case?

that directly undertakes the actions. It is rather a prior on the possible actions to take. Other algorithms that rely on real-time sensory data will be in charge of picking one action out of these predictions. This is very easy to appreciate in robotics: how a car should move depends more upon the real-time data than any amount of past data. This aspect is less often appreciated in non-robotics applications but it holds there as well. Even for something like a recommendation engine that recommends movies in Netflix, the output of a prediction model will typically be modified by a number of algorithms before it is actually recommended to the user, e.g., filters for sensitive information, or toxicity in a chatbot.

1.2 Intelligence: The Beginning (1942-50)

Let us give a short account of how our ideas about intelligence have evolved.

The story begins roughly in 1942 in Chicago. These are Warren McCulloch who was a neuroscientist and Walter Pitts who studied mathematical logic. They built the first model of a mechanical neuron and propounded the idea that simple elemental computational blocks in your brain work together to perform complex functions. Their paper ([McCulloch and Pitts, 1943](#)) is an assigned reading for this lecture.



A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

■ WARREN S. MCCULLOCH AND WALTER PITTS
 University of Illinois, College of Medicine,
 Department of Psychiatry at the Illinois Neuropsychiatric Institute,
 University of Chicago, Chicago, U.S.A.

1 Around the same time in England, Alan Turing was forming his initial
2 ideas on computation and neurons. He had already published his paper on
3 computability by then. This paper (Turing, 2009) is the second assigned
4 reading for this lecture. ¹

VOL. LIX. No. 236.]

[October, 1950

M I N D

A QUARTERLY REVIEW

OF

PSYCHOLOGY AND PHILOSOPHY

I.—COMPUTING MACHINERY AND
INTELLIGENCE

By A. M. TURING

1. *The Imitation Game.*

6 McCulloch was inspired by Turing's idea of building a machine that could
7 compute any function in finitely-many steps. In his mind, the neuron in a
8 human brain, which either fires or does not fire depending upon the stimuli
9 of the other neurons connected to it, was a binary object; rules of logic
10 where a natural way to link such neurons, just like the Pitt's hero Bertrand
11 Russell rebuilt modern mathematics using logic. Together, McCulloch
12 & Pitts' and Turing's work already had all the germs of neural networks
13 as we know them today: nonlinearities, networks of a large number of
14 neurons, training the weights *in situ* etc.

15 Let's now move to Cambridge, Massachusetts. Norbert Wiener, who
16 was a famous professor at MIT, had created a little club of enthusiasts
17 around 1942. They would coin the term "Cybernetics" to study exactly
18 the perception-cognition-action loop we talked about. You can read more
19 in the original book titled "Cybernetics: or control and communication in
20 the animal and the machine" (Wiener, 1965). You can also look at the
21 book "The Cybernetic Brain" (Pickering, 2010) to read more.

22 1.2.1 Representation Learning

23 Perceptual agents, from plants to humans, perform measurements of
24 physical processes ("signals") at a level of granularity that is essentially
25 continuous. They also perform actions in the physical space, which is
26 again continuous. Cognitive science on the other hand thinks in terms of
27 discrete entities like concepts, ideas, objects, or categories. These can be
28 manipulated with tools of logic and inference. It is useful to ask what
29 information is transferred from the perception system to the cognition
30 system to create such symbols from signals, or from cognition to control
31 which creates back signals from the symbols? We will often call these
32 symbols the "internal representation" of an agent.

33 Claude Shannon formulated information theory which is one way to
34 study these kind of ideas. Shannon devised a representation learning
35 scheme for compressing (e.g., taking the intensities at each pixel of the

¹If you need more inspiration to go and read it, the first section titled "The Imitation Game" propounds the Turing Test.



Figure 1.2. The four pioneers of cybernetics (left to right): Ross Ashby, Warren McCulloch, Grey Walter, and Norbert Wiener. Source: de Latil 1954, facing p. 53.

Figure 1.2: The famous four of the first era of intelligence. (From right to left) Norbert Wiener, Grey Walter, Warren McCulloch and Walter Pitts

CONTENTS

Preface to the Second Edition vii

PART I
ORIGINAL EDITION
1948

Introduction 1

I Newtonian and Bergsonian Time 30

II Groups and Statistical Mechanics 45

III Time Series, Information, and Communication 60

IV Feedback and Oscillation 95

V Computing Machines and the Nervous System 116

VI Gestalt and Universals 133

VII Cybernetics and Psychopathology 144

VIII Information, Language, and Society 155

PART II
SUPPLEMENTARY CHAPTERS
1961

IX On Learning and Self-Reproducing Machines 169

X Brain Waves and Self-Organizing Systems 181

Index 205

Figure 1.3: About 75 years later, this course's content is (surprisingly) closely related to the topics in Wiener's book on Cybernetics.

1 camera and encoding them into something less redundant like JPEG),
2 coding (adding redundancy into the representation to gain resilience to
3 noise before transmitting it across some physical medium such as a wireless
4 channel), decoding (using the redundancy to guess the parts of the data
5 packet that were corrupted during transmission) and finally decompressing
6 the data (getting the original signal back, e.g., pixel intensities from JPEG).

7 Information theory as described above is a tool to transmit data
8 correctly between a sender and a receiver. We will use this theory for a
9 different purpose. Compression, decompression etc. care about never
10 losing information from the data; machine learning necessarily requires
11 you forget *some* of the data. If the model focuses too much on the grass
12 next to the dogs in the dataset, it will “over-fit” to the data and next
13 time when you see grass, it will end up predicting a dog. It not easy to
14 determine which parts of the data one should forget and which parts one
15 should remembered.

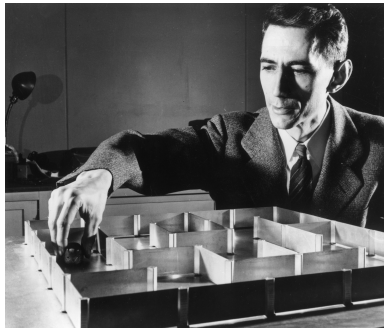


Figure 1.4: Claude Shannon studied information theory. This is a picture of a maze solving mouse that he made around 1950, among the world’s first examples of machine learning; read more at <https://www.technologyreview.com/2018/12/19/138508/mighty-mouse>.

16 The study of artificial intelligence has always had this diverse flavor.
17 Computer scientists trying to understand perception, electrical engineers
18 trying to understand representations and mechanical and control engineers
19 building actuation mechanisms.

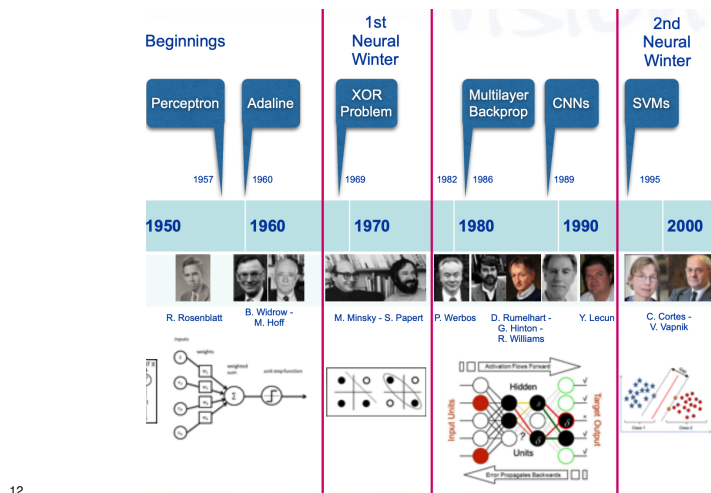
20 1.3 Intelligence: Reloaded (1960-2000)

21 The early period created interest in intelligence and developed some
22 basic ideas. The first major progress of what one would call the sec-
23 ond era was made by Frank Rosenblatt in 1957 at Cornell University.
24 Rosenblatt’s model called the perceptron is a model with a single bi-
25 nary neuron. It was a machine designed to distinguish punch cards
26 marked on the left from cards marked on the right, and it weighed
27 5 tons ([https://news.cornell.edu/stories/2019/09/professors-perceptron-](https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon)
28 [paved-way-ai-60-years-too-soon](https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon)). The input integration is implemented
29 through the addition of the weighted inputs that have fixed weights ob-
30 tained during the training stage. If the result of this operation is larger

1 than a given threshold, the neuron fires. When the neuron fires its output
2 is set to 1, otherwise it is set to 0. It looks like the function

$$f(x; w) = \text{sign}(w^\top x) = \text{sign}(w_1x_1 + \dots + x_dx_d).$$

3 Rosenblatt's perceptron ([Rosenblatt, 1958](#)) had a single neuron so it
4 could not handle complicated data. Marvin Minsky and Seymour Papert
5 discussed this in a famous book titled *Perceptrons* ([Minsky and Papert,](#)
6 [2017](#)). But unfortunately this book was widely perceived as two very
7 well established researchers being skeptical of artificial intelligence itself.
8 Interest in building neuron-based artificial intelligence (also called the
9 connectionist approach) waned as a result. The rise of symbolic reasoning
10 and the rise of computer science as a field coincided with these events in
11 the early 1970s and caused what one would call the “first AI winter”.



12

13 There was resurgence of ideas around neural networks, mostly fueled
14 by the (re)-discovery of back-propagation by [Rumelhart et al. \(1985\)](#); Shun-
15 ichi Amari developed methods to train multi-layer neural networks using
16 gradient descent all the way back in 1967 and this was also written up in a
17 book but it was in Japanese ([Amari, 1967](#)). Multi-layer networks came back
18 in vogue because they could now be trained reasonably well. This era also
19 brought along the rise of convolutional neural networks built upon a large
20 body of work starting from two neuroscientists Hubel and Wiesel who did
21 very interesting experiments in the 60s to discover visual cell types ([Hubel](#)
22 [and Wiesel, 1968](#)) and Fukushima who implemented convolutional and
23 downsampling layers in his famous Neocognitron ([Fukushima, 1988](#)).
24 Yann LeCun demonstrated classification of handwritten digits using CNNs
25 in the early 1990s and used it to sort zipcodes ([LeCun et al., 1989, 1998](#)).
26 Neural networks in the late 80s and early 90s was arguably, as popular a
27 field as it is today.

28 Support Vector Machines (SVMs) were invented in [Cortes and Vapnik](#)
29 ([1995](#)). These were (are) brilliant machine learning models with extremely
30 good performance. They were much easier to train than neural networks.
31 They also had a nice theoretical foundation and, in general were a delight

1 to use as compared to neural networks. It was famously said in the 90s that
2 only the neural network researchers were able to get good performance with
3 neural networks and no one else could train them well. This was largely
4 true even until 2015 or so before the rise of libraries like PyTorch and
5 TensorFlow. So we should give credit to these libraries for popularizing
6 deep learning in addition to all the researchers in deep learning.

7 Kernel methods, although known much before in the context of the
8 perceptron (Aizerman, 1964; Scholkopf and Smola, 2018), made SVMs
9 very powerful (we will see this in Chapter 2). The rise of Internet commerce
10 in the late 90s meant that a number of these algorithms found widespread
11 and impactful applications. Others such as random forests (Breiman,
12 2001) further led the progress in machine learning. Neural networks,
13 which worked well when they did but required a lot of tuning and expertise
14 to get to work, lost out to this competition. However, there were other
15 neural network-based models in the natural language processing (NLP)
16 community such as LSTMs (Hochreiter and Schmidhuber, 1997) which
17 were discovered in this period and have remained very popular and
18 performant all through.

19 1.4 Intelligence: Revolutions (2006-)

20 The growing quantity of data and computation came together in late 2000s
21 to create ideas like deep Belief Networks (Hinton et al., 2006), deep
22 Boltzmann machines (Salakhutdinov and Larochelle, 2010), large-scale
23 training using GPUs (Raina et al., 2009) etc. The watershed moment
24 that got everyone's attention was when Krizhevsky et al. (2012) trained
25 a convolutional neural network to show dramatic improvement in the
26 classification performance on a large dataset called ImageNet. This is a
27 dataset with 1.4 million images collected across 1000 different categories.
28 Performing well on this dataset was considered very difficult, the best
29 approaches in 2011 (ImageNet challenge used to be an annual competition
30 until 2016) achieved about 25% error. Krizhevsky et al. (2012) managed
31 to obtain an error of 15.3%. Many significant results in the world of
32 neural networks have been achieved since 2012. Today, deep networks
33 in their various forms run a large number of applications in computer
34 vision, natural language processing, speech processing, robotics, physical
35 sciences such as physics, chemistry and biology, medical sciences, and
36 many many others (LeCun et al., 2015).

37 This progress in deep learning has been driven by the availability
38 of data and cheap computation. Most importantly, it is driven today
39 by the intense curiosity of people from diverse fields of inquiry. Deep
40 learning in its modern form is a very young field. As is typical in new
41 fields, consolidation of ideas has not happened yet; so you will often see
42 conflicting explanations for the same concept. The dramatic progress
43 today is driven by ideas that are often quite unusual and a large number
44 of open problems remain in how we may build a more sophisticated
45 understanding of deep networks.

▲ You can explore the ImageNet dataset at
<https://navigu.net/#imagenet>.

1 1.5 A summary of our goals in this course

2 This course will take off from around late 1990s (kernel methods) and
3 develop ideas in deep learning that bring us to 2023. Our goals are to

- 4 1. become good at using modern deep networks, i.e., implementing
5 them, training them, modeling specific problems using ideas in deep
6 learning;
- 7 2. understanding why the many quixotic-looking ideas in deep networks
8 work.

9 After taking this course, we expect to be able to not only develop methods
10 that use deep learning, but more importantly improve existing ideas using
11 foundational understanding of the mathematics behind these ideas and
12 develop new ways of improving deep learning theory and practice.

1 Chapter 2

2 Linear Regression, 3 Perceptron, Stochastic 4 Gradient Descent

Reading

1. Bishop 3.1, 4.1, 4.3
2. Goodfellow Chapter 5.1-5.4

5 2.1 Problem setup for machine learning

6 Nature gives us data X and targets Y for this data.

$$X \rightarrow Y.$$

7 Nature does not usually tell us what property of a datum $x \in X$ results in
8 a particular prediction $y \in Y$. We would like to learn to imitate Nature,
9 namely predict y given x .

10 What does such learning mean? It is simply a notion of being able
11 to identify patterns in the input data without explicitly programming a
12 computer for prediction. We are often happy with a learning process
13 that identifies correlations: if we learn correlations on a few samples
14 $(x^1, y^1), \dots, (x^n, y^n)$, we may be able to predict the output for a new
15 datum x^{n+1} . We may not need to know *why* the label of x^{n+1} was
16 predicted to be so and so.

17 Let us say that Nature possesses a probability distribution P over
18 (X, Y) . We will formalize the problem of machine learning as Nature
19 drawing n independent and identically distributed samples from this

1 distribution. This is denoted by

$$D_{\text{train}} = \{(x^i, y^i) \sim P\}_{i=1}^n$$

2 is called the “training set”. We use this data to identify patterns that help
3 make predictions on some future data.

4 **What is the task in machine learning?**

5 Suppose D_{train} consists of $n = 50$ RGB images of size 100×100 of
6 two kinds, ones with an orange inside them and ones without. 10^4 is a
7 large number of pixels, each pixel taking any of the possible 255^3 values.
8 Suppose we discover that one particular pixel, say at location $(25, 45)$,
9 takes distinct values in all images inside our training set. We can then
10 construct a predictor based on this pixel. This predictor, it is a binary
11 classifier, perfectly maps the training images to their labels (orange: +1
12 or no orange: -1). If x_{ij}^k is the $(ij)^{\text{th}}$ pixel for image x^k , then we use the
13 function

$$f(x) = \begin{cases} y^k & \text{if } x_{ij}^k = x_{ij} \text{ for some } k = 1, \dots, n \\ -1 & \text{otherwise.} \end{cases}$$

14 This predictor certainly solves the task. It correctly works for all images
15 in the training set. Does it work for images outside the training set?

16 Our task in machine learning is to learn a predictor that works *outside*
17 the training set. The training set is only a source of information that Nature
18 gives us to find such a predictor.

Designing a predictor that is accurate on D_{train} is trivial. A hash function that memorizes the data is sufficient. This is NOT our task in machine learning. We want predictors that generalize to new data outside D_{train} .

19 **2.1.1 Generalization**

20 If we never see data from outside D_{train} why should we hope to do well
21 on it? The key is the distribution P . Machine learning is formalized as
22 constructing a predictor that works well on new data that is also drawn
23 independently from the distribution P . We will call this set of data the
24 “test set”.

$$D_{\text{test}}$$

25 This assumption is important. It provides coherence between past and
26 future samples: past samples that were used to train and future samples
27 that we will wish to predict upon.

28 How to find such predictors that work well on new data? The central
29 idea in machine learning is to restrict the set of possible binary functions
30 that we consider.

🔗 How many such binary classifiers are there at most?

We are searching for a predictor that generalizes well but only have the training dataset to ascertain which predictor generalizes well.

1 The *right* class of functions f is such that is not too large. Otherwise
 2 we will find our binary classifier discussed above as the solution of the
 3 problem (after all, it does achieve zero training error). This is not very
 4 useful. The class of functions that we should search over cannot be too
 5 small either, otherwise we won't be able to make accurate predictions for
 6 difficult images.

Finding an appropriate class of functions that is neither too big nor too small and finding one predictor from within this class that fits the training dataset well is what machine learning is all about.

🔗 Can you now think how machine learning is different from other fields you might know such as statistics or optimization?

7 2.2 Linear regression

8 Let us focus on a simpler problem. We fix the class of functions, our
 9 predictors, to only have linear classifiers. We will consider that our data
 10 $X \subset \mathbb{R}^d$ and labels $Y \subset \mathbb{R}$. If the labels/targets are real-valued, we call it
 11 a regression problem. Our predictor for any $x \in X$ is

$$f(x; w, b) = w^\top x + b. \quad (2.1)$$

12 This is a linear function in the data x with parameters $w \in \mathbb{R}^d$ and
 13 $b \in \mathbb{R}$. Different settings of w and b give different functions f . Picking a
 14 particular function f is therefore equivalent to picking particular values
 15 of the parameters. Parameters are also called weights. We can visualize
 16 what this predictor does in two ways. Consider the case of $d = 2$.

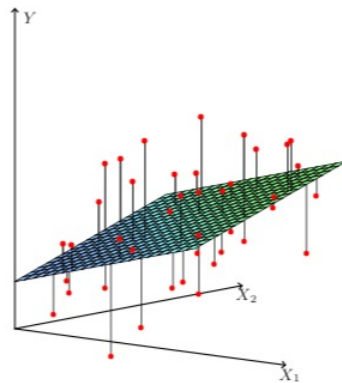


Figure 2.1: Linear least squares with $X \subset \mathbb{R}^2$.

17 Figure 2.1 shows the hyperplane corresponding to a particular (w, b)
 18 with the data x^i, y^i (in red). Each hyperplane is a particular predictor

1 $f(x; w, b)$. You can also think of the function f as a point in three
2 dimensional space $w \in \mathbb{R}^2$ and $b \in \mathbb{R}$.

3 Predicting the target accurately using this linear model would require us
4 to find values (w, b) that minimize the average distance to the hyperplane
5 of each sample in the training dataset. We write this as an *objective*
6 *function*.

$$\begin{aligned} \ell(w, b) &:= \frac{1}{2n} \sum_{i=1}^n (y^i - \hat{y}^i)^2 \\ &= \frac{1}{2n} \sum_{i=1}^n (y^i - w^\top x^i - b)^2 \end{aligned} \quad (2.2)$$

7 where we have written the prediction as

$$\hat{y}^i = w^\top x^i + b.$$

8 The quadratic term for each datum $\frac{1}{2} (y^i - \hat{y}^i)^2$ is known as the *loss*
9 *function*, or *loss for short*. The objective above is thus an average of the
10 loss for each datum. Finding the best weights w, b now boils down to
11 solving the optimization problem

$$w^*, b^* = \underset{w \in \mathbb{R}^d, b \in \mathbb{R}}{\operatorname{argmin}} \ell(w, b). \quad (2.3)$$

12 **How do we solve the optimization problem?** We will learn many
13 techniques to solve problems of the form (2.3). We have a simple case
14 here and therefore can use what you did in HW0. The solution is given by

$$w^* = (\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^\top \mathbf{Y} \quad (2.4)$$

15 where we have denoted by $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times (d+1)}$ the matrix whose i^{th} row is
16 the datum with a constant entry 1 appended at the end $[x^i, 1]$. Similarly
17 $\mathbf{Y} \in \mathbb{R}^n$ is a vector whose i^{th} entry is the target y^i .

18 2.2.1 Maximum Likelihood Estimation

19 There is another perspective to fitting a machine learning model. We will
20 *suppose* that our training data was created using a statistical model. We
21 can write this as

$$y = w^\top x + b + \epsilon \quad (2.5)$$

22 Of course we do not know whether Nature used this particular model
23 $f(x; w, b) := w^\top x + b$ to create the data. It might have created the data
24 using some other model, e.g., $f(x; A, w, b) := w^\top \sin(Ax) + b$.

This discrepancy between the two, the model that we fit upon the data and the true model that Nature could have used to create the data, is *modeled* as noise ϵ . Noise in machine learning comes from the fact that we—the users—do not know Nature's model.

❓ Why use the average, as opposed to say the maximum value?

❓ When is our solution to least squares regression in (2.4) not defined?

❓ What are we losing by fitting a linear predictor? Will this work if the true model from which Nature generates the data was different, say a polynomial?

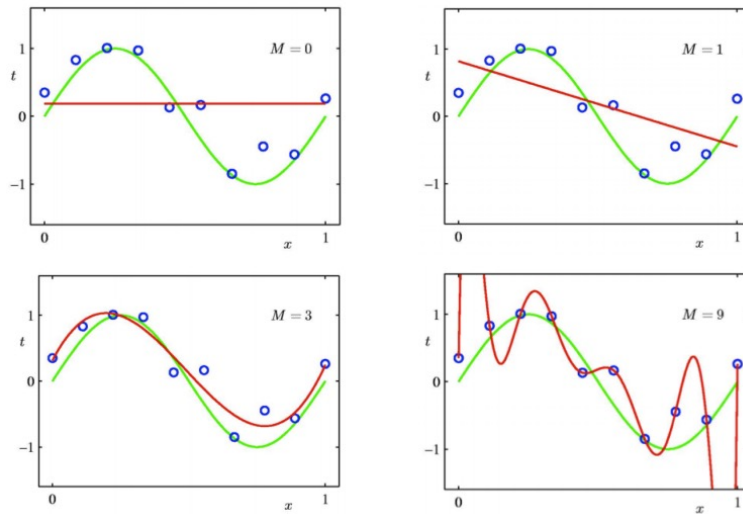


Figure 2.2: Least squares fitting using polynomials. As the degree of the polynomial M increases the predictor f fits the training data (in blue) better and better. But such a well-fitted predictor may be very different from the true model from which Nature generated the data (in green). The red curve in the fourth panel in these cases is said to have been *over-fitted*.

- 1 What model is appropriate for the noise ϵ ? There can be many models
- 2 depending upon your experiment (think of a model that predicts the arrival
- 3 time of a bus at the bus stop, what noise would you use?). For our purpose
- 4 we will use zero-mean Gaussian noise

$$\epsilon \sim N(0, \sigma_\epsilon^2)$$

- 5 that does not depend on the input x . The probability that a sample (x^i, y^i)
- 6 in our dataset D_{train} was created using our statistical model is then

$$p(y^i | x^i; w, b) = N(w^\top x^i + b, \sigma_\epsilon^2).$$

- 7 We have assumed that the data was drawn iid by Nature so the likelihood
- 8 of our entire dataset is

$$p(D_{\text{train}}; w, b) = \prod_{i=1}^n p(y^i | x^i; w, b).$$

- 9 Finding good values of w, b can now be thought of as finding values that
- 10 maximize the likelihood of our observed data

$$w^*, b^* = \underset{w, b}{\operatorname{argmin}} -\log p(D_{\text{train}}; w, b). \quad (2.6)$$

- 11 Observe that our objective is written as the minimization of the *negative*
- 12 *log-likelihood*. This is equivalent to maximizing the likelihood because

🔗 Can you think any other sources of noise?
For instance, if you scraped some images
from the Internet, how will you label them?

1 logarithm is monotonic function. We can now rewrite the objective as

$$-\log p(D_{\text{train}}; w, b) = \frac{n}{2} \log(\sigma_\epsilon^2) + \frac{n}{2} \log(2\pi) + \frac{1}{2\sigma_\epsilon^2} \sum_{i=1}^n (y^i - w^\top x^i - b)^2.$$

2 Notice that only the third term depends on w, b . The first term is a function
3 of our *chosen* value σ_ϵ^2 , the second term is a constant. In other words,
4 finding maximizing the likelihood boils down to solving the optimization
5 problem

$$w^*, b^* = \underset{w, b}{\operatorname{argmin}} \frac{1}{2\sigma_\epsilon^2} \sum_{i=1}^n (y^i - w^\top x^i - b)^2. \quad (2.7)$$

6 This objective is nothing other than our least squares regression objective
7 with σ_ϵ^2 set to 1. This objective known as the maximum likelihood
8 objective (MLE).

9 Maximum likelihood objective has an interesting offshoot. In the least
10 squares case, given an input x , all that our fitted model could predict was

$$\hat{y} = w^{*\top} x + b^*.$$

11 MLE has helped us fit a statistical model to the data. So we can now
12 predict the entire probability distribution

$$p(y | x; w^*, b^*) = N(w^{*\top} x + b^*, \sigma_\epsilon^2).$$

13 The solution of least squares is the mean of the Gaussian random variable

$$y | x; w^*, b^*$$

14 the variance of this random variable is σ_ϵ^2 . So instead of just predicting
15 \hat{y} , the machine learning model can now give the probability distribution
16 $p(y | x, w^*, b^*)$ as the output and the user is free to use it as they wish,
17 e.g., compute the mean, the median, the 5% probability value of the right
18 tail etc.

19 2.3 Perceptron

20 Let us now solve a classification problem. We will again go around
21 the model selection problem and consider the class of linear classifiers.
22 Assume binary labels $Y \in \{-1, 1\}$. To keep the notation clear, we will
23 use the trick of appending a 1 to the data x and hide the bias term b in the
24 linear classifier. The predictor is now given by

$$\begin{aligned} f(x; w) &= \operatorname{sign}(w^\top x) \\ &= \begin{cases} +1 & \text{if } w^\top x \geq 0 \\ -1 & \text{else.} \end{cases} \end{aligned} \quad (2.8)$$

🔗 How does using a different value of σ_ϵ in (2.7) change the least squares solution in (2.4)?

🔗 Is a linear model appropriate if our data consisted of natural images? What properties have we lost by restricting the classifier to be linear?

⚠️ The linear classifier remains unchanged if we reorder the pixels of all images consistently in our entire training set and the weights w . The images will look nothing like real images to us. The perceptron does not care about which pixels in the input are close to which others.

We have used the sign function denoted as “sign” to get binary $\{-1, +1\}$ outputs from our real-valued prediction $w^\top x$. This is the famous perceptron model of Frank Rosenblatt. We can visualize the perceptron the same way as we did for linear regression.

Let us now formulate an objective to fit/train the perceptron. As usual, we want the predictions of the model to match those in the training data.

$$\ell_{\text{zero-one}}(w) := \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{\{y^i \neq f(x^i; w)\}}. \quad (2.9)$$

The indicator function inside the summation measures the number of mistakes that the perceptron makes on the training dataset. The objective is thus designed to find w that minimizes the average number of mistakes, also known as the *training error*. Such a loss that incurs a penalty of 1 for a mistake and zero otherwise is called the “zero-one loss”.

2.3.1 Surrogate Losses

The zero-one loss is the clearest indication of whether the perceptron is working well. It is however non-differentiable, so we cannot use powerful ideas from optimization theory to minimize it and find w^* . This is why surrogate losses are constructed in machine learning. These are proxies for the actual loss function that we wish to minimize (the number of mistakes in classification problems). The key property that we desire from a surrogate loss is that a small surrogate loss should imply fewer mistakes for the classifier.

The hinge loss is one such surrogate loss. It is given by

$$\ell_{\text{hinge}}(w) = \max(0, -y w^\top x).$$

If the predicted label $\hat{y} = \text{sign}(w^\top x)$ has the same sign as that of the true label y , then the hinge-loss is zero. If they have opposite signs, the hinge loss increases linearly. The exponential loss

$$\ell_{\text{exp}}(w) = e^{-y (w^\top x)}$$

or the logistic loss

$$\ell_{\text{logistic}}(w) = \log \left(1 + e^{-y w^\top x} \right)$$

are some other popular surrogate losses for classification.

2.4 Stochastic Gradient Descent

We will now fit a perceptron using the hinge loss with a very simple optimization technique. At each iteration, this algorithm updates the weights w in the direction of the negative gradient. So first, let us compute

❓ Can you think of some quantity other than the zero-one loss that we may wish to optimize?

❓ Draw the three losses and observe their differences.

❓ There are also instances when we may want to use surrogate losses for regression, can you think of some?

❓ You may have seen the hinge loss written as $\ell_{\text{hinge}}(w) = \max(0, 1 - y w^\top x)$. Why?

1 the gradient of the hinge loss. It is easily seen to be

$$\frac{d\ell_{\text{hinge}}(w)}{dw} = \begin{cases} -y x & \text{for incorrect prediction on } x, \\ 0 & \text{else.} \end{cases} \quad (2.10)$$

2 We will use a very naive algorithm, called the perceptron algorithm, to
3 update the weights using this gradient.

The Perceptron algorithm Perform the following steps for iterations $t = 1, 2, \dots$

1. At the t^{th} iteration, sample a datum with index $\omega_t \in \{1, \dots, n\}$ from D_{train} uniformly randomly, call it $(x^{\omega_t}, y^{\omega_t})$.
2. Update the weights of the perceptron as

$$w^{t+1} = \begin{cases} w^t + y^{\omega_t} x^{\omega_t} & \text{if } \text{sign}(w^{(t)\top} x^{\omega_t}) \neq y^{\omega_t} \\ w^t & \text{else.} \end{cases} \quad (2.11)$$

4 Observe that a mistake happens if $w^{(t)\top} x^{\omega_t}$ and y^{ω_t} are of different
5 signs, i.e., their product $y^{\omega_t} w^{\top} x^{\omega_t}$ is negative. The perceptron's weights
6 are changed only if it makes a mistake on the datum $(x^{\omega_t}, y^{\omega_t})$. The update
7 to the weights is such that it improves the prediction of the perceptron
8 on this sample. We can see this as follows. The updated weights of the
9 perceptron for the latest sample satisfy the following identity.

$$\begin{aligned} y^{\omega_t} (w^{(t)} + y^{\omega_t} x^{\omega_t})^{\top} x^{\omega_t} &= y^{\omega_t} \langle w^{(t)}, x^{\omega_t} \rangle + (y^{\omega_t})^2 \langle x^{\omega_t}, x^{\omega_t} \rangle \\ &= y^{\omega_t} \langle w^{(t)}, x^{\omega_t} \rangle + \|x^{\omega_t}\|_2^2. \end{aligned}$$

10 In simple words, the value of $y^{\omega_t} \langle w, x^{\omega_t} \rangle$ increases as a result of the
11 update, it becomes more positive. If the perceptron makes mistakes on
12 the same datum repeatedly, this value will eventually become positive. Of
13 course, mistakes on other data in the training set may steer the perceptron
14 towards other directions and it may continue to cycle ad infinitum. It is
15 easy to show that the above algorithm ceases its updates when all data are
16 correctly classified. More precisely, if the training data are such that they
17 can be correctly classified using a linear predictor, then the perceptron
18 will find this predictor after a finite number of iterations.

19 It turns out that we have just seen one of the most powerful algorithms
20 in machine learning. This algorithm is called stochastic gradient descent
21 (SGD) and it is very general: so long as you can take the gradient of an
22 objective, you can execute SGD. The algorithm for fitting the perceptron
23 above was given by Rosenblatt in 1957 and is popularly known as the
24 “perceptron algorithm”. The way we developed it, the perceptron algorithm
25 is simply SGD for the hinge loss. SGD-like algorithms were known in the

1 optimization literature long before 1957 (Robbins and Monro, 1951).

2 2.4.1 The general form of SGD

3 SGD is a very general algorithm. We can use it so long as you have
4 a dataset and an objective that is differentiable. The purpose of the
5 following section is to introduce some basic notation regarding SGD and
6 optimization that we will use in the following lectures.

7 Consider an optimization problem

$$w^* = \operatorname{argmin}_w \frac{1}{n} \sum_{i=1}^n \ell^i(w)$$

8 where the function ℓ^i denotes the loss on the sample (x^i, y^i) and $w \in \mathbb{R}^p$
9 denotes the weights. Solving this problem using SGD corresponds to
10 iteratively updating the weights using

$$w^{(t+1)} = w^{(t)} - \eta \left. \frac{d\ell^{\omega_t}(w)}{dw} \right|_{w=w^{(t)}}.$$

11 The index of the sample in the training set over which we compute the
12 gradient is ω_t . This is a random variable

$$\omega_t \in \{1, \dots, n\}.$$

13 The gradient of the loss $\ell^{\omega_t}(w)$ with respect to w is denoted by

$$\begin{aligned} \nabla \ell^{\omega_t}(w^{(t)}) &:= \left. \frac{d\ell^{\omega_t}(w)}{dw} \right|_{w=w^{(t)}} \\ &= \begin{bmatrix} \nabla_{w_1} \ell^{\omega_t}(w^{(t)}) \\ \nabla_{w_2} \ell^{\omega_t}(w^{(t)}) \\ \vdots \\ \nabla_{w_p} \ell^{\omega_t}(w^{(t)}) \end{bmatrix} \\ &\in \mathbb{R}^p. \end{aligned}$$

14 The gradient $\nabla \ell^{\omega_t}(w^{(t)})$ is therefore a vector in \mathbb{R}^p . We have written

$$\nabla_{w_1} \ell^{\omega_t}(w^{(t)}) = \left. \frac{d\ell^{\omega_t}(w)}{dw_1} \right|_{w=w^{(t)}}$$

15 for the scalar-valued derivative of the objective $\ell^{\omega_t}(w^{(t)})$ with respect to
16 the first weight $w_1 \in \mathbb{R}$. We can therefore write SGD as

$$w^{(t+1)} = w^{(t)} - \eta \nabla \ell^{\omega_t}(w^{(t)}). \quad (2.12)$$

17 The non-negative scalar $\eta \in \mathbb{R}_+$ is called the step-size or the learning rate.
18 It governs the distance traveled along the negative gradient $-\nabla \ell^{\omega_t}(w^{(t)})$
19 at each iteration.

Chapter 3

Kernels, Beginning of neural networks

Reading

1. Bishop 6.1-6.3
2. Goodfellow 6.1-6.4
3. “Random features for large-scale kernel machines” by [Rahimi and Recht \(2008\)](#).

3.1 Digging deeper into the perceptron

3.1.1 Convergence rate

How many iterations does a perceptron need to fit on a given dataset? We will assume that the training data are bounded, i.e., $\|x^i\| \leq R$ for some R and for all $i \in \{1, \dots, n\}$. Let us also assume that the training dataset is indeed linearly separable, i.e., a solution w^* exists for the perceptron weights with training error exactly zero. This means

$$y^i w^{*\top} x^i > 0 \quad \forall i.$$

We will also assume that this classifier *separates the data well*. Note that the distance of each input x^i from the decision boundary (i.e., all x such that $w^{*\top} x = 0$) is given by the component of x^i in the direction of w^* if the label is $y^* = +1$ and in the direction $-w^*$ if the label is negative. In other words,

$$\frac{y^i w^{*\top} x^i}{\|w^*\|} = \rho^i$$

🔗 Draw a picture to convince yourself that the distance to the boundary is indeed ρ .

1 gives the distance to the decision boundary. The quantity on the right hand
 2 side is called the *margin*, it is simply the distance of the sample i from the
 3 decision boundary. If w^* is the classifier with the largest average margin,

$$\rho = \min_{i \in \{1, \dots, n\}} \rho^i$$

4 is a good measure of how hard a particular machine learning problem is.

5 You can now prove that after each update of the perceptron the inner
 6 product of the current weights with the true solution $\langle w_t, w^* \rangle$ increases at
 7 least linearly and that the squared norm $\|w_t\|^2$ increases at most linearly
 8 in the number of updates t . Together, the two will give you a result that
 9 after t weight updates

$$t \leq \frac{R^2}{\rho^2} \quad (3.1)$$

10 all training data are classified correctly. Notice a few things about this
 11 expression.

- 12 1. The quantity $\frac{R^2}{\rho^2}$ is dimension independent; that the number of steps
 13 reach a given accuracy is independent of the dimension of the data
 14 will be a property shared by optimization algorithms in general.
- 15 2. There are no constant factors, this is also the worst case number of
 16 updates; this is quite rare and we cannot get similar results usually.
- 17 3. We can think of the quantity R^2/ρ^2 as a measure of the difficult of
 18 the problem. The number of updates scales with the difficulty; if
 19 the margin ρ was small, we need lots of updates to drive the training
 20 error to zero.
- 21 4. This formula also provides some insight into generalization. Let us
 22 imagine that our assumptions are valid to the data distribution that
 23 creates the training and test sets, namely bounded input $\|x\| \leq R$
 24 and some non-zero margin ρ for the true solution $yw^{*\top}x \geq \rho\|w^*\|$.
 25 Imagine a simpler training procedure that does not reuse any data,
 26 i.e., the perceptron looks at each sample only once, updates the
 27 weights and then throws away the sample before getting a new
 28 one. The number of mistakes that the perceptron makes before it
 29 starts classifying new data correctly is also exact R^2/ρ^2 . So once
 30 these two assumptions are satisfied, we obtain a nice guarantee of
 31 generalization. It is very important to realize that such a perceptron
 32 needs to know when it has made a mistake.

33 3.1.2 Dual representation

34 Let us see how the parameters of the perceptron look after training on the
 35 entire dataset. At each iteration, the weights are updated in the direction
 36 (x^t, y^t) or they are not updated at all. Therefore, if α^i is the number of
 37 times the perceptron sampled the datum (x^i, y^i) during the course of its

▲ If the perceptron makes a mistake in the t^{th} update,

$$\begin{aligned} \langle w_t, w^* \rangle &= \langle w_{t-1} + y^{\omega_t} x^{\omega_t}, w^* \rangle \\ &\geq \langle w_{t-1}, w^* \rangle + \rho \|w^*\| \\ \Rightarrow \langle w^*, w_t \rangle &\geq t\rho \|w^*\|. \end{aligned}$$

Now,

$$\begin{aligned} \|w_t\|^2 &= \|w_{t-1} + y^{\omega_t} x^{\omega_t}\|^2 \\ &= \|w_{t-1}\|^2 + 2 \langle w_{t-1}, y^{\omega_t} x^{\omega_t} \rangle + \|x^{\omega_t}\|^2 \\ &\leq \|w_{t-1}\|^2 + \|x^{\omega_t}\|^2 \\ &\leq \|w_{t-1}\|^2 + R^2. \\ \Rightarrow \|w_t\|^2 &\leq tR^2. \end{aligned}$$

The cosine distance between w_t and w^* is

$$\begin{aligned} 1 \geq \cos(w_t, w^*) &= \frac{\langle w_t, w^* \rangle}{\|w^*\| \|w_t\|} \\ &\geq \frac{t\rho}{R\sqrt{t}} \\ \Rightarrow t &\geq \frac{R^2}{\rho^2}. \end{aligned}$$

1 training and got it wrong, we can write the weights of the perceptron as

$$w^* = \sum_{i=1}^n \alpha^i y^i x^i + w^{(0)}. \quad (3.2)$$

2 where $\alpha^i \in \{0, 1, \dots\}$ and $w^{(0)}$ is the initial weight configuration of the
 3 perceptron. Let us assume that $w^{(0)} = 0$ for the following discussion.
 4 The perceptron therefore is using the classifier

$$\begin{aligned} f(x, w) &= \text{sign}(\hat{y}) \\ \text{where } \hat{y} &= \left(\sum_{i=1}^n \alpha^i y^i x^i \right)^\top x \\ &= \sum_{i=1}^n \alpha^i y^i x^i \top x. \end{aligned} \quad (3.3)$$

5 Remember this special form: the inner product of the new input x
 6 with all the other inputs x^i in the training dataset is combined linearly to
 7 get the prediction. The weights of this linear combination are the dual
 8 variables which measure of how many tries it took the perceptron to fit
 9 that sample during training.

10 3.2 Creating nonlinear classifiers from linear 11 ones

12 Linear classifiers such as the perceptron, or the support vector machine
 13 (SVM) can be extended to nonlinear ones. The trick is essentially the
 14 same that we use when we fit polynomials (polynomials are nonlinear)
 15 using the formula for linear regression. We are interested in mapping
 16 input data x to some different space, this is (usually) a higher-dimensional
 17 space called the *feature space*.

$$x \mapsto \phi(x).$$

18 The quantity $\phi(x)$ is called a feature vector.

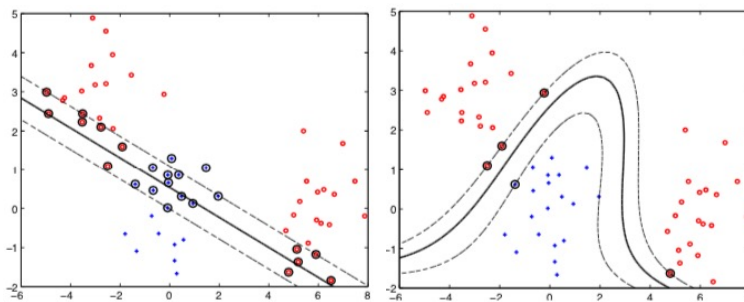


Figure 3.1

▲ As you see in (3.3), computing the prediction for a new input x involves, either remembering all the weights w at the end of training, or storing all the $\{\alpha^i\}_{i=1, \dots, n}$ along with the training dataset. The latter is called the dual representation of a perceptron and the scalars $\{\alpha^i\}$ are called the dual parameters.

1 For example, in the polynomial regression case for scalar input data
2 $x \in \mathbb{R}$ we used

$$\phi(x) := [1, \sqrt{2}x, x^2]^\top$$

3 to get a quadratic feature space. The role of $\sqrt{2}$ will become clear shortly.
4 Certainly, this trick of creating polynomial features also works for higher
5 dimensional input

$$\phi(x) := [1, x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2]^\top.$$

6 Having fixed a feature vector $\phi(x)$, we can now fit a linear perceptron on
7 the input data $\{\phi(x^i), y^i\}$. This involves updating the weights at each
8 iteration as

$$w^{(t+1)} = \begin{cases} w^{(t)} + y^t \phi(x^t) & \text{if } \text{sign}(w^{(t)\top} \phi(x^t)) \neq y^t \\ w^{(t)} & \text{else.} \end{cases} \quad (3.4)$$

9 At the end of such training, the perceptron is

$$w^* = \sum_{i=1}^n \alpha^i y^i \phi(x^i)$$

10 and predictions are made by first mapping the new input to our feature
11 space

$$f(x; w) = \text{sign} \left(\sum_{i=1}^n \alpha^i y^i \phi(x^i)^\top \phi(x) \right). \quad (3.5)$$

12 Notice that we now have a linear combination of the *features* $\phi(x^i)$, not
13 the data x^i , in our formula to compute the output.

14 3.3 Kernels

15 Observe the expression of the classifier in (3.5). Each time we make
16 predictions on the new input, we need to compute n inner products of the
17 form

$$\phi(x^i)^\top \phi(x).$$

18 If the feature dimension is high, we need to enumerate the large number
19 of feature dimensions if we are using the weights of the perceptron, or
20 these inner products if we are using the dual variables. Observe however
21 that even if the feature vector is large, we can compactly evaluate the inner
22 product

$$\begin{aligned} \phi(x) &= [1, \sqrt{2}x, x^2] \\ \phi(x') &= [1, \sqrt{2}x', x'^2] \\ \phi(x)^\top \phi(x') &= 1 + 2xx' + (xx')^2 = (1 + xx')^2. \end{aligned}$$

🔍 The concept of a feature space seems like a panacea. If we have complex data, we simply map it to some high-dimensional feature and fit a linear function to these features.

However, the “curse of dimensionality” coined by Richard Bellman states that to fit a function in \mathbb{R}^d the number of samples needs to be exponential in d . It therefore stands to reason that we need a lot more data to fit a classifier in feature space than in the original input space. Why would we still be interested in the feature space then?

⚠️ Feature spaces can become large very quickly. What is the dimensionality of $\phi(x)$ for a tenth-order polynomial with a three-dimensional input data? Look up something called as “triangular number” (https://en.wikipedia.org/wiki/Trinomial_expansion)

1 for input $x \in \mathbb{R}$. Kernels are a formalization of this idea. A kernel

$$k : X \times X \rightarrow \mathbb{R}.$$

2 is any symmetric, positive semi-definite function with two arguments such
3 that

$$k(x, x') = \phi(x)^\top \phi(x')$$

4 for some feature ϕ for all x, x' . Few examples of kernels are

$$k(x, x') = (x^\top x' + c)^2,$$

$$k(x, x') = \exp\left(-\|x - x'\|^2 / (2\sigma^2)\right).$$

5 3.3.1 Kernel perceptron

6 We can now give the kernel version of the perceptron algorithm. The
7 idea is to simply replace any inner product in the algorithm that looks like
8 $\phi(x)^\top \phi(x')$ by the kernel $k(x, x')$.

Kernel perceptron Initialize dual variables $\alpha^i = 0$ for all $i \in \{1, \dots, n\}$. Perform the following steps for iterations $t = 1, 2, \dots$

1. At the t^{th} iteration, sample a data point with index ω_t from D_{train} uniformly randomly, call it $(x^{\omega_t}, y^{\omega_t})$.
2. If there is a mistake, i.e., if

$$0 \geq y^{\omega_t} \left(\sum_{i=1}^n \alpha^i y^i \phi(x^i)^\top \phi(x^{\omega_t}) \right)$$

$$= y^{\omega_t} \left(\sum_{i=1}^n \alpha^i y^i k(x^i, x^{\omega_t}) \right),$$

then update

$$\alpha^{\omega_t} \leftarrow \alpha^{\omega_t} + 1.$$

9 Notice that we do not ever compute $\phi(x)$ so it does not matter what
10 the dimensionality of the feature vector is. It can even be infinite, e.g., for
11 the radial basis function kernel. Observe also that we do not maintain
12 weights w . We instead maintain the dual variables $\{\alpha^1, \dots, \alpha^n\}$ while
13 running the algorithm.

14 Note that the kernel perceptron computes the kernel over *all* data
15 samples in the training set at each iteration. It is expensive and seems
16 wasteful. The Gram matrix denoted by $G \in \mathbb{R}^{n \times n}$

$$G_{ij} = k(x^i, x^j) \tag{3.6}$$

17 helps address this problem by computing the kernel on all pairs in the

🔗 Kernels look great, e.g., you can fit perceptrons in powerful feature spaces using essentially the same algorithm. How expensive is each iteration of the perceptron?

⚠️ When ML algorithms are implemented in a system, there exist tradeoffs between the feature-space version and the Gram matrix version of linear classifiers. The former is preferable if the number of samples in the dataset is large, while the latter is used when the dimensionality of features is large.

🔗 Logistic regression with a loss function

$$\ell_{\text{logistic}}(w) = \log\left(1 + e^{-yw^\top x}\right)$$

is also a linear classifier. Write down how you will fit a logistic regression using stochastic gradient descent; this is similar to the perceptron algorithm. Write down the feature-space version of the algorithm and a kernelized logistic regression that uses the Gram matrix.

1 training dataset. We can now write step 2 in the kernel perceptron

$$y^t \left(\sum_{i=1}^n \alpha^i y^i k(x^i, x^t) \right) = y^t (\alpha \odot Y)^\top G e_t.$$

2 where $e_t = [0, \dots, 0, 1, 0, \dots]$ with a 1 on the t^{th} element, $\alpha = [\alpha^1, \dots, \alpha^n]$
 3 denotes the vector of all the dual variables, $Y = [y^1, \dots, y^n]$ is a vector
 4 of all the labels, and the notation $\alpha \odot Y = [\alpha^1 y^1, \dots, \alpha^n y^n]$ denotes
 5 the element-wise (Hadamard) product. This expression now only involves
 6 a matrix-vector multiplication, which is much easier than computing
 7 the kernel at each iteration. Gram matrices can become very big. If
 8 the number of samples is $n = 10^6$, not an unusual number today, the
 9 Gram matrix has 10^{12} elements. The big failing of kernel methods is
 10 that they require a large amount of memory at training time. Nystrom
 11 methods compute low-rank approximations of the Gram matrix which
 12 makes operations with kernels easier.

13 3.3.2 Mercer's theorem

14 This theorem shows that any kernel that satisfies some regularity properties
 15 can be rewritten as an inner product in some feature space.

16 **Theorem 3.1 (Mercer's Theorem).** For any symmetric function k :
 17 $X \times X \rightarrow \mathbb{R}$ which is square integrable in $X \times X$ and satisfies

$$\int_{X \times X} k(x, x') f(x) f(x') dx dx' \geq 0 \quad (3.7)$$

18 for all square integrable functions $f \in L_2(X)$, there exist functions
 19 $\phi_i : X \rightarrow \mathbb{R}$ and numbers $\lambda_i \geq 0$ where

$$k(x, x') = \sum_{i=1}^{\infty} \lambda_i \phi_i^\top(x) \phi_i(x')$$

20 for all $x, x' \in X$. The condition in (3.7) is called Mercer's condition.
 21 You will also have seen Mercer's condition written as follows: "for any
 22 finite set of inputs $\{x^1, \dots, x^n\}$ and any choice of real-valued coefficients
 23 c_1, \dots, c_n a valid kernel should satisfy

$$\sum_{i,j} c_i c_j k(x^i, x^j) \geq 0."$$

24 There can be an infinite number of coefficients λ_i in the summation.

25 **Remark 3.2 (Checking if a function is a valid kernel).** Note that
 26 Mercer's condition states that the Gram matrix of any dataset is positive
 27 semi-definite:

$$u^\top G u \geq 0 \quad \text{for all } u \in \mathbb{R}^n. \quad (3.8)$$

💡 A function $f : X \rightarrow \mathbb{R}$ is square integrable iff

$$\int_{x \in X} |f(x)|^2 dx < \infty.$$

💡 We can think of a function $f(x)$ as a long vector with one entry for each $x \in X$. The integral in Theorem 3.1 in Mercer's condition is analogous to a vector-matrix-vector multiplication like $u^\top G u$.

1 This is easy to show.

$$\begin{aligned}
 u^\top G u &= \sum_{ij=1}^n u_i u_j G_{ij} \\
 &= \sum_{ij} u_i u_j \left(\sum_{k=1}^{\infty} \lambda_k \phi_k(x^i)^\top \phi_k(x^j) \right) \\
 &= \sum_{k=1}^{\infty} \lambda_k \left(\sum_{ij} u_i u_j \phi_k(x^i)^\top \phi_k(x^j) \right) \\
 &= \sum_{k=1}^{\infty} \lambda_k \left(\sum_i u_i \phi_k(x^i) \right)^\top \left(\sum_j u_j \phi_k(x^j) \right) \\
 &= \sum_{k=1}^{\infty} \lambda_k \left\| \sum_i u_i \phi_k(x^i) \right\|^2 \\
 &\geq 0.
 \end{aligned}$$

2 On the second line, we have expanded the term $G_{ij} = k(x^i, x^j) =$
 3 $\sum_k \lambda_k \phi_k(x^i)^\top \phi_k(x^j)$ using Mercer's condition. So if you have a function
 4 that you would like to use as a kernel, checking its validity is easy by
 5 showing that the Gram matrix is positive semi-definite.

6 Kernels are powerful because they do not require you to think of the
 7 feature and parameter spaces. For instance, we may wish to design a
 8 machine learning algorithm for spam detection that takes in a variable
 9 length of feature vector depending on the particular input. If $x[i]$ is the i^{th}
 10 character of a string, a good way to build a feature vector is to consider
 11 the set of all length k sub-sequences. The number of components in this
 12 feature vector is exponential. However, as you can imagine, given two
 13 strings x, x'

14 this string is interesting
 15 txws sbhtqg is atso iyubqtnhpqg

16 you can write a Python function to check their similarities with respect
 17 to some rules *you define*, e.g., a small edit distance between the strings.
 18 Mercer's theorem is useful here because it says that so long as your
 19 function satisfies the basic properties of a kernel function, there exists
 20 some feature space which your Python function implicitly constructs.

21 3.4 Learning the feature vector

The central idea behind deep learning is to learn the feature vectors ϕ instead of choosing them *a priori*.

22 How do we choose what set of feature vectors to learn from? For instance,
 23 we could pick all polynomials; we could pick all possible Gabor filters

❓ Checking your Python function for whether it is a good kernel is great using (3.8). Can you think of a situation when you can get a wrong answer using this approach, i.e., your kernel is not a legitimate kernel but (3.8) says that it is?

❓ These are two different images of related concepts, what feature space can we use to say that they are similar?



1 that you saw in HW 1; we could also pick all possible string kernels.

2 3.4.1 Random features

3 Suppose that we have a finite-dimensional feature $\phi(x) \in \mathbb{R}^p$. We saw in
4 the perceptron that

$$f(x; w) = \text{sign} \left(\sum_i w_i \phi_i(x) \right)$$

5 where $\phi(x) = [\phi_1(x), \dots, \phi_p(x)]$ and $w = [w_1, \dots, w_p]$ are the feature
6 and weight vectors respectively. We will set

$$\phi(x) = \sigma(S^\top x), \quad (3.9)$$

7 where $S \in \mathbb{R}^{d \times p}$ is a matrix. The function $\sigma(\cdot)$ is a nonlinear function of
8 its argument and acts on all elements of the argument element-wise

$$\sigma(z) = [\sigma(z_1), \dots, \sigma(z_p)]^\top.$$

9 We will abuse notation and denote both the vector version of σ and the
10 element-wise version of σ using the same Greek letter.

11 Notice that this is a special type of feature vector (or a special type of
12 kernel), it is a linear combination of the input elements. What matrix S
13 should we pick to combine these input elements? The paper by [Rahimi
14 and Recht \(2008\)](#) proposed the idea that for shift-invariant kernels (which
15 have the property that $k(x, x') \equiv k(x - x')$ one may use a matrix with
16 random elements as our S

$$S^\top = \begin{bmatrix} \omega_1^\top \\ \vdots \\ \omega_p^\top \end{bmatrix}$$

17 where $\omega_i \in \mathbb{R}^d$ are random variables drawn from, say, a Gaussian
18 distribution and

$$\sigma(z) = \cos(z).$$

19 Using a random matrix is a cheap trick, it lets us create a lot of features
20 quickly without worrying about their quality. Our classifier is now

$$f(x; w) = \text{sign}(w^\top \sigma(S^\top x)) \quad (3.10)$$

21 and we can again solve the optimization problem

$$w^* = \underset{w}{\text{argmin}} \frac{1}{n} \sum_{i=1}^n \ell_{\text{hinge}}(y^i, \hat{y}^i; w) \quad (3.11)$$

22 with $\hat{y}^i = w^\top \sigma(S^\top x^i)$ and fit the weights w using SGD as before.

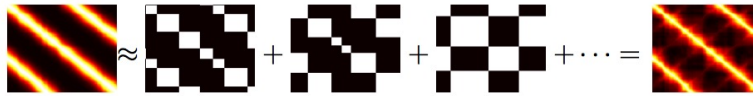


Figure 3.2

1 As an example consider the heatmap of Gabor-like kernel $k(x, x')$
 2 in Figure 3.2 on the left. We can think of the decomposition

$$\begin{aligned}
 & \text{pixel } (i, j) \text{ of the left-most picture} \\
 &= k(x^i, x^j) \\
 &= \phi(x^i)^\top \phi(x^j) \\
 &= \sum_{k=1}^p \sigma(\omega_k^\top x^i) \sigma(\omega_k^\top x^j) \quad (\text{each black-white matrix}) \\
 &= \text{pixel } (i, j) \text{ in the right-most picture.}
 \end{aligned}$$

3 In other words, the p random elements of the matrix S , namely ω_k come
 4 together to give us a useful kernel on the left. A large random matrix S
 5 has many such terms on the right hand-side.

6 3.4.2 Learning the feature matrix as well

7 Random features do not work well for all kinds of data. For instance, if
 8 you have an image of size 100×100 , and you are trying to find a fruit



9
 10 we can design random features of the form

$$\phi_{ij,kl} = \mathbf{1}_{\{\text{mostly red color in a box formed by pixels } (ij) \text{ and } (kl)\}}.$$

11 We will need lots and lots of such features before we can design an object
 12 detector that works well for this image. In other words, random features
 13 do not solve the problem that you need to be clever about picking your
 14 feature space/kernel.

Simply speaking, deep learning is about learning the matrix S

🔗 What kind of data do you think random features will work well for?

in (3.10) in addition to the coefficients w . The classifier now is

$$f(x; w, S) = \text{sign}(w^\top \sigma(S^\top x)) \quad (3.12)$$

but we now solve the optimization problem

$$w^*, S^* = \underset{w, S}{\text{argmin}} \frac{1}{n} \sum_{i=1}^n \ell_{\text{hinge}}(y^i, \hat{y}^i) \quad (3.13)$$

with $\hat{y}^i = w^\top \sigma(S^\top x^i)$ as before. This is our first *deep network*, (3.12) is a two-layer neural network.

1 Moving from the problem in (3.11) to this new problem in (3.13) is a
2 very big change.

3 1. **Nonlinearity.** The classifier in (3.12) is not linear anymore. It is a
4 nonlinear function of its parameters w, S (both of which we will
5 call weights).

6 2. **High-dimensionality.** We added a lot more weights to the classifier,
7 the original classifier had $w \in \mathbb{R}^p$ parameters to learn while the new
8 one also has $S \in \mathbb{R}^{d \times p}$ more weights. The curse of dimensionality
9 suggests that we will need a lot more data to fit the new classifier.

10 3. **Non-convex optimization.** The optimization problem in (3.13)
11 much harder than the one in (3.11). The latter is a convex function
12 (we will discuss this soon) which are easy to minimize. The former
13 is a non-convex function in its parameters w, S because they interact
14 multiplicatively, such functions are harder to minimize. We could
15 write down the solution of the perceptron using the final values of
16 the dual variables. We cannot do this for a two-layer neural network.

Chapter 4

Deep fully-connected networks, Backpropagation

Reading

1. Bishop 5.1, 5.3
2. Goodfellow 6.3-6.5
3. Notes at <http://cs231n.github.io/optimization-2/>

4.1 Deep fully-connected networks

A deep neural network takes the idea of a two-layer network to the next step. Instead of having one matrix S in the classifier

$$f(x; v, S) = \text{sign}(v^\top \sigma(S^\top x))$$

a deep network has many matrices S_1, \dots, S_L

$$f(x; v, S_1, \dots, S_L) = \text{sign}(v^\top \sigma(S_L^\top \dots \sigma(S_2^\top \sigma(S_1^\top x) \dots))). \quad (4.1)$$

We will call each operation of the form $\sigma(S_k^\top \dots)$, a *layer*. Consider the second layer: it takes the features generated by the first layer, namely $\sigma(S_1^\top x)$, multiplies these features using its feature matrix S_2^\top and applies a nonlinear function $\sigma(\cdot)$ to this result element-wise before passing it on to the third layer.

A deep network creates new features by composing older features.

This composition is very powerful. Not only do we not have to pick a particular feature vector, we can create very complex features by

1 sequentially combining simpler ones. For example Figure 4.1 shows the
 2 features (more precisely, the kernel) learnt by a deep neural network. The
 3 first layer of features are called Gabor-like, they are similar to ones you
 4 constructed in HW 1. These features are *combined* linearly along with
 5 a nonlinear operation to give richer features (spirals, right angles) in the
 6 middle panel. The third layer combines the lower features to get even
 7 more complex features, these look like patterns (notice a soccer ball in the
 8 bottom left), a box on the bottom right etc.

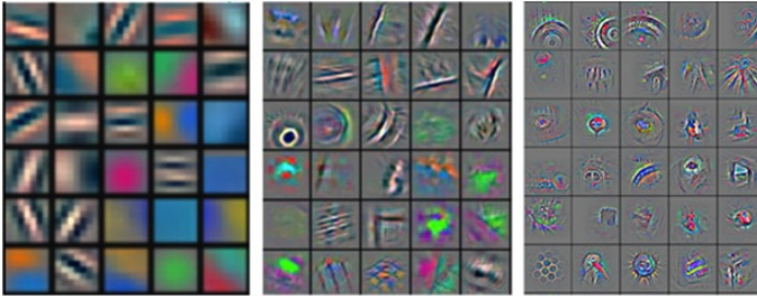


Figure 4.1

9 The optimization problem for fitting a deep network is written as

$$v^*, S_1^*, \dots, S_L^* = \operatorname{argmin}_{v, S_1, \dots, S_L} \frac{1}{n} \sum_{i=1}^n \ell_{\text{hinge}}(y^i, \hat{y}^i) \quad (4.2)$$

10 where the output prediction is

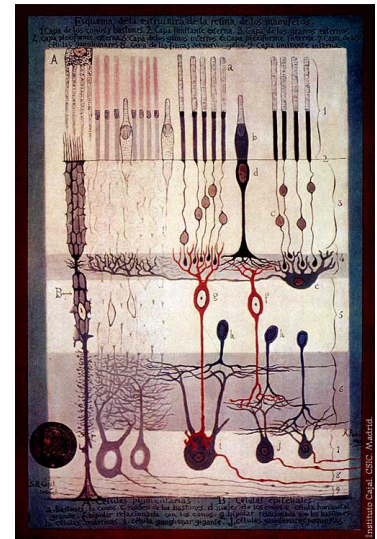
$$\hat{y} = v^\top \sigma(S_L^\top \dots \sigma(S_2^\top \sigma(S_1^\top x)) \dots).$$

11 Notice that if fitting a two-layer network was difficult, then fitting a
 12 multi-layer neural network like (4.1) is even harder. There are *lots* of
 13 parameters and consequently we need a lot more data to fit such a model.
 14 The optimization problem in (4.2) is also naturally much harder than its
 15 two-layer version. The benefit for going through this difficulty is many
 16 fold and quite astounding.

- 17 1. Not having to pick features is very powerful. Notice that we do not
 18 need to worry about what kind of data x is at the input. So long as
 19 we can write it into a vector, the classifier as written in (4.1) works.
 20 In other words, the same type of classifier works for image-based
 21 data, data from natural language processing, speech processing, and
 22 many other types. This is the primary reason why a large number
 23 of scientific fields are adopting deep networks.
- 24 2. Before the resurgence of deep learning, each of these fields essen-
 25 tially had their own favorite kernels they preferred, these kernels
 26 were designed across decades of insights from that specific field
 27 (wavelets in signal processing, keypoint detectors and descriptors
 28 in computer vision, n -grams in NLP etc.). It was very difficult for a
 29 researcher to use ideas from a different field. With deep learning,

▲ The mammalian retina Circuits in the retina are hard-wired at birth because being able to see is so important to survival; there is no learning in the retina itself although there is a clear hierarchy of neurons that successively process information. Later parts of the visual cortex get learned during your lifetime.

The retina transcribes photons that are incident upon the eye using rod cells (function better in low light) and cone cells (function better in bright conditions). This is further processed by “bipolar” cells into action potentials, or “spikes”. Amacrine cells make lateral, inhibitory connections to remove redundancy in the stimuli. Ganglion cells create ~ 20 visual features (edges/spots, local motions at $90^\circ/120^\circ$ angles, colors, etc.). Altogether, ~ 80 types of neurons transmit ~ 10 Mbps of information to the brain. These neurons are surprisingly similar to each other, e.g., all cell types fire at 4-8 Hz and different ganglion cells learn highly redundant features. Read [Balasubramanian \(2015\)](#) for an exciting description of why neural circuits are wired the way they are.



A picture of the neurons in the retina drawn by Santiago Ramón y Cajal using a microscope in the 1900s.

1 this has become much easier. There is still a significant amount of
 2 domain insight that you need to make deep networks work well but
 3 the bar for entering a new field is much lower.

- 4 3. Deep neural networks are universal approximators. In simple words,
 5 it means that provided the deep network has enough number of
 6 layers and enough number of features in each layer, it can fit any
 7 dataset. This is a theorem in approximation theory.

8 4.1.1 Some deep learning jargon

9 We have defined the essential parts of a deep network. Let us briefly take
 10 a look at some typical jargon you will encounter as you read more.

11 **Activation function.** The nonlinear function $\sigma(\cdot)$ in (4.1) is called the
 12 activation function (motivated from the threshold-based activation of
 13 McCulloch-Pitts neuron). It is also called a nonlinearity because it is
 14 the only nonlinear operation in the classifier. There are many activation
 15 functions that have been used over the years.

- 16 1. Threshold

$$\text{threshold}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{else.} \end{cases}$$

- 17 2. Sigmoid/Logistic

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

- 18 3. Hyperbolic tangent

$$\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- 19 4. Rectified Linear Units (ReLU)

$$\begin{aligned} \text{relu}(x) &= |x|_+ \\ &= \max(0, x). \end{aligned}$$

- 20 5. Leaky ReLUs

$$\sigma_c(x) = \begin{cases} x & \text{if } x > 0 \\ c x & \text{else.} \end{cases}$$

- 21 6. Swish

$$\sigma(x) = x \text{ sigmoid}(x).$$

22 Different activation functions work differently. ReLU nonlinearities are
 23 the most popular and we will see the reasons why they work better than
 24 older ones such as sigmoid/tanh nonlinearities in the backpropagation
 25 section.

1 **Logits for multi-class classification.** The output

$$\hat{y} = v^\top \sigma(S_L^\top \dots \sigma(S_2^\top \sigma(S_1^\top x)) \dots)$$

2 are called the logits corresponding to the different classes. This name
3 comes from logistic regression where logits are the log-probabilities of
4 belonging to one of the two classes. A deep network affords an easy way
5 to solve a multi-class classification problem, we simply set

$$v \in \mathbb{R}^{P \times C}$$

6 where C is the total number of classes in the data. Just like logistic
7 regression predicts the logits of the two classes, we would like to *interpret*
8 the vector \hat{y} as the log-probabilities of an input belonging to one of the
9 classes.

10 **Mid-level features.** The features at any layer can be studied once you
11 create a deep network. You pass an input image x and compute

$$h^l = S_l^\top \dots \sigma(S_2^\top \sigma(S_1^\top x)) \dots \quad (4.3)$$

12 to get the *pre-activation* output of the l^{th} layer. The *post-activation* output
13 is given by applying the nonlinearity

$$\sigma(h^l).$$

14 Sometimes people will call the $\sigma(h^L)$ as the *feature* created by a deep
15 network; the rationale here is that just like a kernel-based classifier uses
16 features $\phi(x)$ and fits a linear classifier to these features we may think of
17 the feature of a deep network to be $\sigma(h^L)$. These features are often very
18 useful, e.g., you can use the lower layers of a deep network trained on a
19 different dataset, say classifying cats vs. dogs, as the feature generator but
20 retrain the classifier weights v on your specific problem, say classifying
21 apples vs. oranges. Such pre-training is typically used to exploit the fact
22 that someone else has trained a large deep network on a large dataset, and
23 thereby learnt a rich feature generator. Training the large model yourself
24 on a large dataset like ImageNet would be quite difficult.

25 **Hidden layers/neurons.** The intermediate layers that create the features
26 h^1, \dots, h^L are called the hidden layers. A feature is the same as a neuron;
27 think of the McCulloch-Pitts picture, just like a neuron takes input from all
28 the other neurons connected to it via some weights, a feature is computed
29 using a weighted combination of the features at the lower layer. We will
30 say that a neural network is *wide* if it has lots of features/neurons on each
31 hidden layer. We will say that it is *thin* if it has few features/neurons on
32 each hidden layer.

❓ How would you use a binary classifier to classify 10 classes?

❓ What would the shape of v be if you were performing regression using a deep network?

4.1.2 Weights

It is customary to not differentiate between the parameters of different layers of a deep network and simply say *weights* when we want to refer to all parameters. The set

$$w := \{v, S_1, S_2, \dots, S_L\}$$

is the set of *weights*. This set is typically stored in PyTorch as a set of matrices, one for each layer.

Important. Every time we want to write down mathematical equations, we will imagine w to be a large vector. This is less cumbersome notation. We denote by p the dimensionality of w and imagine that

$$w \in \mathbb{R}^p.$$

The dimensionality p keeps things consistent with linear classifiers where the features were $\phi(x) \in \mathbb{R}^p$. When you use PyTorch to implement an algorithm that requires you to iterate over the weights, say you were implementing SGD from scratch, you will iterate over elements of the set of weights. Using this new notation, we will write down a deep network as simply

$$f(x, w) \tag{4.4}$$

and fitting the deep network to a dataset involves the optimization problem

$$w^* = \operatorname{argmin}_w \frac{1}{n} \sum_{i=1}^n \ell(y^i, \hat{y}^i; w). \tag{4.5}$$

We will often denote the loss of the i^{th} sample as simply

$$\ell^i(w) := \ell(y^i, \hat{y}^i; w).$$

4.2 The backpropagation algorithm

We would like to using SGD to fit a deep network on a given dataset. As we saw in Chapter 2, if the loss function is denoted by $\ell^{\omega_t}(w)$ where ω_t was the index of the datum sampled at iteration t , we would like to update the weights using

$$w^{(t+1)} = w^{(t)} - \eta \left. \frac{d\ell^{\omega_t}(w)}{dw} \right|_{w=w^{(t)}}.$$

We have used a scalar $\eta > 0$ as the step-size or the learning rate. It governs the distance traveled along the negative gradient at each iteration.

- 1 Let us ignore the index of the datum ω_t in this section, imagine $\omega_t = 1$.
 2 Implementing SGD therefore boils down to computing the gradient

$$\frac{d\ell(w)}{dw}.$$

Backpropagation is an algorithm for computing the gradient of the loss function with respect to weights of a deep network.

3 4.2.1 One hidden layer with one neuron

- 4 Consider the linear regression problem with one layer and one datum,
 5 $w, x \in \mathbb{R}^d$ and $v, y \in \mathbb{R}$:

$$\ell(w, v) = \frac{1}{2}(y - v\sigma(w^\top x))^2$$

- 6 where $\sigma(\cdot)$ is some activation function and our weights are $\{v, w\}$. Let us
 7 understand the computational graph of how the loss is computed:

$$w, x \xrightarrow{\text{layer 1}} z \xrightarrow{\text{layer 2}} h \xrightarrow{\text{layer 3}} vh \xrightarrow{\text{layer 4}} \ell. \quad (4.6)$$

- 8 where $h = \sigma(z)$ and $z = w^\top x$. Each node in this graph is either the
 9 input/output or an intermediate result of the computation. The gradient of
 10 the loss with respect to the weights using the chain rule is

$$\frac{\partial \ell}{\partial v} = (y - v\sigma(w^\top x)) (-\sigma(w^\top x)) \quad (4.7)$$

- 11 and

$$\frac{\partial \ell}{\partial w} = (y - v\sigma(w^\top x)) (-v\sigma'(w^\top x)) (x). \quad (4.8)$$

1. **Caching computations for computing the chain rule.** The

first idea behind backpropagation is to realize that quantities like $(y - v\sigma(w^\top x))$ or $z = w^\top x$ are computed multiple times in the chain rule in (4.7) and (4.8). If we can cache these quantities we can compute the chain rule-based gradient for the different parameters quickly.

2. **Cache is the output of each layer.** The second idea behind backpropagation is to realize that quantities like $(y - vh)$, $h = \sigma(z)$ and $z = w^\top x$ are outputs of the third, second and first layers respectively. In other words, the quantities we need to cache in the chain rule computation are simply the outputs of the individual layers.
3. **Derivatives of the loss with respect to the input of a layer only depends on what happens in that layer and the derivative of the loss with respect to the output of that layer.** The third observation is to see that the quantity $\sigma'(z)$ in (4.8) is the derivative of the output of the activation function, namely $h = \sigma(z)$ with respect to z , its input argument

$$\sigma'(z) = \frac{dh}{dz}.$$

This derivative is combined with the forward computation $(y - vh)$ to get the gradient with respect to the weights w .

Backpropagation is simply a book-keeping exercise that caches the forward computation of the graph in (4.6) and uses these cached values to compute the derivative of the loss ℓ with respect to the parameters of each layer sequentially.

- 1 We will use a clever notation to denote the backprop gradient which
- 2 will make this process very easy. Denote by

$$\bar{v} = \frac{d\ell}{dv} \tag{4.9}$$

- 3 the derivative of the loss ℓ with respect to a parameter v . For our simple
- 4 two layer (one neuron) neural network, we are interested in computing the
- 5 quantities

$$\bar{w}, \quad \bar{v}.$$

- 6 Let us also denote the output of the second linear layer (layer 3) as

$$e = vh.$$

1 Now observe the following “forward computation”

$$z = w^\top x \quad (4.10)$$

$$h = \sigma(z) \quad (4.11)$$

$$e = vh \quad (4.12)$$

$$\ell = \frac{1}{2} (y - e)^2. \quad (4.13)$$

2 Let us imagine that we have cached all the quantities on the left hand side
 3 of the equalities above. We use these quantities to perform the “backward”
 4 computation as follows

$$\begin{aligned} \frac{d\ell}{d\ell} &= \bar{\ell} = 1. \\ \mathbb{R} \ni \bar{e} &= \bar{\ell} \frac{d\ell}{de} \\ &= -1 (y - e) = \bar{\ell} (-(y - e)). \quad (\text{from (4.13)}) \\ \mathbb{R} \ni \bar{v} &= \bar{e} \frac{de}{dv} \\ &= -(y - e) h = \bar{e} h. \quad (\text{from (4.12)}) \\ \mathbb{R} \ni \bar{h} &= \bar{e} \frac{de}{dh} \\ &= \bar{e} (v). \quad (\text{from (4.12)}) \\ \mathbb{R} \ni \bar{z} &= \bar{h} \frac{dh}{dz} \\ &= \bar{h} \sigma'(z). \quad (\text{from (4.11)}) \\ \mathbb{R}^d \ni \bar{w} &= \bar{z} \frac{dz}{dw} \\ &= \bar{z} x. \quad (\text{from (4.10)}) \\ \mathbb{R}^d \ni \bar{x} &= \bar{z} \frac{dz}{dx} \\ &= \bar{z} w. \quad (\text{from (4.10)}) \end{aligned}$$

5 **Remark 4.1.** An interesting mnemonic to remember backprop is to see
 6 that if the forward graph is

$$z = w_1 x_1 + w_2 x_2$$

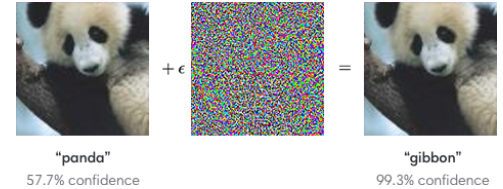
7 the backprop gradient is $\bar{w}_1 = \bar{z} x_1$ and $\bar{w}_2 = \bar{z} x_2$. If x_1 was large and
 8 dominated the computation of z during the forward propagation, then w_1
 9 which is the multiplier of x_1 also gets a dominant share of the backprop
 10 gradient \bar{z} . The backprop gradient is shared equitably among the different
 11 quantities that took part in the forward computation. This is useful to
 12 remember when you build neural networks with complex architectures
 13 on your own: if there is a part of the network whose activations are very
 14 small and it is being combined with another part of the network whose
 15 activations have a large magnitude, then the former is not going to going
 16 to get a large enough backprop gradient.

1 **Remark 4.2 (Gradient with respect to the input x).** Notice that we
 2 obtain the gradient of the loss with respect to the input x

$$\frac{d\ell}{dx}$$

3 as a by-product of backpropagation. Backpropagation computes the
 4 gradient of the input activations to each layer \bar{v} because this is precisely
 5 the gradient that is propagated downwards. So the gradient \bar{x} should not
 6 be surprising, after all x is nothing but the input activation to the first layer.
 7 This gradient is useful, you can use to find what are called adversarial
 8 examples, i.e., input images which look like natural images to us humans
 9 but contain imperceptible noise that gives a large value of \bar{x} .

▲ An example adversarial input to a deep network



10 4.2.2 Implementation of backpropagation

11 Consider our neural network classifier given by

$$f(x; v, S_1, \dots, S_L) = \text{sign} \left(w^\top \sigma \left(S_L^\top \dots \sigma \left(S_2^\top \sigma \left(S_1^\top x \right) \dots \right) \right) \right).$$

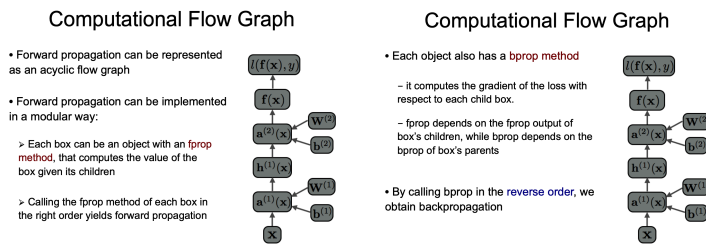


Figure 4.2: Schematic of forward and backward computations in backpropagation.

12 When you build such a multi-layer network in PyTorch, the k^{th} layer
 13 is automatically equipped with two member functions.

```

14 def forward(self, h^{k-1}, S_k):
15     # computes the output of the k^th layer
16     # given output of previous layer h^k and
17     # parameters of current layer S_k
18     return h^k
19
20
21 def backward(self, h^k, d loss/dh^{k}, S_k):
22     # computes two quantities
23     # 1. d loss/d{S_k}
24     # 2. d loss/d{h^{k-1}}
25     return d loss/d{S_k}, d loss/d{h^{k-1}}
26

```

27 Such forward and backward functions exist for every layer, including the
 28 nonlinearities. If you implement a new type of layer in a neural network,
 29 say a new nonlinearity, you only need to write the forward function.
 30 The autograd module inside PyTorch automatically writes the backward
 31 function by looking at the forward function. This is why PyTorch is so
 32 powerful, you can build complex functions inside your deep networks
 33 without having to compute the derivatives yourself.

Chapter 5

Convolutional Architectures

Reading

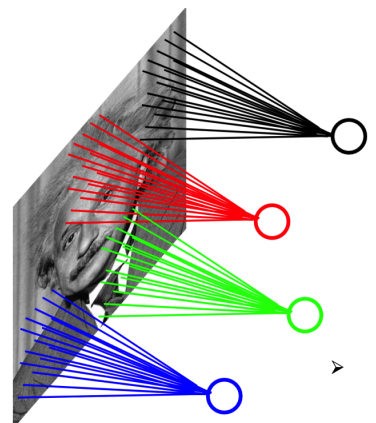
1. Goodfellow 9
2. “Striving for simplicity: The all convolutional net”, by [Springenberg et al., 2014](#)

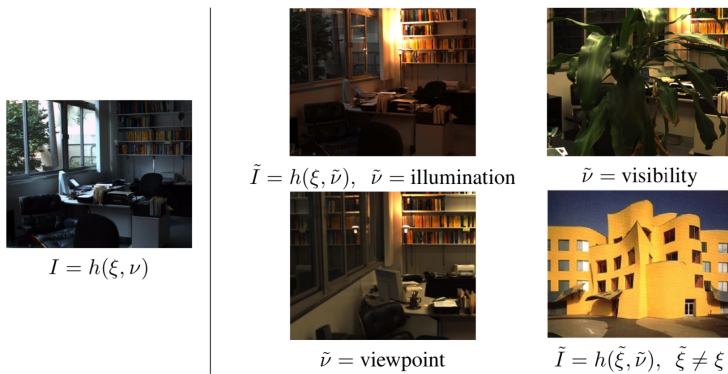
So it turns out that we have been talking about what are called “fully-connected” neural networks in the past chapter. There are a few problems that are apparent even in our limited experience.

Fully-connected layers have a lot of parameters. If an input image is of size $100 \times 100 = 10^4$ grayscale pixels and we would like to classify it as belonging to one out of 1000 classes, we need 10M parameters. It is difficult to perform so many add-multiply operations quickly even on sophisticated GPUs. Further, the curse of dimensionality never goes away; we need lots of data to fit these many parameters.

Natural data is full of “nuisances” that are not useful for tasks such as classification. E.g., illumination, viewpoint, and occlusions

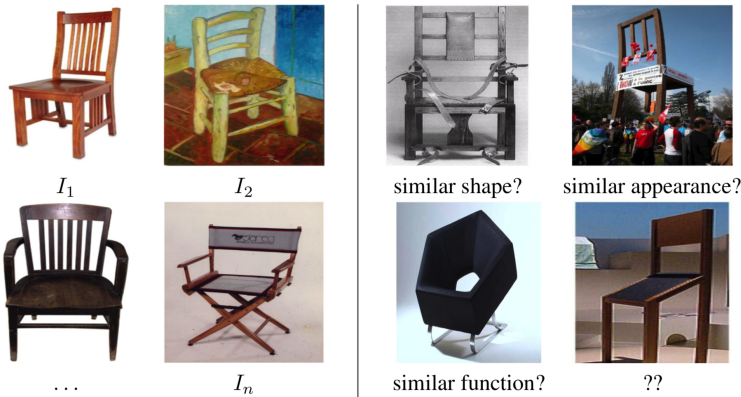
Let us consider an example using local connections instead of a fully-connected layer. If each output neuron is connected to only 25 pixels of the 100×100 image and there are 1000 output neurons, how many weights will this layer have?





1

2 or even semantic ones shown below



3

4 Do fully connected networks work for such different images?

5 Nuisances can be defined as operations that act on the data before you
 6 get to see it (nature creates these nuisances). Some of them are special and
 7 they have a group structure, i.e., they satisfy certain algebraic conditions
 8 [https://en.wikipedia.org/wiki/Group_\(mathematics\)](https://en.wikipedia.org/wiki/Group_(mathematics)). For instance, images
 9 of the same chair taken from different vantage points are projections of
 10 different rigid body transformations of the camera. Some other nuisances
 11 such as occlusions do not have a group structure, e.g., there is no rigid
 12 body transformation that allows us to backcalculate the pixels belonging
 13 to a person standing behind a car. Convolutional layers are a simple way
 14 to tackle one particular kind of nuisance, that of translations.

15 5.1 Basics of the convolution operation

16 So far, we have seen that the basic unit of a neural network is

$$\sigma(w^\top x).$$

17 The basic unit of a convolutional neural network is

$$\sigma(x * w)$$

1 where the $*$ denotes a convolution operation. Consider two one-dimensional
 2 vectors $x \in \mathbb{R}^3$ and $w \in \mathbb{R}^3$; we will imagine these to be arrays of infinite
 3 length with all the entries at indices $[4, \infty)$ set to zero; this is known as
 4 zero-padding the input

$$x = [2, -1, 1, 0, 0, \dots]$$

$$w = [1, 1, 2, 0, 0, \dots].$$

5 The convolution of x with w (which is called the filter) is denoted by

$$(x * w)_k = \sum_{\tau=-\infty}^{\infty} x_{\tau} w_{k-\tau}. \tag{5.1}$$

6 The element $(x * w)_k$ at the k^{th} index is a composition of all the terms
 7 in the summation on the right hand side. The term $w_{k-\tau}$ for negative
 8 arguments is interpreted as a mirror flip of the vector w . For continuous
 9 functions, you will have seen the expression

$$(x * w)(t) = \int_{-\infty}^{\infty} x(\tau)w(t - \tau) d\tau.$$

10 for the convolution operation. For our vectors x, w with three entries the
 11 convolution operation looks as follows.

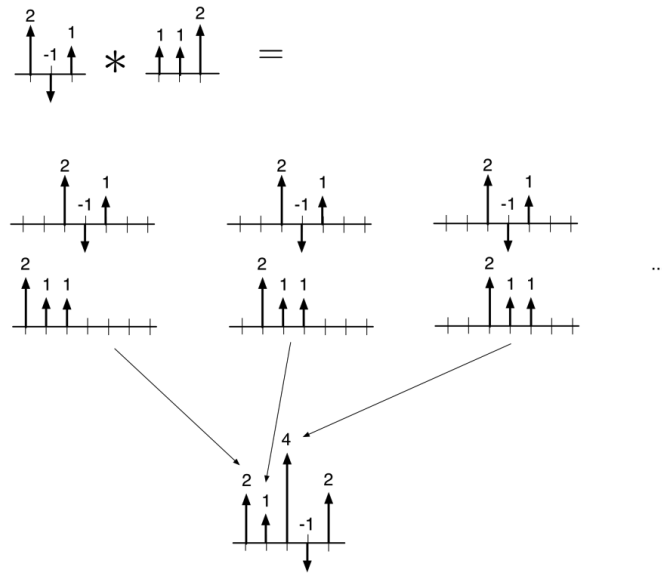


Figure 5.1: Flip and filter style computation of a convolution corresponding to the summation in (5.1).

12 **Remark 5.1 (Some identities regarding convolutions).** Notice that we

▲ In the signal processing literature, the words filter and kernels are used equivalently, so convolutional filters are also often called convolutional kernels.

🔗 Discuss the convolution of a square wave x with a saw-tooth wave w .

1 can change the variable of integration and set $s = t - \tau$ to get

$$\begin{aligned}(x * w)(t) &= \int_{-\infty}^{\infty} x(\tau)w(t - \tau) d\tau \\ &= - \int_{\infty}^{-\infty} x(t - s) w(s) ds \\ &= \int_{-\infty}^{\infty} w(s) x(t - s) ds \\ &= (w * x)(t).\end{aligned}$$

2 Convolutions are therefore commutative; you can show similarly that they
3 are also distributive $(f * g) * h = f * (g * h)$. Convolution is a linear
4 operator, you can show that

$$(f + g) * h = (f * h) + (g * h)$$

5 for any integrable functions f, g, h .

6 **Remark 5.2 (Padding for implementing convolutions).** In order to
7 implement the summation in convolution, we need to pad the input vector
8 x by zeros. How many zeros should we pad it by? You will notice that if
9 the kernel w has $2k + 1$ elements, the input vector x need not be padded
10 all the way to infinity, we only need to pad it with $2k$ extra elements on
11 each side.

12 5.1.1 Convolutions of 2D images

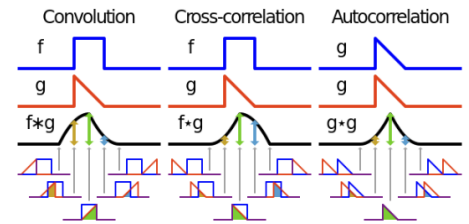
13 Convolutions work in the same way for two-dimensional or three-dimensional
14 input signals. The kernel w will be a matrix of size $k \times k$ in the former
15 case and of size $k \times k \times k$ in the latter.

$$(x * w)_{i,j} = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} x_{s,t} w_{i-s,j-t}. \quad (5.2)$$

▲ Most deep learning libraries implement a slightly different operation instead of convolution, even though they call it a convolution. They implement the cross-correlation operation

$$(x * w)_k = \sum_{\tau=-\infty}^{\infty} x_{\tau} w_{k+\tau}.$$

In simple words, the kernel w is not mirror flipped about the Y axis before computing the summation in (5.1). While such an operation is not strictly a convolution (you can see the difference if you consider an asymmetric kernel w ; cross-correlation and convolution are the same for symmetric kernels), the difference does not matter for deep learning because the kernel w is learned during training. You can mirror flip the kernel after training and interpret the network as indeed performing a convolution with the flipped kernel.



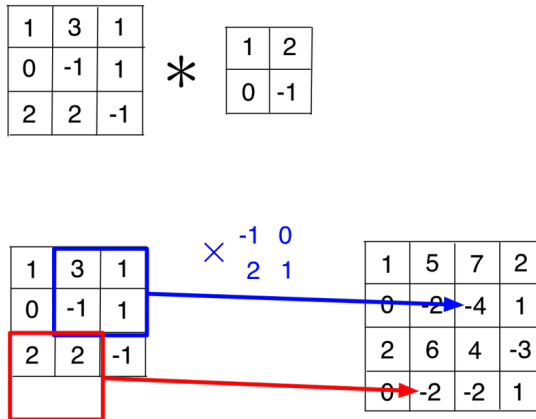


Figure 5.2: Flip and filter style computation of a convolution for a 2D input image corresponding to the summation in (5.2).

5.1.2 Some examples

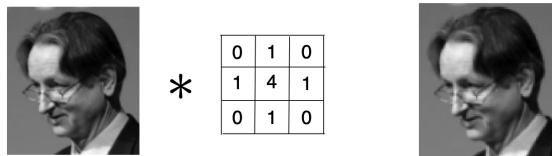
1. Since convolution is a linear operator we should be able to write it as a matrix-vector multiplication. We take the kernel, flip it and sweep it left to right to get the rows of the matrix.

$$(2, -1, 1) * (1, 1, 2) = \begin{bmatrix} 1 \\ 1 & 1 \\ 2 & 1 & 1 \\ & 2 & 1 \\ & & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}.$$

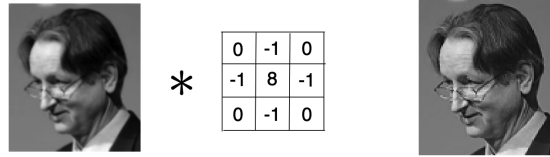
Such a matrix is called a Toeplitz matrix https://en.wikipedia.org/wiki/Toeplitz_matrix.

Two-dimensional convolutions can be written as a matrix-matrix multiplication using a similar construction; see <https://stackoverflow.com/questions/16798888/2-d-convolution-as-a-matrix-matrix-multiplication>.

2. Lots of non-trivial transformations of the image are possible using slight changes in the weights. E.g., blurring

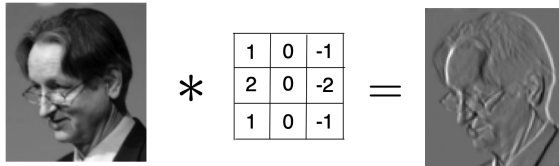


or sharpening,



1

2 We can also detect edges

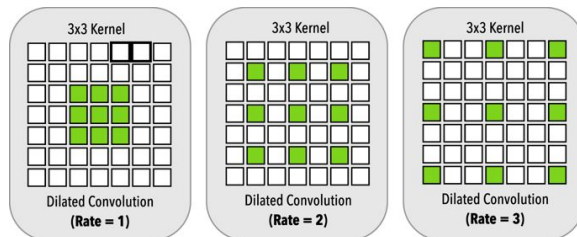


3

4 This filter is called the Sobel filter and is an integral part of image
5 pre-processing pipelines in computer vision.

- 6 3. Just like fully-connected layers, we can also stack up convolutions.
7 The effective receptive field, i.e., the pixels that are considered by
8 the kernel in the convolutional operation increases as we go up the
9 layers.
- 10 4. The operation $S^\top x$ has $S \in \mathbb{R}^{d \times p}$ weights and returns a vector
11 in \mathbb{R}^p . A convolution operator returns a vector $(x * w) \in \mathbb{R}^d$
12 using K parameters in the kernel w . It is important to note
13 that a lot of parameter sharing is happening while computing
14 the values of the output neurons. You can find some animations
15 at <https://colah.github.io/posts/2014-07-Conv-Nets-Modular> and
16 <https://colah.github.io/posts/2014-07-Understanding-Convolutions>.
- 17 5. Padding the input by zeros is common in signal processing because
18 the signals are usually a function of time. We can do a bit better for
19 images than zero padding ($\text{RGB} = (0, 0, 0)$) which is akin to creating
20 an artifact of a dark black border around the image. Reflection
21 padding is a technique (`torch.nn.ReflectionPad2d` in PyTorch) that
22 mirrors the pixels at the boundary and does not create such artifacts.

23 **Remark 5.3 (Dilated convolutions).** You don't need to use a kernel that
24 looks like a contiguous array. We can create holes in the kernel and expand
25 the receptive field. Dilated convolutions do precisely this.



26

🔍 What convolutional kernel does a dilated convolution correspond to?

1 These operators are very useful for image segmentation because they
 2 capture correlations across large parts of the input image while still
 3 enabling the parameter sharing of a convolutional layer.

4 **Remark 5.4 (Separable convolutions).** There are 9 weights in a 3×3
 5 kernel. Even convolutional layers can get really big, e.g., a standard
 6 CNN used for ImageNet has about 25M weights and is almost entirely
 7 convolutional. Thus we might want to reduce the number of weights even
 8 further. Separable convolutions are a trick to doing so. Consider a 3×3
 9 kernel and split it into two kernels of 3×1 and 1×3

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.$$

10 Using the original kernel requires 9 multiply operations to compute each
 11 pixel value. Using the split kernels requires only 6, it also has fewer
 12 weights. These are called separable convolutions. The Sobel filter which
 13 we saw before can be written as a separable convolution

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

14 because it measures the gradient of the image intensity independently in
 15 the two directions; an edge in an image is a region such that it is either
 16 an edge in the horizontal direction or an edge in the vertical direction.
 17 Separable convolutions are very useful when you can use high-dimensional
 18 data in deep learning, e.g., medical images out of MRI are 4-dimensional
 19 images (width, height, depth, channel).

20 5.2 How are convolutions implemented?

21 Convolutions are the most heavily used operator in a deep network. We
 22 therefore need to implement them as efficiently as we can. There are a
 23 few different ways of implementing convolutions.

- 24 1. Write a simple for loop. This works well if the kernel is small in
 25 size and this is indeed how PyTorch implements convolutions for
 26 kernels of size 3×3 (the operation is coded up in C, not Python of
 27 course).
- 28 2. We can expand out the kernel as a matrix and in this way a convolu-
 29 tional layer is simply a matrix-vector multiplication. This method is
 30 most commonly implemented and works well for sizes up to 5×5 .
- 31 3. We can use the Fast Fourier Transform (FFT) to compute the
 32 convolution as

$$x * w = \mathcal{F}^{-1} [\mathcal{F}[x] \mathcal{F}[w]].$$

❓ Can we write every 2D convolutional filter as a separable convolution? The answer is no: you will notice that a separable kernel is a rank-1 matrix. The singular value decomposition (SVD) of a separable kernel A is therefore

$$A = u v^\top$$

for two vectors u, v (we incorporated the singular value into u and v). Can we however *approximate* any convolutional kernel as a sum of separable convolutions? The answer to this is yes: observe using the SVD of the kernel $A \in \mathbb{R}^{p \times p}$ that it can be written as

$$A = \sum_{i=1}^p \lambda_i u_i v_i^\top.$$

where u_i, v_i are the singular vectors and λ_i are singular values. You don't have to pick all the factors, if you pick a few terms in this summation, you get a good spectral approximation of the matrix A . You will see in Section 5.3 how the convolutional layer in a deep network is structured and may allow the network to learn a complicated kernel A even if the operations are only separable $u_i v_i^\top$.

1 This is efficient for large kernels, say greater than 7×7 .

2 Typically, deep learning libraries will choose an algorithm for convolution
 3 in *run-time* after looking at your neural architecture; you do not have
 4 to worry about the specific algorithm. A library called cuDNN from
 5 Nvidia implements a bunch of convolution algorithms on GPUs efficiently.
 6 PyTorch will pick one of these algorithms by checking how long it takes
 7 for the first forward-pass on your deep network. But the fact remains that
 8 large kernels which allow a larger receptive field (long-range correlations
 9 in the input image) are more expensive to compute than smaller kernels.
 10 Architectures such as Inception that we will see soon are an attempt to get
 11 a large receptive field while still keeping computations in the convolutional
 12 layer small.

13 **Remark 5.5 (Stride in convolutional layers).** If you see the documenta-
 14 tion for the convolutional layer in PyTorch at
 15 (<https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html>) you will
 16 also see a parameter known as stride. Stride simply means that the output

$$(x * w)_k = \sum_{\tau=-\infty}^{\infty} x_{\tau} w_{k-\tau}$$

17 is not computed at all values of k ; if the stride is set to 2, the output is
 18 computed only at every alternate value of k . Note that the default stride
 19 as seen in the definition of convolution is 1. Since images change very
 20 little from pixel to pixel, this is a neat trick to reduce the redundancy
 21 of computing the convolution again and again over similar input. The
 22 important artifact of using a stride larger than 1 is that the output $(x * w)$
 23 is no longer the same length (even after padding) as the input, is half the
 24 length if the stride is 2.

25 5.3 Convolutions for multi-channel images in 26 a deep network

27 We will now study how the convolutional layer is implemented in a
 28 typical deep network. Let us denote the 2D convolution operation on a
 29 single-channel 2D image $A \in \mathbb{R}^{w \times h}$ by a kernel $w \in \mathbb{R}^{k \times k}$ by

$$A * w = B \in \mathbb{R}^{w \times h}.$$

30 Imagine that we have an RGB input image of size $w \times h$; the RGB indicates
 31 that there are three input channels, one for each color. The input to a
 32 convolutional layer in a deep network is therefore an array of size $3 \times w \times h$.
 33 Typical deep learning libraries, when they implement a convolutional
 34 layer with a kernel w of size $k \times k$, will output an image of size $c \times w \times h$
 35 where c are the number of channels in the image at the output of the layer.

36 Effectively, a convolutional layer maps

$$\mathbb{R}^{3 \times w \times h} \ni A \mapsto B \in \mathbb{R}^{c \times w \times h}.$$

▲ You can set `torch.cudnn.benchmark = False` to prevent Pytorch from searching for the best algorithm to compute convolutions for your architectures every time it launches. While such automated search speeds up training by a small fraction, it may not be desirable in case when you want to debug your code, or evaluate the run time of your algorithm.

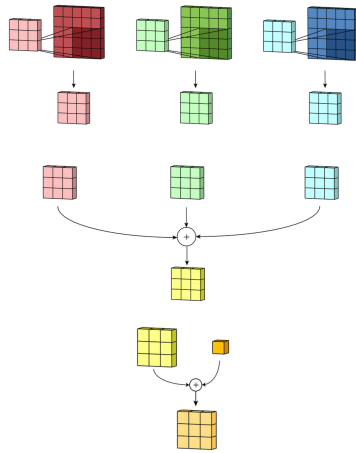


Figure 5.3: Convolutional layer in a typical deep network

1 The layer performs the operation

$$v_j + \sum_{i=1}^3 A_i * w^{ij} = B_j$$

2 where A_i for $i \in \{1, 2, 3\}$ denotes the i^{th} channel of the input image and
 3 B_j for $j \in \{1, \dots, c\}$ denotes the j^{th} channel of the output image, and
 4 the kernel $w^{ij} \in \mathbb{R}^{k \times k}$ is the convolutional kernel. The scalar $v_j \in \mathbb{R}$
 5 denotes the bias. Effectively, there are $3c$ different kernels in one layer
 6 and the convolutional layer sums up the result of convolutions on all the
 7 input channels and adds a bias to create *each* output channel.

8 5.4 Translational equivariance using convolu- 9 tions

10 We now discuss the most important reason for using convolutions in deep
 11 networks. Let us take our 1-dimensional signal x and translate it by Δ
 12 units to the right

$$x'(t + \Delta) := x(t).$$

❓ We said that convolutional filters are used to learn the correlations across nearby pixels. What would be the utility of 1×1 convolutions?

❓ If there are 10 input channels and 25 output channels, how many parameters does a convolutional layer with a 5×5 kernel have? What is the size of the output feature map if convolution is performed with a stride of 2? Does stride change the number of parameters in a convolutional layer?

1 You will see from the definition of convolution in (5.1) that the convolution
2 also gets translated

$$\begin{aligned}
 (x' * w)_k &= \sum_{\tau=-\infty}^{\infty} x'_\tau w_{k-\tau} \\
 &= \sum_{\tau=-\infty}^{\infty} x_{\tau-\Delta} w_{k-\tau} \\
 &= \sum_{s=-\infty}^{\infty} x_s w_{k-s-\Delta} \quad (s = \tau - \Delta) \\
 &= (x * w)_{k-\Delta}.
 \end{aligned}
 \tag{5.3}$$

3 In other words, if you translate the signal by Δ then the output of
4 convolution is also translated by the same amount

$$(x' * w)_{k+\Delta} = (x * w)_k.$$

5 This property is called equivariance. Equivariance also holds for 2D
6 convolutions. Equivariance to translations allows us to build an important
7 property in a deep network. If we have a convolutional kernel that has
8 weights such that the output is high for a certain object (star in adjoining
9 picture, vertical/slanted strips in your Gabor filter homework), the output
10 of a convolutional layer is such that the features also “move” if the input
11 moves in the receptive field.

12 We can easily build a binary classifier using such equivariant features.
13 If we want to build a star classifier, we simply check if some features in
14 the output are large after convolution, e.g., we check if the largest feature
15 in the 2D-feature map is greater than some pre-determined threshold

$$f(x, w) := \mathbf{1}_{\{\max_{i,j} \{(x*w)_{i,j}\} \geq \epsilon\}}. \tag{5.4}$$

16

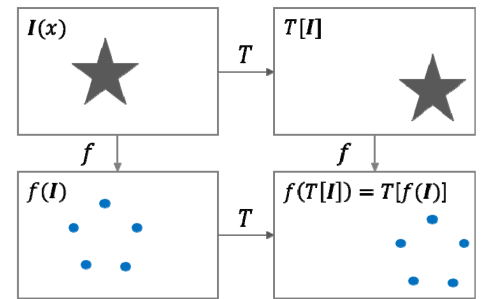
17 5.5 Pooling to build translational invariance

18 We would like to build a classifier such that if the object moves to some
19 other location in the input image, the output of the classifier remains
20 unchanged, i.e., the deep network detects a test image as a cat even if it is
21 in some other part of the image in the training data. Equivariance is only
22 one part of the story to doing so. Remember that the last layer in a deep
23 network looks like

$$f(x, w) = \text{sign}(v^\top h^L) = \text{sign}\left(\sum_{i=1}^p v_i h_i^L\right).$$

24 Even if the features h^L are equivariant when the input x is translated in
25 the 2D plane, the inner product $v^\top h^L$ cannot be equivariant. Essentially,
26 if a few weights v_i are trained to check for objects like cat/dog in one

▲ Translational equivariance is much more insightful for 2D images. Let us consider an example.



▲ The pre-activation features of a convolutional layer are sometimes called the *feature map*.

particular part of the image, even if the features h^L move accordingly, the output $v^\top h^L$ need not be constant because the weights v_i at those new locations of features may be different.

In other words, we want features of a deep network to be *invariant* to translations in the input.

Pooling is an operation that smears out the features locally in the neighborhood of each pixel.

We can use our idea of setting all the weights to 1 to get what is called the average pooling operation. It is a linear operation and equivalent to convolving the input features using a kernel

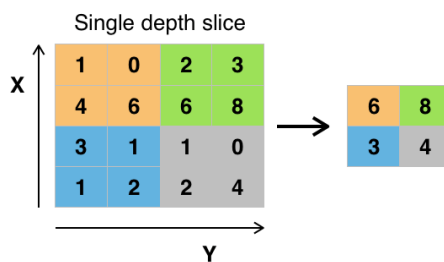
$$w_{\text{avg-pool}} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}. \quad (5.5)$$

The average-pooling kernel is fixed during training and does not have any weights, otherwise it would be just another convolutional kernel.

Average pooling does not solve our problem of making the features invariant; the smeared out version simply moves less than Δ when the input translates by Δ . If we add many average pooling layers at various stages in a deep network, we make the features move even less and this may be sufficient to allow for weights v to be discriminative.

Max-pooling is another operation that builds invariance. It takes an input $x \in \mathbb{R}^{w \times h}$ and computes

$$(\text{max-pool}(x))_{ij} = \max_{-k \leq s \leq k} \max_{-k \leq t \leq k} x_{i-s, j-t}. \quad (5.6)$$



Example of Maxpool with a 2x2 filter and a stride of 2

Figure 5.4: Max-pooling with a 2×2 kernel and a stride of 2 reduces the size of the input image by half. A stride of 1 would preserve the image size but would give less invariance.

This is a clever way of building invariance, you simply take the maximum value of the input in a window of size $k \times k$, so even if the input translates by k pixels in either direction, the output of a max-pooling layer remains the same. If we add multiple max-pooling layers at

▲ Making the weights of the top layer v all equal to 1 will solve this problem, but this is of course a very poor classifier. It smears the entire input signal h^L together by just averaging the features and therefore does not have much discriminative power; it cannot easily build a multi-class classifier for instance.

▲ Average pooling blurs the image. We saw this in the example in Section 5.1.2. Such blurring at intermediate layers gives *some* translational invariance by smearing out the features.

1 intermediate depths in a deep network, we achieve translational *invariance*
 2 in a convolutional neural network.

3 **Remark 5.6 (Max-pooling destroys information).** As we see in Fig-
 4 ure 5.4, max-pooling destroys a lot of information in the input image.
 5 The result of max-pooling is a much smaller feature map. This results
 6 in a large loss of information in the input data and often leads to a loss
 7 of discriminative power, i.e., accuracy, during training. This trade-off
 8 between building a classifier that is invariant to changes in the input and
 9 discriminative enough to distinguish between many different categories is
 10 fundamental.

11 Max-pooling has a side-benefit, it reduces the number of operations in
 12 a deep network and the number of parameters by sequentially reducing the
 13 size of the feature map with layers. This is useful because a typical image
 14 you get from an autonomous car is easily about 10MP (10^7 pixels) and we
 15 need to boil it down into, say 10 categories that are relevant to driving,
 16 i.e., $h^L \in \mathbb{R}^{10}$. Max-pooling is a very useful for this, with the caveat that
 17 too much pooling will dramatically reduce the signal in the input image.

❓ Does max-pooling make sense for a fully-connected network? There is no equivariance property in such a network, so even if we do perform max-pooling, it is just like another activation function operating on the features.

❓ We have talked about invariance to translations in this lecture. Images taken from a fish-eye camera, or MRI images of the brain, are such that objects *rotate* in the field of view.



Can you think of a trick to build invariance to rotations?

Chapter 6

Data augmentation, Loss functions

Reading

1. Bishop Chapter 5.5.3, 4.3
2. Goodfellow Chapter 7.4

6.1 Data augmentation

In the previous chapter, we looked at convolutions as a way to reduce the parameters in a deep network, but more importantly as a way of building equivariance/invariance to translations. There are a lot of nuisances other than translation that do not have a group structure which precludes operations such as convolutions that we can perform to generate equivariance/invariance.

In this section, we will discuss techniques to build invariance to nuisances that are more complex than just translations, these techniques will seem brute-force but they also allow us to handle more complex nuisances. The main trick is to *augment* the data, i.e., create variants of each input datum in some simple way such that we *know* that its label is unchanged. If our original dataset is $D = \{(x^i, y^i)\}_{i=1, \dots, n}$ we create an augmented dataset

$$T(D) := \{(T(x^i), y^i)\}_{i=1, \dots, n} \cup D. \quad (6.1)$$

where T is some operation of our choice. We have therefore expanded the number of samples in the training dataset to $2n$ instead of the original n . Effectively, data augmentation is a technique to create a dataset that is sampled from some other data distribution P than the original one.

6.1.1 Some basic data augmentation techniques

The most popular data augmentation techniques are setting T to be changes in brightness, contrast, cropping the image to simulate occlusions, flipping the image horizontally or vertically, jittering the pixels of the input image to simulate noise in the CCD of the camera/weather, padding the image which changes the borders of the input image, warping the image using a projection that simulates the same picture taken from a different viewpoint, thresholding the RGB color channels, zooming into an image to simulate changes in the scale etc.

You can see these operations at <https://fastai1.fast.ai/vision.transform.html#List-of-transforms>.

▲ FastAI is a wrapper on top of PyTorch and is an excellent library to learn for doing your course projects.

6.1.2 How does augmentation help?

A number of such augmentations are applied to the input data while training a deep network. This increases the number of samples n we have for training but note that different samples share a lot of information, so the effective novel samples has not increased by much. Let us get an idea of when augmentation is useful and when it is not. Consider a regression and classification problem as shown below.



Figure 6.1: Cows live in many different parts of the world. A classifier that also uses background information to predict the category is likely to make mistakes when it is run in a different part of the world. Augmenting the input dataset on the left by replacing the background to include a mountain or a city is therefore a good idea if we want to run the classifier in a different part of the world. This will also force the classifier to *ignore* the background pixels when it classifies the cow, in other words the classifier is forced to become invariant to backgrounds by brute-force showing it different backgrounds.

In essence, data augmentation forces the model to tackle a larger dataset than our original dataset. The model is forced to learn what

1 nuisances the designer would like it to be invariant to. Compare this to the
 2 previous chapter: by replacing fully-connected layers with convolutions
 3 and pooling we made the model invariant to translations. In principle,
 4 we could have trained a fully-connected deep network on a very large
 5 augmented dataset with translated objects. In principle, this would make
 6 the fully-connected network invariant to translations as well.

7 6.1.3 What kind of augmentation to use when?

8 In the example with regression, we saw that the regressor on the augmented
 9 data was essentially linear and had much less discriminative power than a
 10 polynomial regressor. This was of course by design, we chose to augment
 11 the data. If the test data for the problem came from the polynomial instead
 12 of our augmented distribution, the new classifier will perform poorly.



Figure 6.2: The second panel shows the original scene with a mirror flip (i.e., across the horizontal axis) while the third panel shows the original scene after a water reflection (i.e., flip across the vertical axis). The latter is an image that is very unlikely to occur in the real world, so it is not a good idea to use it for training the model.

By being invariant to a larger set of nuisances than necessary, we are wasting the parameters of the model and risk getting a large error if the test data was not from the augmented distribution. By being invariant to a smaller set of nuisances than necessary, we are risking the situation that the test data will have some new nuisances which the classifier will perform poorly on. It is important to bear in mind that we do not always know what nuisances the model should be invariant to, the set of transformations in data augmentations necessarily depends—often critically—upon the application.

13 Data augmentation requires a lot of domain expertise and often plays
 14 a huge role in the performance of a deep network. You should think about
 15 what kind of augmentations you will apply to data for speech processing,
 16 or for data from written text.

17 6.1.4 Data augmentation for text

18 6.2 Loss functions

19 We next discuss the various loss functions that are typically used for
 20 training neural networks. As usual, we are given a dataset

$$D = \{(x^i, y^i)\}_{i=1, \dots, n}.$$

❗ If you are building a classifier for detecting cars, motorbikes, people etc. for autonomous driving application, do you want to be the invariant to rotations?

6.2.1 Regression

MSE loss. If the labels are real-valued $y^i \in \mathbb{R}$, e.g., we are predicting the price of housing in Boston given features of the houses (like you did in HW 0), we are solving a regression problem and the loss function to use for a deep network is also simply the regression loss.

$$\ell_{\text{mse}}(w) := \frac{1}{2} (f(x; w) - y)^2 \quad (6.2)$$

If you think about it carefully, it seems silly to add different dimensions of the input x using the weights w . Consider the case of $x = [\text{miles/gallon, number of other people with the same car, price of the car}]$. The three elements of x are in totally different units and totally different scales. A popular trick to make things a bit more uniform for regression is take a logarithmic transformation of the input, i.e., fit a model to $\log x$ using the loss

$$\frac{1}{2} (f(\log x; w) - y)^2;$$

we can compute the logarithm element-wise for vector valued inputs.

Huber loss. The square-residual loss in (6.2) works in most cases but it does not work well if there are outliers in the data. Outliers are data in the training set that are noisy or did not come from the true model. In such cases, we can use the Huber loss. If the residual is $r = f(x; w) - y$, the Huber loss is

$$\ell_{\text{huber}}(w; \delta) = \begin{cases} \frac{1}{2} |r|^2 & \text{if } |r| \leq \delta \\ \delta (|r| - \frac{1}{2}\delta) & \text{else.} \end{cases} \quad (6.3)$$

Observe that this does not penalize the model egregiously if the predictions are bad ($|r| \geq \delta$) for a particular datum. Doing so prevents the outliers from biasing the loss towards themselves and ruining the residuals for the other data.

MAE loss. The absolute-error loss (or ℓ_1)

$$\ell_{\text{mae}}(w) = |f(x; w) - y| \quad (6.4)$$

has a similar motivation: it does not penalize the residual on the outliers.

Using a subset-selection technique or the ℓ_{mae} loss leads to sparse weights w^* . This makes the model more interpretable than a model fitted using ℓ_{mse} loss. This is easy to understand for linear models: input dimensions corresponding to weights w_i^* that are zero do not take part in making predictions. So one may answer questions of the form “is variable x_i a relevant predictor of the target y ”.

Variable importance. For linear models, another way to answer the same question is to fit two models, one with w_i fixed to zero and all other

▲ We can perform regression in a clever way: first set all weights $w_i = 0$ and iteratively allow a subset of the weights (say the ones that improve the residuals the most) to become non-zero; non-zero weights are fitted using ℓ_{mse} . This is known as forward selection. Backward selection starts with weights w^* which minimize ℓ_{mse} and iteratively prune the weights. Both forward and backward selection are techniques to fit a model w^* with sparse weights.

1 weights fitted using the MSE loss (6.2) and another model without fixing
 2 w_i ; the difference between the average square residuals in the two cases
 3 is a measure of how important the feature x_i is for the prediction. These
 4 techniques are called variable importance methods. We can also undertake
 5 the same program for nonlinear models on non-image based data.

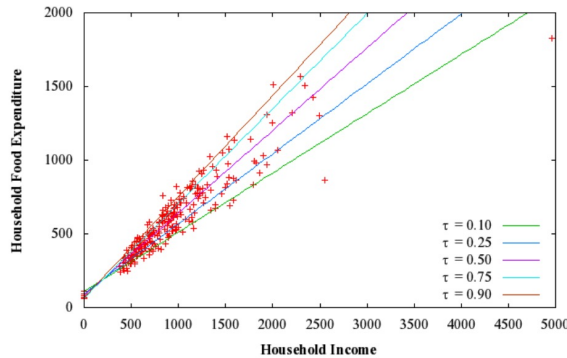
6 **Quantile loss.** The quantile loss is another simple trick to make the model
 7 more robust to outliers and get more information from the model than
 8 simply the prediction $f(x; w)$. Observe that if we have targets Y that are
 9 random variables with cumulative distribution function $F(y) = \mathbb{P}(Y \leq y)$,
 10 the τ^{th} quantile of Y is given by

$$Q_Y(\tau) = F^{-1}(\tau) = \inf \{y : F(y) \geq \tau\}$$

11 for $\tau \in (0, 1)$. We now learn a predictor for $Q_Y(\tau) = f(x; w)$. It turns
 12 out (you can try to prove this) that this corresponds to the loss function

$$\begin{aligned} \ell_{\text{quantile}}(w; \tau) &= \begin{cases} r(\tau - 1) & \text{if } r < 0 \\ r\tau & \text{else.} \end{cases} \\ &= r(\tau - \mathbf{1}_{\{r < 0\}}). \end{aligned} \quad (6.5)$$

13 where $r = y - f(x; w)$ is the residual. A standard technique is to
 14 fit multiple models using the quantile loss for different quantiles, say
 15 $\tau = 0.25, 0.5, 0.75$ and give multiple predictions of the target $f(x; w^\tau)$.
 16 A typical example of quantile linear regression looks as follows.



▲ The quantile loss is also called the pinball loss. Unlike the regression loss, it is highly asymmetric around the origin. If $r > 0$, we are penalizing the model by $\tau|r|$, and if $r < 0$, i.e., if we predict something that is larger than the true y , then we are penalizing the model by $(1 - \tau)|r|$.

17

18 6.2.2 Classification: Cross-Entropy loss

19 We next discuss the case when the targets are categorical and we wish to
 20 train a discriminative model that classifies the input into one of these m
 21 categories

$$y \in \{1, \dots, m\}.$$

1 **One hot encoding.**

2 An alternative representation of the targets in classification is so-called
3 the *one-hot* encoding where y is transformed to

$$\text{one-hot}(y) = e_y \in \mathbb{R}^m;$$

4 the vector e_y has a 1 at the y^{th} element and zeros everywhere else. The
5 notation e_y denotes the y^{th} row of the identity matrix $I_{m \times m}$.

6 **Predicting class probabilities.**

7 Instead of using the regression loss by treating y as a real-valued quantity,
8 it is more natural to predict the log-probability $\log p(k | x)$ for every
9 category k using weights w and predict the category using

$$f(x; w) = \underset{k}{\operatorname{argmax}} \log p_w(k | x). \quad (6.6)$$

10 Just like we denoted the raw predictions of the model by \hat{y} in linear/logistic
11 regression, we will denote

$$\mathbb{R}^m \ni \hat{y} = v^\top \sigma(S_L^\top \dots \sigma(S_2^\top \sigma(S_1^\top x)) \dots) \quad (6.7)$$

12 where $v \in \mathbb{R}^{p \times m}$. As we saw in Chapter 4, \hat{y} are also called logits.
13 Observe that the logits \hat{y} are simply vectors in \mathbb{R}^m . How can we transform
14 these logits to get $\log p_w(k | x)$ for all $k \in \{1, \dots, m\}$ as the output of
15 the model?

16 **Logistic loss.**

17 Linear logistic regression has a scalar output $\hat{y} \in \mathbb{R}$ which is interpreted
18 as the log-odds of the class probabilities

$$\log \frac{p(1 | x)}{p(0 | x)} = \hat{y} = w^\top x. \quad (6.8)$$

19 This expression can be rewritten as $p(1 | x) = \text{sigmoid}(\hat{y})$. The likelihood
20 of data x under this model for $y^i \in \{0, 1\}$ is

$$p_w(\{(x^1, y^1), \dots, (x^n, y^n)\}) = \prod_{i=1}^n p_w(1 | x^i)^{y^i} p_w(0 | x^i)^{1-y^i}.$$

21 Maximizing this probability (MLE) is the same as minimizing the
22 log-probability

$$\begin{aligned} \ell_{\text{logistic}}(w) &:= -\log p_w(\{(x^1, y^1), \dots, (x^n, y^n)\}) \\ &= -\sum_{i=1}^n y^i \log p_w(1 | x^i) + (1 - y^i) \log p_w(0 | x^i) \end{aligned} \quad (6.9)$$

23 In other words, the logistic loss is simply maximum-likelihood estimation
24 for the model (6.8).

🔗 We saw a different expression for the logistic loss in Chapter 3

$$\ell_{\text{logistic}}(w) = \log(1 + e^{-y \hat{y}}).$$

What is the difference?

1 Binary Cross-Entropy loss.

2 Let us turn back to neural networks and multi-class classification. Imagine
 3 if each logit of a neural network in (6.7) acts independently, i.e., it predicts
 4 whether there is class k in this input or not without paying heed to what
 5 the other logits predict. This is not very prudent, for instance, if we know
 6 beforehand that there is only one object in the input image, then such a
 7 classifier is likely to have lots of false positives. Nevertheless, observe
 8 that this is exactly like running m independent binary logistic classifiers
 9 with the same feature $h^L \in \mathbb{R}^p$. We can write the loss for such a classifier
 10 succinctly as

$$\ell_{\text{bce}}(w) = - \sum_{k=1}^m \text{one-hot}(y)_k \log p_w(k | x). \quad (6.10)$$

11 If the ground-truth labels y^i are such that there is only one class in each
 12 input image, all entries of $\text{one-hot}(y^i)$ at other categories will be zero, so
 13 this loss penalizes only the output of one of the m independent logistic
 14 classifiers.

15 6.2.3 Softmax Layer

16 Observe that our classifier which employs m binary logistic classifiers
 17 for predicting all the categories independently does not predict a valid
 18 probability distribution because

$$\sum_{k=1}^m p_w(k | x)$$

19 is not always equal to 1. We can however posit that the model predicts
 20 logits \hat{y} that are proportional to the log-probabilities

$$\begin{aligned} \log p_w(k | x) &\propto \hat{y}_k \\ \Rightarrow p_w(k | x) &= \frac{e^{\hat{y}_k/T}}{\sum_{k'=1}^m e^{\hat{y}_{k'}/T}}. \end{aligned} \quad (6.11)$$

21 The result $p_w(k | x)$ is a valid distribution on k because it sums up to 1.
 22 This operation, namely taking the logits \hat{y} and constructing a probabilities
 23 out of them is called as a softmax operator. The constant T in (6.11) is
 24 called the temperature. A large value of T results in a smoother probability
 25 distribution $p_w(k | x)$ because the individual values of the logits matter
 26 less. A small value of T results in a very large weight due to the exponent
 27 on the largest logit and the distribution $p_w(k | x)$ is therefore highly
 28 spiked. The temperature is set to 1 by default in PyTorch.

29 The cross-entropy loss is now simply the maximum-likelihood loss

▲ You will often see people calling

$$\log \sum_{k'=1}^m e^{\hat{y}_{k'}/T}$$

as the “softmax” of vector \hat{y} . This is actually a more appropriate usage of the word because

$$\log \sum_{k=1}^m e^{\hat{y}_k/T} \approx \max_k \hat{y}_k$$

if one of the entries of \hat{y} is much larger than the others, or if $T \rightarrow 0$. We will however use the word “softmax” to refer to the operation of transforming \hat{y} into $p_w(k | x)$ because we do not have any need for this softened version of the max operator.

1 after the softmax operation

$$\begin{aligned} \ell_{\text{ce}}(w) &= - \sum_{k=1}^m \text{one-hot}(y)_k \log p_w(k | x) \\ &= - \frac{\hat{y}_y}{T} + \log \left(\sum_{k'=1}^m e^{\hat{y}_{k'}/T} \right). \end{aligned} \quad (6.12)$$

2 Observe that the logit corresponding to the true class \hat{y}_y is being pushed
3 higher; at the same time, if the logits of the incorrect classes are large they
4 are being pulled *down* in the summation. This is an important point to
5 keep in mind: the cross-entropy loss after softmax affects *all* logits, not
6 just the logit of the correct class.

7 6.2.4 Label smoothing

8 The correct logit in (6.12) is encouraged to go to $+\infty$ while the incorrect
9 logits are encouraged to go to $-\infty$. This can lead to dramatic over-fitting
10 when the number of classes m is very large. Label smoothing is a trick
11 that alleviates the problem: instead of using a one-hot encoding of the
12 true label y , it uses the encoding

$$\text{label-smoothing}(y)_k = \begin{cases} 1 - \epsilon & \text{if } k = y, \\ \frac{\epsilon}{m-1} & \text{else.} \end{cases} \quad (6.13)$$

13 The cross-entropy loss with this new encoding is now

$$\begin{aligned} \ell_{\text{label-smoothing-ce}}(w) &= - \sum_{k=1}^m \text{label-smoothing}(y)_k \log p_w(k | x) \\ &= -(1 - \epsilon) \log p_w(y | x) - \frac{\epsilon}{m-1} \sum_{k \neq y} \log p_w(k | x) \end{aligned} \quad (6.14)$$

14 If you take the derivative of this loss with respect to \hat{y} you will see that
15 the value of \hat{y} that minimizes the loss is

$$\hat{y}_k^* = \begin{cases} \log((m-1)(1-\epsilon)/\epsilon) + \alpha & \text{if } k = y \\ \alpha & \text{else.} \end{cases} \quad (6.15)$$

16 where α is an arbitrary real number. Notice that logits for both the correct
17 and the incorrect classes are finite in this case, they no longer blow up to
18 infinity.

19 6.2.5 Multiple ground-truth classes

20 If there are multiple classes that are all present in the input image, i.e., if
21 the ground truth data has multiple labels, we can easily use the vector

$$\text{multi-hot}(y) = \sum_k e_k$$

1 for all the present classes k and set

$$\ell_{\text{bce}}(w) = - \sum_{k=1}^m \text{multi-hot}(y)_k \log p_w(k | x) \quad (6.16)$$

2 in the BCE loss. We can also use this trick in the cross-entropy loss
3 after the softmax operator but it will not work well because the softmax
4 operator is designed to amplify only the largest logit in \hat{y} ; if we tried the
5 network would still be incentivized to predict only one class instead of all
6 classes.

Chapter 7

Bias-Variance Trade-off, Dropout, Batch-Normalization

Reading

1. Bishop 1.3, 3.2, 14.2-14.3
2. Goodfellow 5.1-5.4, 7.1-7.3
3. Dropout [Srivastava et al. \(2014\)](#)
4. Batch-Normalization [Ioffe and Szegedy \(2015\)](#)

In this chapter, we will take our first look at how machine learning classifiers generalize to new data. We will first discuss the so-called Bias-Variance Tradeoff which indicates that the variance of the predictions of a model can be reduced by increasing its bias. Regularization is a technique to give us control over this tradeoff. We will then see a few popular regularization techniques, in particular two that are important in deep learning called Dropout and Batch-Normalization.

7.1 Bias-Variance Decomposition

Ideally, we want a classifier that accurately captures the regularity in the data but also works well for unseen data. Turns out this is typically impossible to both simultaneously. We will introduce this using regression.

Given our dataset $D = \{(x^i, y^i)\}_{i=1, \dots, n}$ we fit a model $f(x; w) \in \mathcal{F}$ where \mathcal{F} is some class of models, say all neural networks with a given architecture; we will keep the dependence of f on w implicit in this section

1 because we don't need it. We use a loss $\ell(f(x), y) = |f(x) - y|^2$ to fit
 2 this model by minimizing

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^n (f(x^i) - y^i)^2 \quad (7.1)$$

3 This is of course the training loss, also called the empirical risk. A
 4 classifier that minimizes $\hat{R}(f)$ works well on the training data. If we want
 5 to measure how well a model works on new data from the distribution P
 6 we are interested in the the *population risk*

$$\begin{aligned} R(f) &= \int (f(x) - y)^2 P(x, y) \, dx \, dy \\ &= \mathbb{E}_x \left[\int (f(x) - y)^2 P(y | x) \, dy \right]. \end{aligned} \quad (7.2)$$

7 It turns out that because the loss is quadratic, we can write down the
 8 minimizer of the population risk, formally, as

$$f^*(x) = \mathbb{E}[y | x]. \quad (7.3)$$

9 In other words, the optimal regression is the conditional expectation of
 10 the targets y given a datum x . Since we do not know the data distribution
 11 P , we cannot compute the model f^* . We now compare some regression
 12 f that we may have obtained by minimizing (7.1) with the optimal f^* .

13 Observe that

$$\begin{aligned} (f(x) - y)^2 &= (f(x) - f^*(x) + f^*(x) - y)^2 \\ &= (f(x) - f^*(x))^2 + 2(f(x) - f^*(x))(f^*(x) - y) + (f^*(x) - y)^2. \end{aligned}$$

14 Substitute this expression in (7.2) to get

$$R(f) = \mathbb{E}_x \left[(f(x) - f^*(x))^2 \right] + \mathbb{E}_{(x,y) \sim P} \left[(f^*(x) - y)^2 \right] \quad (7.4)$$

15 Observe that the cross-term

$$\mathbb{E}_x \left[\int 2(f(x) - f^*(x))(f^*(x) - y)P(y | x) \, dy \right] = 0$$

16 vanishes because $f^*(x) = \mathbb{E}[y | x] = \int yP(y | x) \, dy$. In the first term,
 17 there is no y because the distribution $P(y | x)$ when integrated with
 18 respect to y is 1. The decomposition in (7.4) is insightful. The first term
 19 tells us how far our model $f(x)$ is from the optimal $f^*(x)$, at any input x .
 20 The second term tells us how much the optimal model itself is from the
 21 data (x, y) . The second term is not under our control because it does not
 22 depend on $f(x)$ at all. This term is called the

$$\text{Bayes error} = \mathbb{E}_{(x,y) \sim P} \left[(f^*(x) - y)^2 \right]. \quad (7.5)$$

23 It is irreducible error of any classifier f that we can train. It is only zero

1 if the data (x, y) is coming from a deterministic source, i.e., there is no
 2 noise in the true targets y created by Nature and Nature's model. It is
 3 important to realize that Nature's model is *not* f^* .

4 We will now investigate the first term better. Our model f is created
 5 using a finite training dataset D . Let us emphasize it as

$$f(x; D)$$

6 and rewrite the first term in (7.4) as

$$\begin{aligned} (f(x; D) - f^*(x))^2 &= \left(f(x; D) - \mathbb{E}_D [f(x; D)] + \mathbb{E}_D [f(x; D)] - f^*(x) \right)^2 \\ &= \left(f(x; D) - \mathbb{E}_D [f(x; D)] \right)^2 \\ &\quad + \left(\mathbb{E}_D [f(x; D)] - f^*(x) \right)^2 \\ &\quad + 2 \left(f(x; D) - \mathbb{E}_D [f(x; D)] \right) \left(\mathbb{E}_D [f(x; D)] - f^*(x) \right). \end{aligned}$$

7 Recall that the dataset is a random variable as well: it is a bunch of samples
 8 from the Nature's distribution over (x, y) denoted by P . Effectively,
 9 $f(x; D)$, which is our fitted model is a random variable that depends on
 10 the randomness of D . We now take the expectation over the *dataset* D on
 11 both sides of this equation.

$$\mathbb{E}_D \left[(f(x; D) - f^*(x))^2 \right] = \underbrace{\left(\mathbb{E}_D [f(x; D)] - f^*(x) \right)^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_D \left[\left(f(x; D) - \mathbb{E}_D [f(x; D)] \right)^2 \right]}_{\text{variance}} \quad (7.6)$$

12 The cross-term again vanishes when we take the expectation over the
 13 dataset (convince yourself of this by writing out the cross-term). The
 14 first term is called the squared bias: it is the gap between the predictions
 15 of our model compared to the optimal model f^* created across many
 16 experiments, each with a different dataset D . The second term is the
 17 variance and it measures how sensitive our model $f(x; D)$ is to a particular
 18 training dataset D . If our model fitted on D does not work well on most
 19 others datasets, then the variance is large. We will parse these quantities
 20 further soon.

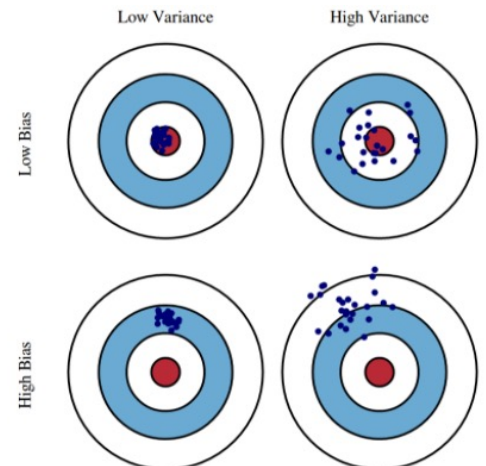
21 We have therefore shown that

$$R(f) = \mathbb{E}_x [\text{bias}^2 + \text{variance}] + \text{Bayes error} \quad (7.7)$$

22 Recall that we want to minimize the population risk $R(f)$. We cannot do
 23 much about the Bayes error. If the model $f(x; D)$ is large and is fitted
 24 very well on the dataset D , i.e., if its predictions match true y (notice that
 25 the optimal models predictions f^* are also close to y), the gap between
 26 the predictions of the fitted model and the optimal model is small on the
 27 dataset D . In other words, if our model is large we will have a small bias.
 28 The bias of a model decreases as we consider larger models $f(x; D)$. If
 29 our dataset is small, the model $f(x; D)$ is likely to have a large variance

▲ You can think of the Bayes error as being non-zero if the sensor used to measure y is noisy, there is no way we can get deterministic data in that case. If on the other hand the sensor is perfect, e.g., a large number of humans are annotating data very carefully like we often do in modern machine learning, the Bayes error is essentially zero.

▲ Here is a good mnemonic to remember. Imagine the center of the bull's eye as the optimal classifier f^* and our darts as the model $f(x; D)$. We have to collect n samples for every dart we throw.



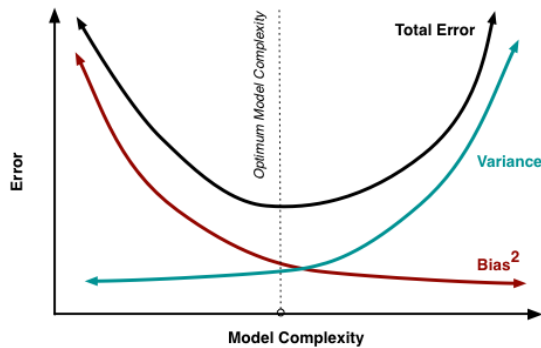


Figure 7.1: Population risk as a function of model capacity

1 because it has not seen a large amount of data. The effect increases
 2 for larger models because they may use a larger number of nuisances
 3 i.e., features that are not relevant to prediction of targets. We call this
 4 over-fitting.

5 If we plot a picture of how the bias and variance change as model
 6 capacity (you can think of capacity simply as the number of parameters
 7 in a model for now) increases, we see a famous U-shaped curve for the
 8 sum of squared bias and variance shown in Figure 7.1. Given a dataset
 9 D we should pick a model that lies at the bottom of this curve to get a
 10 good population risk; this model makes a good tradeoff between bias and
 11 variance.

12 The caveat is that we do not have access to a lot of different datasets to
 13 measure the bias or the variance. This is why the bias-variance trade-off,
 14 although fundamental in machine learning/statistics and a great thinking
 15 tool, is of limited direct practical value.

16 Bias-variance tradeoff for classification

17 We have only talked about the bias-variance trade-off for regression. The
 18 development for classification is not very different and same principles
 19 hold. We first define an optimal classifier

$$f^*(x) = \operatorname{argmin}_{f \in \mathcal{F}} \mathbb{E}_{(x,y) \sim P} [\ell(y, f(x))]$$

20 for a loss function ℓ . The bias, variance of a given classifier $f(x; D)$
 21 relative to this optimal classifier and the Bayes error are given by

$$\begin{aligned} \text{bias} &= \mathbb{E}_x [\ell(f^*(x), f(x; D))] \\ \text{variance} &= \mathbb{E}_D [\ell(f(x; D), f^{\text{avg}}(x))] \\ \text{Bayes error} &= \mathbb{E}_{(x,y) \sim P} [\ell(y, f^*(x))]. \end{aligned} \quad (7.8)$$

22 where $f^{\text{avg}}(x) = \operatorname{argmin}_f \mathbb{E}_D [\ell(y, f(x))]$; under the MSE loss this is the
 23 average of predictions of regressions on different datasets, for the MAE

⚠ You should not try to draw analogies between the bias-variance tradeoff for regression and that for classification given below. The former is classical but the latter has many different formulations that are designed more to follow the vague principles of what bias and variance mean in the context of classification.

1 loss this is the median of the predictions of models trained on different
 2 datasets, for the zero-one loss it is the most frequent prediction of models
 3 trained on different datasets. We again have a trade-off that is obtained by
 4 decomposing the population risk

$$\mathbb{E}_{(x,y) \sim P} \left[\mathbb{E}_D [\ell(y, f(x; D))] \right] = \text{bias} + c_2 \text{variance} + c_1 \text{Bayes error}.$$

5 where c_1, c_2 are constants. You can read more about this in [Pedro \(2000\)](#).

6 Double-descent

7 The surprising thing is that for deep networks, we do not see this classical
 8 bias-variance trade-off. The population risk looks like

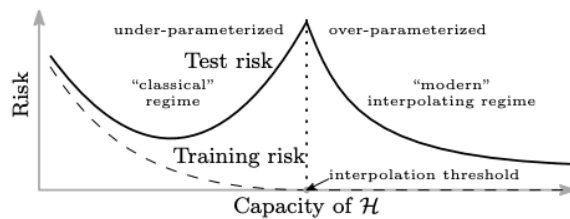


Figure 7.2: Double-descent curve: the validation error of deep networks decreases even if more and more complex models are fitted on the same data; there is no apparent over-fitting and growth in the variance of the classifier.

9 in what is now called the “double-descent” curve. The population risk
 10 of deep networks keeps decreasing even if we fit very large models on
 11 relatively small datasets, e.g., CIFAR-10 has 50,000 images, the model
 12 you will fit in HW 2 has about 1.6M weights and is considered a very small
 13 model by today’s standards. We will see some heuristic derivation into why
 14 the population risk may look like this for deep networks but understanding
 15 this phenomenon which goes flat against established knowledge in machine
 16 learning is one of the big open problems in the study of deep networks
 17 today.

18 7.1.1 Cross-Validation

19 We have seen that the bias-variance trade-off requires us to consider
 20 multiple datasets. In practice, we only have *one* dataset that we collected
 21 by running an experiment. If this data is large, we can split it into two
 22 three parts

$$\text{data} = \text{training set} \cup \text{validation set} \cup \text{test set}.$$

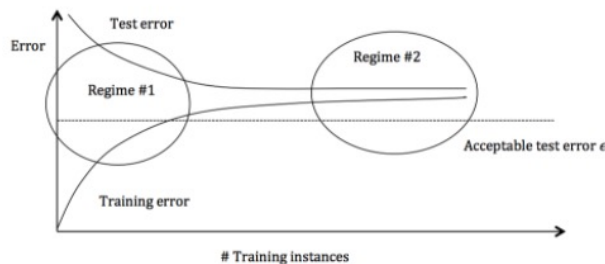
23 The validation set is used to compare multiple models that we fit on the
 24 training set and pick the best performing one. This model is then run on
 25 the test set to demonstrate how well we have learned the data. The test
 26 set is necessary because across your design efforts to fit different models,

1 you will evaluate on the validation set multiple times and this may lead to
2 over-fitting on the validation set.

3 If the available data is not a lot, we want to use as much of the data
4 as possible for training. If however only use a small fixed validation set
5 for comparing models, we risk making mistakes in our choices. Cross-
6 validation is a solution to this problem: it trains k different models, each
7 time a fraction $(k - 1)/k$ of the data is used as the training set and the
8 remainder is used as the validation set. The validation performance of k
9 models obtained by this process is averaged and used as a score to evaluate
10 a particular model design (architecture, hyper-parameters etc).

11 Some practical tips

12 It is useful to think of the bias-variance trade-off when you fit deep
13 networks in practice. If the training or test error is high, there are a number
14 of ways to improve performance using the bias-variance tradeoff as a
15 thinking tool.



16
17 In the first regime on the left, we have high validation error across cross-
18 validation folds and low training error. This indicates that we have a
19 high variance in the bias-variance trade-off. Typical techniques to counter
20 this is to use a smaller model, get more data, or bagging a set of models
21 together (will cover this in Section 7.3). In the second regime on the right,
22 if the test error *and* the training error are close to each other but both are
23 large, the model is likely to have high bias. In these cases, we should
24 fit a more complex model (say increase the number of weights, or pick
25 a different architecture), add more features to the training data (in the
26 non-deep-learning setting) to give our model more discriminative features
27 to use, or use boosting (we will cover this in Section 7.3).

28 Cautionary Tale

29 You will however notice that a lot of research papers in deep learning
30 simply use validation data as test data. Their reasons for doing so are
31 as follows. All researchers have the same large dataset from which they
32 would create a potential test set, the researchers therefore also know the
33 ground-truth labels of test images and it is difficult to trust them not to
34 peek at the ground-truth labels to choose between models. If the test
35 data is hidden from everyone, we need a centralized server for evaluating
36 everyone's results. This is difficult because research is fundamentally

▲ 4-fold cross-validation.



1 about discovering new knowledge. Kaggle competitions or the ImageNet
 2 Challenge <http://image-net.org/challenges/LSVRC> are few instances where
 3 such a centralized server is available.

4 It is therefore debatable whether the current practice of using validation
 5 set as the test set should be considered valid. On the positive side, it
 6 makes results across different publications comparable to each other; if
 7 everyone reports the error of their model on the same validation set, it
 8 is easy to compare Algorithm A versus Algorithm B. On the negative
 9 side, this incentivizes extensive hyper-parameter tuning and risks results
 10 that are over-fitted on the validation data, e.g., new fields such as neural
 11 architecture search are particularly problematic in this context. This is also
 12 the main reason for the current “style of research” where folks judges the
 13 merit of machine learning research simply by checking whether Algorithm
 14 A gets better validation error than Algorithm B on standard datasets. This
 15 is not the correct way to do scientific research. The more appropriate
 16 way to instantiate the scientific method is to first formulate a hypothesis,
 17 e.g., is gene X correlated with cancer Y, then collect data that allows
 18 us to evaluate such an hypothesis and undertake appropriate statistical
 19 precautions report whether the hypothesis stands/does not stand.

20 That said, there are researchers who have evaluated others’ claims
 21 (obtained using validation data, namely A better than B) on independent test
 22 data and reached similar conclusions, see for example <https://arxiv.org/abs/1902.10811>,
 23 so the evaluation methodology is broken but the progress is real.

24 7.2 Weight Decay

25 The set of models with smaller complexity are a subset of the set of models
 26 with larger complexity, e.g., if you are fitting a polynomial regression, you
 27 can consider the subset of models with coefficients of the higher-order
 28 terms equal to zero and have thus created the set of linear regressions.
 29 Effectively, the space of *models* looks as follows.

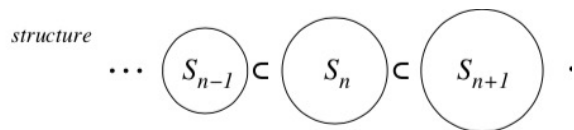


Figure 7.3: A cartoon of the space of models. The n in the picture refers to number of parameters in the model, not the number of data.

30 Let’s say we are fitting a class of models with large complexity and are
 31 unsure whether the variance in the bias-variance trade-off will be large.
 32 We can either collect more data, or we can modify the loss function to
 33 encourage the training process to pick models of lower complexity.

Restricting the space of models that the training process searches

over to fit the data is called *regularization*. We will denote regularizers by

$$\text{regularizer} = \Omega(w)$$

and modify our loss function for fitting data to be

$$\ell'(w; x, y) := \ell(w; x, y) + \Omega(w).$$

1 Weight decay is one of the simplest regularization techniques and uses

$$\Omega(w) = \frac{\alpha}{2} \|w\|_2^2. \quad (7.9)$$

2 This is more widely known as ℓ_2 regularization because we use the ℓ_2 norm
 3 of the weights as the regularizer. It is also called Tikonov regularization,
 4 a name that comes from the literature on partial differential equations.
 5 The name weight decay comes from the neural networks literature of the
 6 1980s. The gradient of the modified loss is

$$\nabla \ell'(w; x, y) = \nabla \ell(w; x, y) + \alpha w,$$

7 which gives

$$w^{(t+1)} = (1 - \eta \alpha) w^{(t)} - \eta \nabla \ell(w^{(t)}; x, y);$$

8 where η is the learning rate. In other words the weights w are encouraged
 9 to become smaller in magnitude when SGD takes a step using the negative
 10 gradient.

11 If we have a linear regression problem with $f(x; w) = w^\top x$ and
 12 X, Y are the matrices for the data and targets respectively, the regularized
 13 objective is

$$\frac{1}{2} \|Y - Xw\|_2^2 + \frac{\alpha}{2} \|w\|_2^2$$

14 and you can compute the minimizer by taking derivatives and setting them
 15 to zero to be

$$w^* = (X^\top X + \alpha I)^{-1} X^\top Y.$$

16 In other words, weight decay for linear regression adds elements to
 17 the diagonal of the data covariance matrix $X^\top X$. This results in a
 18 smaller inverse and thereby a smaller magnitude of w^* . Notice that if the
 19 covariance matrix is rank deficient, the regularized matrix is no longer
 20 rank deficient. If the covariance matrix has a large condition number (ratio
 21 of the largest and smaller eigenvalue), which makes taking the inverse
 22 numerically difficult, the regularized matrix has a better condition number.

23 7.2.1 Do not do weight decay on biases

24 If the input data and targets in linear regression are centered we do not
 25 need a bias parameter in our model. Notice however that if the dataset is
 26 not centered, the bias parameter is essential. Should we perform weight

1 decay on the bias parameter in this case? The weight decay penalty
 2 prevents the bias parameter to adapt to the non-zero mean of the data.
 3 This is also important to keep in mind while training neural networks. We
 4 should not impose weight decay regularization on the bias parameters of
 5 the convolutional and fully-connected layers.

6 7.2.2 Maximum a posteriori (MAP) Estimation

7 MAP estimation gives a Bayesian perspective to regularization in machine
 8 learning. In maximum likelihood (ML) estimation, we were interested in
 9 solving for weights that maximize the likelihood of the observed data:

$$w_{\text{MLE}}^* = \underset{w}{\operatorname{argmin}} -\frac{1}{n} \sum_{i=1}^n \log p_w(y^i | x^i; w).$$

10 MAP estimation enforces some prior knowledge we may have about the
 11 weights w . In Bayesian statistics, such prior knowledge is represented as
 12 a probability distribution, known as the *prior*, on the parameters w *before*
 13 *we see any data in the training process*, i.e., *a priori probability*

$$\text{prior} = p(w)$$

14 MAP estimation is regularized ML estimation. Given a prior distribution,
 15 we can use Bayes law to find the *posterior distribution* on the weights
 16 after observing the data

$$p(w | D) = \frac{p(D | w) p(w)}{p(D)} \quad (7.10)$$

17 Remember that the left hand side is a legitimate probability distribution
 18 with the denominator given by

$$Z := p(D) = \int p(D | w) p(w) \, dw.$$

19 The denominator Z called the “evidence” or the partition function lies at
 20 the heart of all statistics, we will see why in Module 4.

21 MAP estimation finds the weights that maximize this *a posteriori*
 22 probability

$$\begin{aligned} w_{\text{MAP}}^* &= \underset{w}{\operatorname{argmax}} \log p(D; w) + \log p(w) \\ &= \sum_{i=1}^n \log p_w(y^i | x^i; w) + \Omega(w) + \log Z(D). \end{aligned} \quad (7.11)$$

23 In the second step, we have denoted the log-prior by Ω

$$\log \text{prior}(w) := \Omega(w).$$

24 Note that $Z(D)$ is not a function of the weights w and can therefore can

▲ Weight decay is closely related to other norm-based penalties, e.g., ℓ_1 regularization sets

$$\Omega_{\ell_1}(w) = \alpha \|w\|_1.$$

As we discussed briefly in Chapter 6, such a regularizer encourages the weights to become sparse. Sparsity penalties are very common in the signal processing literature (e.g., compressed sensing, phase retrieval problems) but they are less common in the deep learning literature.

1 be ignored in the optimization.

2 Frequentist vs. Bayesian point of view

3 This section was our first view into Bayesian probabilities, as opposed
 4 to frequentist methods where we estimate probabilities by counting how
 5 many times a certain event occurs across our experiments. Frequentist
 6 probabilities are not designed to handle all situations. For instance we may
 7 be interested in estimating the probability of a very unlikely event, say
 8 that of the sun going supernova. This event has of course not happened
 9 yet and a frequentist notion of probability where we repeat the experiment
 10 many times and estimate the probability as the fraction of times the event
 11 occurs is not appropriate. The Bayesian point of view provides a natural
 12 way to answer these questions and the key idea is to encode our belief that
 13 the sun cannot go supernova as a prior probability.

14 An alternate way to think about this is that the weights w of a model
 15 are considered a fixed quantity that we are supposed to estimate in a
 16 frequentist setting. The likelihood $p(D; w)$ is used to compare different
 17 models w and if one wanted an estimate of how much error we are making
 18 in our estimate, we would compute the variance in the Bias-variance
 19 tradeoff namely, the variance of our estimate across different draws of the
 20 dataset D . In the Bayesian point of view, there is a single dataset D and
 21 the uncertainty of our estimate of w^* would be expressed as the variance
 22 of the posterior distribution $p(w | D)$ in Bayes law.

23 Weight decay regularization is MAP estimation with Gaussian prior

24 Weight decay can be seen as using a Gaussian prior

$$p_{\text{weight-decay}}(w) \propto e^{-\frac{\|w\|_2^2}{(2\alpha^{-1})}}.$$

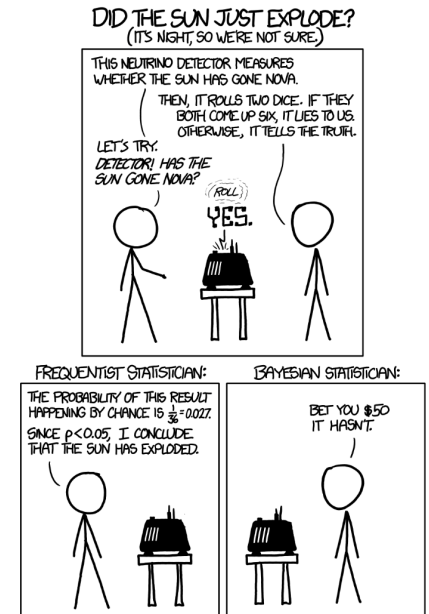
25 This is a multi-variate Gaussian distribution with mean zero and a diagonal
 26 covariance matrix with α^{-1} on the diagonal. The denominator is a function
 27 of α^{-1} and we do not need to worry about it while performing MAP
 28 estimation because it does not depend on w .

29 In other words, we have seen that weight decay in the training objective
 30 can be thought of as a MAP estimation using a Gaussian prior instead of
 31 ML estimation.

32 The Gaussian prior captures our a priori estimate of the true weights:
 33 the probability of the weights w being large is low (it is distributed as a
 34 Gaussian/Normal distribution). The likelihood term fits the weights to the
 35 data but instead of relying completely on the data which may result in a
 36 large variance (in cases when data is few), we also rely on the prior while
 37 fitting the model. This reasoning is captured in Bayes law.

38 Similarly, a sparsity penalty is MAP estimation with a Laplace prior
 39 For scalar random variables, the Laplace distribution is given by

$$p(w) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}.$$



1 If we have

$$\Omega(w) = \|w\|_1$$

2 we can see that regularized ML, i.e., MAP estimation corresponds to using
3 a Laplace prior on the weights w .

4 7.3 Dropout

5 We will next look at a very peculiar regularization technique that is unique
6 to deep networks. Consider a two-layer network given by

$$\hat{y} = v^\top \text{dropout}(\sigma(S^\top x)).$$

7 Dropout is an operation that is defined as

$$\text{dropout}_{1-p}(h) = h \odot r \quad (7.12)$$

8 where $r \in \{0, 1\}^p$ is a binary mask and the notation \odot denotes element
9 multiplication. Each element of this mask r_k is a Bernoulli random
10 variable with probability $1 - p$

$$r_k = \begin{cases} 0 & \text{with probability } p \\ 1 & \text{with probability } 1 - p. \end{cases}$$

11 In simple words, dropout takes the input activations h and zeros out a
12 random subset of these; on an average p fraction of the activations are set
13 to zero and the rest are kept to their original values. In pictures, it looks
14 as follows.

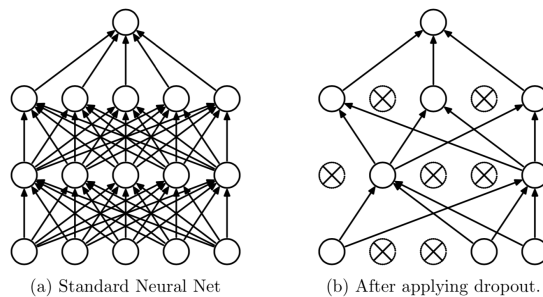


Figure 7.4: Dropout picks a random sparse subnetwork of a large deep network using the mask.

15 The default Dropout probability is $p = 0.5$ in PyTorch, i.e., about half
16 of the activations are set to zero for each input. Although you will see a
17 lot of online code and architectures with this default value, you should
18 experiment with the value of p , different values often given drastically
19 different training and validation errors.

⚠ It is important to remember that a new dropout mask r is chosen for every input in the mini-batch.

❓ The dropout mask is chosen at random for each image. Let us imagine that we have one dropout layer after every fully-connected layer. For the network shown in the figure with two hidden layers and 5 neurons at each layer, how many distinct sparse networks can we choose using dropout? Does the answer depend upon the probability p ?

7.3.1 Bagging classifiers

Bagging, which is short for *bootstrap aggregation*, can be explained using a simple experiment. Suppose we wanted to estimate the average height μ of people in the world. We can measure the height of N individuals and obtain *one* estimate of the mean μ . This is of course unsatisfying because we know that our answer is unlikely to be the mean of the entire population. Bootstrapping computes multiple estimates of the mean μ_k over many *subsets* of the data N and reports the answer as

$$\mu := \text{mean}(\mu_k) + \text{stddev}(\mu_k).$$

Each subset of the data is created by sampling the original data with N samples *with replacement*. This is among the most influential ideas in statistics (Efron, 1992) because it is a very simple and general procedure to obtain the uncertainty of the estimate.

Effectively, the standard deviation of our new bootstrapped estimate of the mean is simply the standard deviation in the Bias-Variance trade-off with the big difference that we created multiple datasets D by sub-sampling with replacement of the original dataset.

Bagging is a classical technique in machine learning (Breiman, 1996) that trains multiple predictive models $f(x; w^k)$ for $k \in \{1, \dots, M\}$, one each for bootstrapped versions of the training dataset $\{D^1, \dots, D^M\}$. We aggregate the outputs of all these models together to form a *committee*

$$f(x; w^1, \dots, w^M) = \frac{1}{M} \sum_{k=1}^M f(x; w^k).$$

You can see that this procedure reduces the variance of the model (the first term in (7.4)) in the bias-variance trade-off by a factor of M if the errors with respect to the optimal classifier f^* of all the models $\{w^k\}$ are zero-mean and uncorrelated. In other words, the average error of a model can be reduced by a factor of M by simply averaging M versions of the model.

Bagging is always a good idea to keep in your mind. The winners of most high-profile machine learning competitions, e.g., the Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize) or the ImageNet challenge, have been bagged classifiers created by fitting multiple architectures on the same dataset. Even today, random forests are among the most popular algorithms in the industry; these are ensembles of hundreds of models called decision trees on bootstrapped versions of data. A lot of times, if we are combining diverse architectures in the committee, we do not even need to bootstrap the data. Bagging does not work when the errors of the different models are correlated; this is however easy to fix by censoring out features in addition to bootstrapping like it is done while training a random forests.

7.3.2 Some insight into how dropout works

Consider the following, very heuristic but nevertheless beautiful, argument in the original paper on dropout (Srivastava et al., 2014).

We will remove the nonlinearities and consider only a single layer linear model with dropout directly applied to the input layer $f(x; w) = w^\top \text{dropout}(x)$. Linear regression minimizes the objective $\|y - Xw\|_2^2$ and similarly the dropout version of linear regression for our model would minimize

$$\min_w \mathbb{E}_R [\|y - (R \odot X)w\|_2^2] \quad (7.13)$$

where each row of the matrix R consist of the dropout mask for the i^{th} row x^i of the data matrix X . Think carefully about the expectation over R on the outside, since we choose a random dropout mask each time an input is presented to SGD, the correct way to write dropout is using this expectation over the masks. Each entry of R is a Bernoulli random variable with probability $1 - p$ of being 1. Note that

$$\mathbb{E}_R [R \odot X] = (1 - p)X$$

and the $(ij)^{\text{th}}$ element is

$$\left(\mathbb{E}_R [(R \odot X)^\top (R \odot X)] \right)_{ij} = \begin{cases} (1 - p)^2 (X^\top X)_{ij} & \text{if } i \neq j \\ (1 - p) (X^\top X)_{ii} & \text{else.} \end{cases}$$

We can use these two expressions to compute the objective in (7.13) to be

$$\mathbb{E}_R [\|y - (R \odot X)w\|_2^2] = \|y - (1 - p)Xw\|_2^2 + \underbrace{p(1 - p)w^\top \text{diag}(X^\top X)w}_{\Omega(w)}.$$

In other words, for linear regression, dropout is equivalent to weight-decay where the coefficient α in (7.9) depends on the diagonal of the data covariance and is different for different weights. If a particular data dimension varies a lot, i.e., $(X^\top X)_{ii}$ is large, dropout tries to squeeze its weight to zero. We can also absorb the factor of $1 - p$ into the weights w to get

$$\mathbb{E}_R [\|y - (R \odot X)w\|_2^2] = \|y - X\tilde{w}\|_2^2 + \underbrace{\left(\frac{p}{1 - p} \right) \tilde{w}^\top \text{diag}(X^\top X)\tilde{w}}_{\Omega(w)} \quad (7.14)$$

where $\tilde{w} = (1 - p)w$. This makes the regularization more explicit, if $p \approx 0$, most activations are retained by the mask and regularization is small.

Next, bagging provides a very intuitive understanding of how dropout works in a deep network at test time. We now write out the classifier explicitly as

$$f(x; w, r^k) = \sum_{i=1}^d w_i (x_i \odot r_i^k);$$

▲ Training with dropout is equivalent to introducing weight decay on the weights. Remember however that this argument is only rigorous for linear regression models (the derivation essentially remains the same for matrix factorization). This connection of dropout with weight decay will also be apparent in Module 4 when we look at how to train a Bayesian deep network.

note that the mask r^k is not a parameter of the model, we have simply chosen to make it more explicit for the sequel. We now imagine each mask as creating a *bootstrapped* version of the model; different masks r^k give different classifiers even if the weights w and the input x is the same for all.

It is important to realize that there is no subsampling of training dataset happening here like classical boosting; we are instead forming multiple models by adding randomness to how the input is propagating through the deep network. For a linear classifier this is equivalent because

$$\sum_{i=1}^d w_i (x_i \odot r_i^k) = \sum_{i=1}^d (w_i \odot r_i^k) x_k =: f(x; w^k);$$

we can either mask out the input or mask the weights and think of the masked weights w^k as a new model.

Remark 7.1. You will often see folks in the literature say that dropout regularizes by preventing co-adaptation of the neurons at each hidden layer. The motivation for this statement is that the weights of the succeeding layer cannot fixate too much upon a particular feature at the input because the feature can be zeroed out by the dropout mask. This prevents specialization of neurons in the hidden layer and ensures that the prediction is made using a large number of diverse features, not just a few specific ones. This is not a rigorous argument but it is a reasonable argument in view of the experiments of Hubel and Wiesel (see http://centennial.rucare.org/index.php?page=Neural_Basis_Visual_Perception). The human brain is quite robust to large parts of it going missing/being inhibited.

Bagging is expensive at test time, it involves having to compute the predictions of all the models in the committee. In the case of dropout, in this linear regression setup, we can compute the committee's prediction to be

$$\begin{aligned} f(x; w) &= \frac{1}{M} \sum_{k=1}^M \sum_{i=1}^d (w_i \odot r_i^k) x_k \\ &= \sum_{i=1}^d \left(w_i \odot \frac{1}{M} \sum_{k=1}^M r_i^k \right) x_k \\ &\approx \sum_{i=1}^d (w_i \odot (1 - p)) x_k. \end{aligned} \quad (7.15)$$

This is very fortunate, it indicates that given weights w of a model trained using dropout, we can compute the *committee average* over models created using dropout masks simply by scaling the weights by a factor $1 - p$. This should not be surprising, after all the equivalent training objective in (7.14) has $\tilde{w} = (1 - p)w$ as the effective weights of the weights. Another important point to note is that there is no masking of activations at test time when we scale the weights.

Although the argument in this section works only for linear models, we will bravely extend the intuition to deep networks.

7.3.3 Implementation details of dropout

The recipe for using dropout is simple: (i) the activations at the input of each dropout layer are zeroed out using a Bernoulli random variable of probability $1 - p$ of being 1; the PyTorch layer takes the probability of zeroing out activations as argument which is p in our derivations; (ii) at test time, the weights of layers immediately following dropout are scaled by a factor of $1 - p$ to compute the predictions of the “committee”.

Inverted Dropout. It is cumbersome to remember the parameter p that was used for training at test time. Deep learning libraries use a clever trick: they simply scale the output activations of the dropout layer by $1/(1 - p)$ during training. Training or testing the modified model using dropout gives an extra factor of $(1 - p)$ like (7.14) and (7.15) respectively and therefore if activations are scaled by $1/(1 - p)$ during training, then the final model can be used as is without any further scaling of the weights or activations at test time.

The operation `model.train()` in PyTorch sets the model in the training mode. This is a null-operation and does not do anything for fully-connected, convolutional, softmax etc. layers. For the dropout later, it sets a boolean variable in the layer that samples the Bernoulli mask for all the input activations and scales the output activations by $1/(1 - p)$. The complementary operation is `model.eval()` in PyTorch which you should use to set the model in evaluation mode. This is again a null-operation for other layers but for the dropout layer, it resets this boolean variable to indicate that no Bernoulli masks should be sampled and no masking should be performed.

7.3.4 Using dropout as a heuristic estimate of uncertainty

We can extend the motivation from bagging to use dropout as a cheap heuristic to get an estimate of the uncertainty of the prediction at test time. Suppose we use dropout at test time just like we do it at training time, i.e., each time one test input is presented to the deep network, we sample multiple Bernoulli masks r^1, \dots, r^M and compute multiple predictions for the same test input

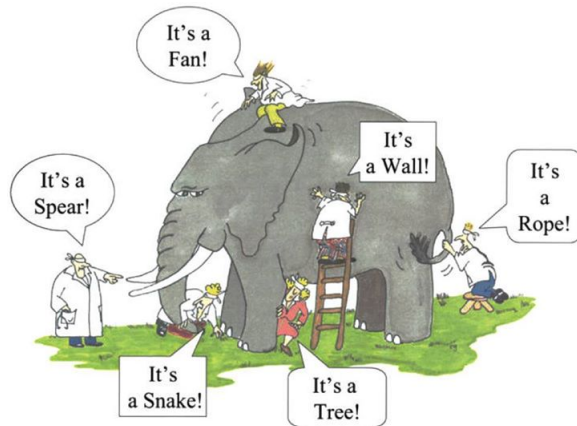
$$\{f(x; w, r^1), \dots, f(x; w, r^M)\}.$$

The variance of these predictions can be used as heuristic of the uncertainty of the deep network while making predictions on the test input x . This is an estimate of the so-called *aleatoric or statistical uncertainty*. It captures our understanding that the weights w of a trained deep network are inherently uncertain and different training experiments, in particular, different masks r^k will give rise to different weights. The variance across a few sampled masks thus indicates how uncertain the model is about its predictions. Dropout is a neat and cheap trick for this purpose; it is quite commonly used in this fashion in medical applications where it is important to not only predict the outcome but also characterize the

1 uncertainty of this prediction. We will see more powerful ways to compute
2 aleatoric uncertainty in Module 4.

3 **Remark 7.2.** Broadly speaking, the connection of dropout with weight
4 decay is not precise. If it were rigorous, we should be able to get the
5 same performance as dropout by using appropriate weight decay (this is a
6 good idea for the course project!). In practice, the validation error using
7 dropout is very good and cannot be achieved by tweaking weight decay.
8 Another aspect is that since we would like to average over lots of dropout
9 masks in the training process, networks with dropout should be trained
10 for many more iterations of SGD than networks without dropout to get
11 the same training error. The benefit is that the test error is much better
12 for dropout. What exactly dropout does is a subject of some mystery and
13 there are other alternative explanations (e.g., Bayesian dropout in Module
14 4).

15 Our understanding of dropout is no different than that of these blind
16 scientists trying to identify an elephant.



18 7.4 Batch-Normalization

19 Batch-Normalization (BN) is another layer that is very commonly used
20 in deep learning. BN is very popular with more than 20,000 citations in
21 about 5 years.

Batch normalization: Accelerating deep network training by reducing internal covariate shift

[S. Ioffe, C. Szegedy](#) - arXiv preprint arXiv:1502.03167, 2015 - arxiv.org

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and ...

☆ 99 Cited by 21278 Related articles All 32 versions Import into BibTeX »

23 7.4.1 Covariate shift

24 Covariate shift is a common problem with real data. The experimental
25 conditions under which training data was gathered are subtly different
26 from the situation in which the final model is deployed. For instance, in
27 cancer diagnosis the training set may have an over-abundance of diseased

1 patients, often of a specific subtype endemic in the location where the
 2 data was gathered. The model may be deployed in another part of the
 3 world where this subtype of cancer is not that common.

4 The mis-match between training and test *input* distribution is called
 5 covariate shift. Even if the labels depend on on the covariates in the same
 6 way, i.e., given the genetic features of a person x their likelihood of a
 7 cancer y is the same regardless of which part of the world the person is
 8 from, the fact that we do not have training data from the entire population
 9 of the world forces the classifier to be tested on a data distribution that is
 10 different from what it was trained for.

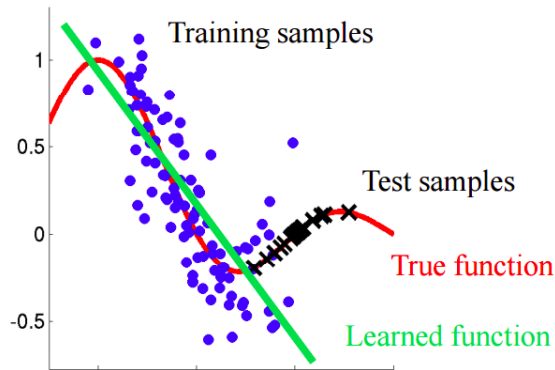


Figure 7.5: Covariate shift correction for a regression problem

11 Covariate shift is outside our fundamental assumption in Chapter 1
 12 that training and test data come from the same distribution. It is however
 13 a problem that is often (perhaps always) seen in practice and typical ways
 14 to counter it look as follows.

- 15 1. Train a classifier \hat{w} on the available training data D .
- 16 2. Update the trained classifier using data from the test distribution
 17 $D' = \{(x^i, y^i)\}_{i=n+1, \dots, n+m}$ in addition to the original training
 18 dataset

$$w^* = \operatorname{argmin}_w \frac{1}{n+m} \sum_{i=1}^{n+m} p^i \ell^i(w) + \Omega(w - \hat{w}) \quad (7.16)$$

19 where p^i is some weighing factor that indicates how similar the
 20 datum (x^i, y^i) is to the *test data distribution*. The regularization
 21 $\Omega(w - w^*)$ forces the new weights w^* to remain close to the old
 22 weights \hat{w} .

23 The above methods go under the umbrella of *doubly robust estimation*.
 24 We will not study it in this course. The results look similar to the ones
 25 shown in Figure 7.5.

7.4.2 Internal covariate shift

If we are working under the standard machine learning assumption of test data being drawn from the same distribution as that of the training data, then there is no covariate shift.

Recall that we whiten the inputs, i.e., transform the data so that its correlation matrix XX^T is identity, we linearly de-correlate the input dimensions. See [Joe Marino's webpage](#) for a good explanation of different kinds of whitening.

Deep networks are like any other model in this aspect and whitening of the inputs is also beneficial; the ZCA transform (or Mahalanobis whitening) is a close cousin of PCA and usually works better for image-based data. It is natural to expect that since each layer of a deep network takes the activations of the preceding layer as input, we should whiten the activations before the computation in the layer.

The authors of the BN paper came upon an interesting thought, but something that is clearly a mistake. Their reasoning was as follows. Say we have a mini-batch of inputs $\{x^1, \dots, x^\ell\}$ and our layer simply adds a learnable bias b to these inputs

$$h = x + b.$$

If this layer removes the mean from its output before passing it on to the next layer, we will have

$$\hat{h} := h - \frac{1}{\ell} \sum_{i=1}^{\ell} h^i$$

for $i \in \{1, \dots, \ell\}$ being the samples in the mini-batch. The output \hat{h}^i does not depend on the bias b . They argued, incorrectly, that the back-propagation update of the bias \bar{b} is equal to $\bar{\hat{h}}$. This is not true because of course

$$\bar{b} = \bar{\hat{h}} \frac{d\hat{h}}{db} = 0$$

in our notation where $\bar{h} = d\ell/dh$.

Nevertheless, the motivation of the batch-normalization operation is sound: we would like to whiten the input activations to each layer of a deep network.

Batch-Normalization is a technique for whitening the output activations of each layer in a deep network.

Naively, this would involve computing expressions of the form

$$\hat{h} = (\text{Cov}(h))^{-1/2} \left(h - \frac{1}{\ell} \sum_{i=1}^{\ell} h^i \right).$$

This is not easy to do because the features are high-dimensional vectors, the

▲ This is the mistake in the original BN paper.

the training set, and $E[x] = \frac{1}{N} \sum_{i=1}^N x_i$. If a gradient descent step ignores the dependence of $E[x]$ on b , then it will update $b \leftarrow b + \Delta b$, where $\Delta b \propto -\partial\ell/\partial\hat{x}$. Then $u + (b + \Delta b) - E[u + (b + \Delta b)] = u + b - E[u + b]$.

1 covariance matrix $\text{Cov}(h)$ is a very large matrix. This makes computing
 2 \hat{h} difficult for every mini-batch. Nevertheless, whitening helps and here is
 3 how it is done in the batch-normalization module:

$$\hat{h} = \frac{h - \mathbb{E}(\{h^1, \dots, h^\ell\})}{\sqrt{\text{Var}(\{h^1, \dots, h^\ell\}) + \epsilon}}. \quad (7.17)$$

4 The constant ϵ in the denominator prevents \hat{h} from becoming very large in
 5 magnitude if the variance is small for a particular mini-batch. It is important
 6 to note that both the expectation and the variance are computed for every
 7 feature. Let us make this clear: if $h \in \mathbb{R}^{\ell \times p}$, i.e., p features for this layer,
 8 the $i^{\text{th}} \in \{1, \dots, \ell\}$ input of the mini-batch and the $j^{\text{th}} \in \{1, \dots, p\}$ of
 9 the feature for \hat{h} is given by

$$\hat{h}_{ij} = \frac{\hat{h}_{ij} - \frac{1}{\ell} \sum_{i=1}^{\ell} h_{ij}}{\sqrt{\text{Var}(\{h_{1j}, h_{2j}, \dots, h_{\ell j}\}) + \epsilon}}.$$

10 Let us give names to these parameters

$$\begin{aligned} \mathbb{R}^p \ni \mu &= \mathbb{E}(\{h^1, \dots, h^\ell\}) \\ \mathbb{R}^p \ni \sigma^2 &= \text{Var}(\{h^1, \dots, h^\ell\}). \end{aligned} \quad (7.18)$$

11 The authors of the original BN paper felt that mere normalization is not
 12 enough, e.g., if you normalize the activations *after a sigmoid activation*, the
 13 layer may essentially become linear because the activations are prevented
 14 from going too far to the right or too far too the left of the origin. This
 15 brings the second idea in BN, that of affine scaling of the output \hat{h} . The
 16 BN layer implements

$$\hat{h} = a \odot \left(\frac{h - \mathbb{E}(\{h^1, \dots, h^\ell\})}{\sqrt{\text{Var}(\{h^1, \dots, h^\ell\}) + \epsilon}} \right) + \odot b. \quad (7.19)$$

17 where $a, b \in \mathbb{R}^p$, i.e., each feature has its own multiplier a and bias b . The
 18 final BN operation in short is

$$\hat{h} = a \left(\frac{h - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + b.$$

The affine scaling parameters a, b are the only trainable parameters

in BN that are updated using back-propagation. The mean μ and variance σ^2 are unique to every mini-batch and therefore do not have any back-propagation gradient.

Execute the following code and check how the BN layer is implemented in PyTorch

```
import torch.nn as nn
m = nn.BatchNorm1d(15)
print(m.weight, m.bias)
print(m.running_mean, m.running_var)
```

The weight and bias here are the affine scaling parameters; and `running_mean`, `running_var` are μ, σ^2 respectively. You will see that `requires_grad` is True only for the former.

1 BN for convolutional layers

2 The activations of a convolutional layer are a 4-dimensional array

$$h \in \mathbb{R}^{\beta \times c \times w \times h}.$$

3 The distinction between convolutional layers compared to fully-connected
4 layers is that the convolutional filter weights are shared for the whole input
5 channel $w \times h$. We can therefore think of each *channel* as a *feature* and
6 compute the BN mean and standard deviation over the batch dimension,
7 as well as the width and height. In pseudo-code, this looks as follows.

```
8
9 # t is still the incoming tensor of shape [bb, c, w, H]
10 # but mean and stddev are computed along (0, 2, 3) axes and
11 # have just [c] shape
12 mean = mean(t, axis=(0, 2, 3))
13 stddev = stddev(t, axis=(0, 2, 3))
14 for i in 0..bb-1, x in 0..h-1, y in 0..w-1:
15     out[i,:,x,y] = normalize(t[i,:,x,y], mean, stddev)
```

17 Running updates of the mean and variance in BN

18 BN computes the statistics over mini-batches. Even if we trained a model
19 using mini-batch updates we would still like to be able to use this model
20 at test time with a single input; it may not always be possible to wait for
21 a few test images to make predictions. The weights of the network are
22 trained to work with whitened features so we definitely need some way to
23 whiten the features of a test input, ignoring the whitening at test time will
24 result in wrong predictions.

25 The BN layer solves this issue by maintaining a running average of the
26 mean and variance statistics of mini-batches during training. Effectively,
27 the buffers `running_mean`, `running_var` (note that these are not parameter-
28 s/weights, they are not updated using backprop) are updated after *each*

1 *mini-batch* during training as

$$\begin{aligned}\text{running_mean}^{t+1} &= \rho \text{running_mean}^t + (1 - \rho) \mu \\ \text{running_var}^{t+1} &= \rho \text{running_var}^t + (1 - \rho) \sigma^2.\end{aligned}$$

2 The parameter ρ is called a momentum parameter for the BN layer and
3 makes sure that updates to `running_mean/var` are slow and one mini-batch
4 cannot affect the stored value too much. Note that whitening is still
5 performed at training time using μ, σ^2 ; we simply record the running
6 average in the buffers `running_mean/var`. If `model.train()` is called, then
7 the mini-batch statistics are used to whiten the features. If `model.eval()` is
8 called, then the stored buffers `running_mean/var` are used to whiten the
9 outputs.

10 How is all this related to internal covariate shift?

11 You might be surprised that nothing in this section is related to covariate
12 shift that we discussed at the beginning. Let us try to understand
13 heuristically why BN is said to help with internal covariate shift.

14 Each layer of a deep network treats its input activations as the data
15 and predicts the output activations. As the weights of different layers
16 are updated using backprop during training, the *distribution* of input
17 activations keeps shifting. Effectively, each layer is constantly suffering a
18 covariate shift because the layers below it are updated and the weights of
19 the top layers have to adapt to this shifting distribution. This is what is
20 known as *internal covariate shift*. BN normalizes the output activations
21 to approximately have zero mean and unit variance and therefore reduces
22 the internal covariate shift.

23 7.4.3 Problems with batch-normalization

24 There are two big problems with BN.

- 25 1. The affine parameters are updated using backpropagation and small
26 changes to mini-batch statistics can result in large changes to the
27 whitened output $(h - \mu) / \sqrt{\sigma^2 + \epsilon}$. This will result in very large
28 updates to a, b . This makes the affine parameters problematic
29 when you train networks. In general, it is a good idea to first fit a
30 model without the affine BN parameters, you can do so by using
31 `affine=False` in `nn.BatchNorm1d`.
- 32 2. The mean and variance buffers of the BN layer are updated using
33 running statistics of the per-mini-batch statistics. This does not
34 affect training because the statistics of each mini-batch are computed
35 independently, but it does affect evaluation because the buffers are
36 used to whiten the features of the test input. If the test input has
37 slightly different pixel intensity statistics than the training image,
38 then the BN buffers are not ideal for whitening and such images are
39 classified incorrectly.

▲ There are many caveats with this heuristic argument. The main one is to observe that the backpropagation gradient of all layers is coupled, so it is not as if the layers are updated independently of each other and cause interval covariate shifts to the other layers; the updates of all the weights in the network are coupled and it is unclear why (or even if) internal covariate shift occurs.

1 **BN before ReLU or ReLU before BN**

2 Should we apply BN before or after the nonlinearity? The purpose of
3 a BN layer is to keep the activations close to zero in their mean and a
4 standard-deviation of one. Imagine if we are using a ReLU nonlinearity
5 after BN, about half of our features h have negative values which the
6 rectification will set to zero. In this case the distribution of features given
7 to the next layer is not zero-mean, unit-variance so we are not achieving
8 our goal of whitening correctly. Further, it is possible that the bias
9 parameter b in BN is negative in which case the activations could mostly
10 be negative and ReLU will set all of them to zero and result in a large loss
11 in information. On the other hand, if we have BN after ReLU, the input to
12 the BN layer has a lot of zeros and we are now computing mean/variance
13 over a number of sparse features; the mini-batch mean/variance estimated
14 here may not be accurate therefore BN may not perform its job of correctly
15 whitening its outputs. You can read more about similar problems at
16 <http://torch.ch/blog/2016/02/04/resnets.html>

17 As you can see, BN is an incredibly intricate operation without
18 necessarily sound theoretical foundation for all the moving parts. But it
19 works, training a deep fully-connected network is very difficult without
20 BN, and even for convolutional layers it often makes training insensitive to
21 the choice of learning rate. You should think about BN very carefully in
22 your implementations; a lot of problems of the kind, “I trained my model,
23 it gives a good training error but very poor validation error”, or “I am fine-
24 tuning from this task, but get very poor validation error on a new task”, or
25 other problems in reinforcement learning, meta-learning, transfer learning
26 etc. can be boiled down to an incorrect/inadequate understanding of batch-
27 normalization. This is further complicated by the interaction with other
28 operations such as Dropout, e.g., see <https://arxiv.org/abs/1801.05134>.
29 Studying the effect of BN in meta-learning/transfer-learning is a good idea
30 for a course project.

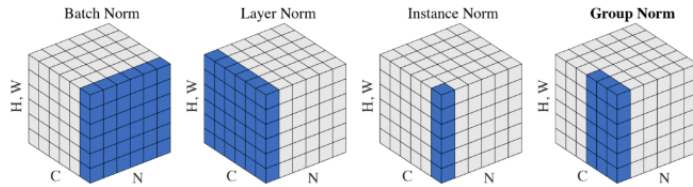
31 **How does Dropout affect BN?**

32 Since dropout is active during training, the buffered statistics are the
33 running mean/variance of the dropped out activations. Dropout is not
34 used at test time, so the test time statistics, even for the same image can be
35 quite different. A simple way to solve this problem is to run the model in
36 training mode once on the validation set (without making weight updates
37 using backpropagation) for the BN buffers to settle to their non-dropped
38 out values and then compute the validation error; this usually results in a
39 marginal improvement in the validation error.

40 **7.4.4 Variants of Batch-Normalization**

41 There are variants of batch-normalization that have cropped out to alleviate
42 some of its difficulties. For instance, layer normalization
43 (<https://arxiv.org/abs/1607.06450>) normalizes across the features instead
44 of the mini-batch which makes it work better for small mini-batches. An-
45 other variant known as group-normalization computes the mean/variance

1 estimate in BN across multiple partitions of the mini-batch which makes
2 the result of group-normalization independent of the batch-size. These
3 variants work in some cases and do not work in some cases and often
4 the specific normalization is largely dependent on the problem domain,
5 e.g., group normalization works better for image segmentation but layer
6 normalization and batch-normalization do not so well there.



The major differences among 4 normalization methods. The blue pixels are normalized by the same mean and variance, computed on the values of these pixels.

7

1 Chapter 8

2 Recurrent Architectures

3 and the Attention

4 Mechanism

Reading

1. Goodfellow 10.1-10.3, 10.5-10.7, 10.9-10.12
2. D2L.ai book Chapters 8, 9, 10
3. Paper on long short-term memory ([Hochreiter and Schmidhuber, 1997](#))
4. Paper on the Transformer architecture ([Vaswani et al., 2017](#))
5. Paper on CLIP ([Radford et al., 2021](#))

5 In this chapter we will consider data that evolves with time. Typical
6 examples of such data are videos and sentences in written/spoken language.
7 Some typical problems that we are interested in solving given such data
8 are classifying the activity going on in a video, classifying the object that
9 is being described in a sentence, etc. We can also think of generative
10 models for such temporal data, i.e., forecasting how the video/sentence
11 will look like a few time-steps into the future using the approaches in this
12 chapter.

13 We will look at three kinds of neural architectures, namely Recurrent
14 Neural Networks (RNNs), and the Long Short-Term Memory (LSTM)
15 and Attention modules, that are typically used to model such data.

8.1 Recursive updates in a Kalman filter, sufficient statistics

Consider a scalar signal in time $h_t \in \mathbb{R}$ that evolves according to some dynamics

$$h_{t+1} = ah_t + \xi_t;$$

with the scalar $a \in \mathbb{R}$ that we have modeled and the noise $\xi_t \in \mathbb{R}$ reflects our understanding that the scalar a in our model of evolution of the signal h_t may not be the same as that of Nature. We model this discrepancy by setting ξ_t to be zero-mean Gaussian noise that is i.i.d across time

$$\xi_t \sim N(0, \sigma_\xi^2).$$

Let us say that our dataset consists of observing the signal for some time $\{x_1, x_2, \dots, x_t\}$. Think of h_t being the location of a car at time t and our dataset being the observation of the trajectory of vehicle up to time t . Assume that we do not observe the true trajectory of the vehicle, but observe some noisy estimate of the state at each time

$$x_t = h_t + \nu_t$$

where $\nu_t \sim N(0, \sigma_\nu^2)$ is the noise in our observation.

In this section, we will estimate the true signal at the next time instant \hat{h}_{t+1} . A good estimate is the one that minimizes the MSE loss with the true (unknown) signal

$$\operatorname{argmin}_{\hat{h}_{t+1}} \mathbb{E}_{\xi_1, \nu_1, \dots, \xi_{t+1}, \nu_{t+1}} \left[\left(h_{t+1} - \hat{h}_{t+1} \right)^2 \mid \underbrace{x_1, \dots, x_t, x_{t+1}}_{\text{"dataset"}} \right]. \quad (8.1)$$

The expectation is taken over the noise because there could be many trajectories that the system could have taken, each corresponding to a particular realization of the noise.

Our estimate should only depend on the dataset

$$\hat{h}_{t+1} = \text{function}(x_1, \dots, x_t, x_{t+1}).$$

Since predictions are likely to be required across a long range of time, we want to construct a *recursive* update for \hat{h}_{t+1} that takes in the estimate at the previous time-step \hat{h}_t and updates it using the most recent observation x_{t+1} .

Kalman filter updates the sufficient statistic

Like we computed the optimal predictor in the bias-variance tradeoff for regression as the conditional distribution of the labels given the data, it is possible to prove that the best estimate \hat{h}_{t+1} is the conditional mean given past data

$$\hat{h}_{t+1} = \mathbb{E}[h_{t+1} \mid x_1, x_2, \dots, x_{t+1}].$$

▲ In machine learning parlance, this setup is called online learning where data occur sequentially one after other and you train/update the model to incorporate the latest datum; future predictions of this model are made using this updated model.

1 Not surprisingly, to estimate the location of the car at time $t + 1$, you need
 2 to watch the entire past trajectory of the car.

3 A major result in control theory is that for our problem (where the model
 4 of the signal is linear with additive Gaussian noise and our observations
 5 x_t are a linear function of h_t corrupted with Gaussian noise) we only need
 6 to recursively update of the first two moments of our estimate. If we have

$$\hat{h}_{t+1} = N(\mu_{t+1}, \sigma_{t+1}^2)$$

7 where

$$\begin{aligned} \mu_{t+1} &= \mathbb{E} \left[\hat{h}_{t+1} \mid x_1, \dots, x_{t+1} \right] \\ \sigma_{t+1} &= \text{var} \left(\hat{h}_{t+1} \mid x_1, \dots, x_{t+1} \right). \end{aligned} \quad (8.2)$$

8 and update the mean and variance recursively using their values at the
 9 previous time-step as

$$\begin{aligned} \mu_{t+1} &= a\mu_t + k_t(x_{t+1} - a\mu_t) \\ \sigma_{t+1}^2 &= (a^2\sigma_t^2 + \sigma_\xi^2)(1 - k_t) \\ k_t &= \frac{a^2\sigma_t^2 + \sigma_\xi^2}{a^2\sigma_t^2 + \sigma_\xi^2 + \sigma_\nu^2}. \end{aligned} \quad (8.3)$$

10 You can derive this part very easily. Show that if the objective in (8.1)
 11 was optimal at time t for \hat{h}_t in (8.3), then the expressions in (8.3) also
 12 minimizes the objective at time $t + 1$. This algorithm is known as the
 13 Kalman filter is one of the most widely used algorithms for estimation of
 14 signals based on their observation. The key property to remember for us
 15 from the Kalman filter is the following.

The two quantities μ_t, σ_t capture *all* the information from the past trajectory x_1, \dots, x_t . Instead of creating our MSE estimate \hat{h}_t using the entire data as shown in (8.1) each time instant, if we maintain these two quantities and recursively update them using (8.3) we obtain the best MSE estimate.

In other words, μ_t, σ_t are sufficient statistics of the data x_1, \dots, x_t for the problem of estimating the next state h_{t+1} . The notion of a *sufficient statistic* means that you do not need anything beyond these two functions of the data x_1, \dots, x_{t+1} to estimate h_{t+1} .

A statistic is simply any function of data. Therefore a sufficient statistic is a quantity such that if you have it, you can throw away the data without losing any information. Not all statistics are sufficient, and not all sufficient statistics look like a few moments of data. For more interesting signals the sufficient statistics are non-trivial and difficult to find.

The structure of neural architectures for sequence modeling is

intimately related to the above result. Just like a CNN learns features that are “sufficient” to classify the input data, a recurrent model learns the statistics of the past sequence that are sufficient to predict future elements.

8.2 Recurrent Neural Networks (RNNs)

The data to an RNN is a set of n sequences

$$D = \{(x_1^i, y_1^i), (x_2^i, y_2^i), \dots, (x_T^i, y_T^i)\}_{i=1, \dots, n}.$$

Each sequence has length T and each element of the sequence $x_t^i \in \mathbb{R}^d$. There can be labels at every time-step, e.g., these labels can be, say, ground-truth annotations of the activity “playing with a basketball” going on the video at that time, or also forecasting the inputs by one (or more) time-steps $y_t^i \equiv x_{t+1}^i$.

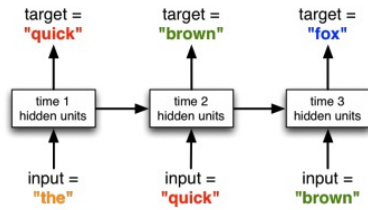


Figure 8.1: A recurrent model predicting the next word in a sentence.

Let us focus on one particular sequence $\{(x_1^i, y_1^i), \dots, (x_T^i, y_T^i)\}$ from the dataset. To predict the labels y_t^i , the RNN maintains a statistic, let us denote it by

$$h_t^i = \varphi((x_1^i, y_1^i), \dots, (x_t^i, y_t^i)).$$

Here φ is some function that we would like to build. Similar to a Kalman filter we *hope to learn* a sufficient statistic. In this case sufficiency means that the quantity h_t can predict the target y_t . Again, we would like to update the statistic recursively.

$$h_{t+1} = \varphi(h_t, x_{t+1}); \quad (8.4)$$

notice the similarity with the updates in (8.3) where updates to μ_t, σ_t also used the latest observation x_{t+1} . We will also have the RNN use the latest input x_{t+1} . You can think of h_t as a summary of the past sequence or some memory that is updated recursively. This summary/statistic is also called the “hidden state” in the RNN literature.

We do not know what function φ to pick (for the Kalman filter we knew that it is the conditional mean/variance of h_t given past observations)

▲ Note that just like we cannot claim that the features learned by a CNN are sufficient features, i.e., the only information from the data necessary to predict the targets, we cannot *claim* that h_t is a sufficient statistic of the past sequence. If the RNN/CNN are making predictions accurately, then it is reasonable to expect that we have learned something close to a sufficient statistic.

1 so we are going to learn it using parameters. We will set

$$h_{t+1} = \sigma(w_h h_t + w_x x_{t+1}); \quad (8.5)$$

2 where $w_h \in \mathbb{R}^{p \times p}$, $w_x \in \mathbb{R}^{p \times d}$ are weights that multiply the previous
3 statistic and the current input to calculate the current statistic. Again $\sigma(\cdot)$
4 is a nonlinearity that is applied element-wise.

5 **Weights of an RNN are not a function of time** It is important to
6 observe that the weights w_h, w_x do not change as the sequence moves
7 forward. The same function is used to update the statistic at different
8 points of time; notice that this does not mean that the statistic h_t^i remains
9 the same across t . In this sense, an RNN is effectively the same neural
10 model unrolled into the future as it takes in inputs of a sequence.

11 **Training objective for an RNN** Output predictions can now be made
12 as usual by learning weights

$$\hat{y}_t^i = v^\top h_t^i. \quad (8.6)$$

13 The loss function of an RNN is a sum of the error in the predictions for all
14 time-steps for all samples

$$\frac{1}{nT} \sum_{i=1}^n \sum_{t=1}^T \ell(y_t^i, \hat{y}_t^i) \quad (8.7)$$

15 and we can train the RNN by updating weights w_h, w_x using back-
16 propagation. In some problems, you may only have targets for the final
17 time-step y_T^i (say predicting whether it is going to rain right now or not
18 based on the weather data of the past few hours). This does not change
19 things much conceptually, we will simply have only one term in the
20 summation above.

21 The exponential of the cross-entropy loss (which is equal to 1 divided
22 by the likelihood of the dataset) is called the “perplexity” in natural
23 language processing problems

$$\text{perplexity} = \exp \left(-\frac{1}{nT} \sum_{i=1}^n \sum_{t=1}^T \sum_{c=1}^C \mathbf{1}_{\{y_t^i=c\}} \log p(\hat{y}_t^i = c) \right);$$

24 where the sum over c ranges over the unique elements of the sequence (e.g.,
25 words/tokens etc.). You will see the perplexity being calculated in base
26 2 many times. The reason for this is as follows. The cross-entropy loss
27 in (8.7) can be interpreted as the number of bits (actually this is called “nats”
28 since we wrote the loss using the natural logarithm) required to predict
29 each element of the sequence. If the cross-entropy loss per time-step
30 (which is what we have written in (8.7)) is 3.55, then the perplexity

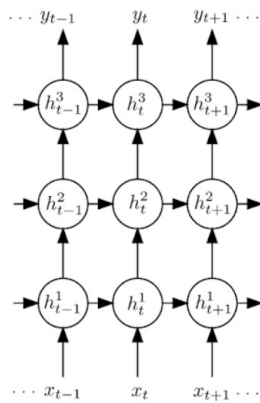
$$2^{3.55} \approx 11.7,$$

▲ If each element of a sequence x_t is distributed uniformly over C possible values, i.e., the sequence is perfectly random with no predictable patterns within it, then the cross-entropy loss for predicting the next element of such a sequence will be equal to the Bayes error, which is $-\log C$. The perplexity $\exp(-\log C) = C$ is therefore the number of distinct alternatives that the learned model predicts over when it predicts the next token.

1 and therefore the learned model is confused as if it had to choose uniformly
 2 among about 12 different possibilities. Smaller the perplexity, better the
 3 predictive model.

4 Multi-layer RNNs

5 We have created a single-layer RNN in (8.5). We can use the same idea to
 6 create a multi-layer RNN the same way that we did for CNNs. We combine
 7 different parts of the hidden state/statistic and use these as features. In an
 8 RNN, it is traditional to combine the features both from the lower layer
 9 and features from the previous time-step of the same layer. As a picture it
 looks as follows



10

11 We can write an expression for this as

$$h_t^{l+1} = \sigma(w_{tt}^l h_{t-1}^{l+1} + w_{hh}^l h_t^l).$$

12 Again we have used trainable weights $w_{tt} \in \mathbb{R}^{p \times p}$ and $w_{hh} \in \mathbb{R}^{p \times p}$
 13 to compute the hidden state/statistic/activations of the top layer. For a
 14 multi-layer RNN with L layers, the predictions at each time step are given
 15 by

$$\hat{y}_t = v^\top h_t^L.$$

16 The utility of having multiple layers in an RNN is similar to that of a
 17 CNN, more layers let us create more complex predictors than the recurrent
 18 perceptron-style predictor in (8.6) by learning a richer set of features.

19 8.2.1 Backpropagation in an RNN

20 Let us see how to compute the gradient of the loss function with respect
 21 to the weights of an RNN in order to train the model using SGD. We will

🔗 How should we initialize the first hidden vector h_0 in an RNN? We have not seen any element of the sequence yet, so the value of h_0 has no meaning per se. Typically, h_0 is initialized either using Gaussian noise or simply to zeros.

1 consider a sequence of two time-steps for a single-layer RNN

$$\begin{aligned}
 h_1 &= \sigma(ux_1) \quad \text{where we set } h_0 = 0 \\
 \hat{y}_1 &= vh_1 \\
 h_2 &= \sigma(ux_2 + wh_1) \\
 \hat{y}_2 &= vh_2
 \end{aligned}
 \tag{8.8}$$

2 The weights we would like to update are u, v and w . Let us say that
 3 the loss function is only computed at the final time-step $t = 2$ as $\ell :=$
 4 $\ell(y_2, \hat{y}_2) = \|y_2 - \hat{y}_2\|^2$. Using our notation for backpropagation we have

$$\begin{aligned}
 \frac{d\ell}{d\ell} &= \bar{\ell} = 1 \\
 \bar{\hat{y}}_2 &= \bar{\ell} \frac{d\ell}{d\hat{y}_2} \\
 &= -(y_2 - \hat{y}_2) \\
 \bar{v} &= \bar{\hat{y}}_2 \frac{d\hat{y}_2}{dv} \\
 &= -(y_2 - \hat{y}_2) h_2 \\
 \bar{h}_2 &= \bar{\hat{y}}_2 v \\
 \bar{u} &= \bar{h}_2 \sigma'(ux_2 + wh_1) \left(x_2 + w \frac{dh_1}{du} \right) \\
 &\vdots
 \end{aligned}$$

5 You should write down the update steps completely for an RNN making
 6 predictions at each time-step, using the loss function

$$\ell = \|y_1 - \hat{y}_1\|^2 + \|y_2 - \hat{y}_2\|^2$$

7 and see how the gradient of the loss at each time-step with respect
 8 to weights “accumulates” in \bar{w}, \bar{v} and \bar{u} . Backpropagation in RNNs
 9 is also called backpropagation-through-time (BPTT). There is nothing
 10 special going on inside BPTT, it is simply backpropagation applied to a
 11 computational graph that is unrolled in time.

12 8.2.2 Handling long-term temporal dependencies

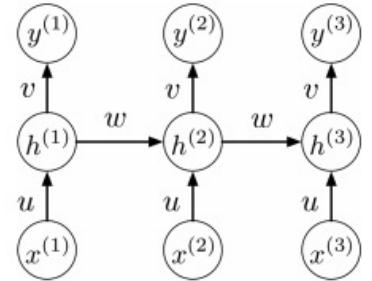
13 Implementation of BPTT for RNNs has a number of numerical issues.

14 Gradient vanishing

15 Notice that the gradient

$$\begin{aligned}
 \bar{u} &= \bar{h}_2 \sigma'(ux_2 + wh_1) \left(x_2 + w \frac{dh_1}{du} \right) \\
 &= -(y_2 - \hat{y}_2)v \sigma'(ux_2 + wh_1) \left(x_2 + w \frac{dh_1}{du} \right)
 \end{aligned}$$

▲ Computational graph of a single-layer RNN. Please ignore the notation in this figure and see (8.8).



▲ Note that expression for \bar{u} where we get a term $\frac{dh_1}{du}$. Such terms are coded up in Pytorch using gradient accumulation, i.e., if the same function is used twice in the forward propagation, the gradient \bar{u} of its parameters will be accumulated for each of the backpropagation calls.

1 in our backprop equations depends on the gradient of non-linearity. If we
 2 have a sigmoid non-linearity and if the input activations to it $ux_2 + wh_1$
 3 have large magnitude, the output h_2 will be saturated. This results in
 4 \bar{u}, \bar{h}_2 having small magnitudes. Further notice that \bar{u} also depends upon
 5 products of the weights v and the inputs x_2 . If you unroll this further
 6 for a few more time-steps (like we did in HW2) you will see that future
 7 activations h_t are recursive products of past activations with weights. It is
 8 easy to observe that if we have a matrix A and a vector x the product

$$\lim_{k \rightarrow \infty} A^k x \quad (8.9)$$

9 goes to zero if the largest singular of A is less than 1, i.e., $\lambda_{\max} = \|A\|_2 < 1$.
 10 The product goes to positive/negative infinity if the largest singular value
 11 is greater than 1 if x has a non-zero inner product with the corresponding
 12 singular vector. In other words, if the length of the sequence is long, it is
 13 due to the recursive computation in an RNN that the activations can blow
 14 up to infinity. This can also lead to gradient explosion. The activations
 15 can also become zero which can result in gradient vanishing.

16 All this is also true for CNNs with many layers: the weights of the
 17 lower layers get their backprop gradient after it goes through multiple
 18 nonlinearities (ReLU leads to saturation as well if the input is negative)
 19 and can therefore receive a small gradient. While typical CNNs have 10
 20 or so layers, typical RNNs handle sequences of length 50–100 (or more).
 21 The chance of having vanishing gradients to the weights is thus much
 22 higher in RNNs.

23 Propagation of information in BPTT

24 You would think that if the objective is a sum of the loss at each time; this
 25 alleviates the problem of gradient vanishing. But there is a deeper point
 26 we are trying to make here. The backprop gradient is an indication of
 27 how much we should change u, h_2 to make more accurate predictions at
 28 some future time-step y_t . If $t \gg 2$, the value of h_2 does not play a strong
 29 role in making predictions too far into the future. In other words, the
 30 predictions of the RNN become myopic we do not learn statistics that are
 31 a function of the entire past trajectory, the statistics are highly dominated
 32 by the near past which makes it difficult to capture long-range correlations
 33 in the sequence and predict high-level concepts.

34 Which nonlinearities are good for RNNs?

35 Think about which nonlinearities are good for training RNNs. Gradient
 36 vanishing is a large problem with sigmoids whereas both gradient vanishing
 37 and gradient explosion can occur for ReLU nonlinearities. You might be
 38 tempted to design a nonlinearity that does not saturate on either side of
 39 the origin but such nonlinearities look closer to and closer to an identity
 40 mapping and, as we have seen, a linear model is much less powerful
 41 than a nonlinear model. In other words, gradient explosion/vanishing is a
 42 problem in BPTT for RNNs but there is really no effective solution to it.

1 Gradient clipping

2 We can avoid gradient explosion ruining the weights being updated by
 3 using gradient clipping. There are many ways of implementing this
 4 idea. The most prevalent one is to clip the ℓ_2 norm of the gradient to a
 5 pre-specified value. The SGD update is modified to be

$$w^{(t+1)} = w^{(t)} - \eta \text{clip}_c(\nabla \ell^{\omega_t}(w^{(t)}))$$

6 where $\nabla \ell^{\omega_t}(w^{(t)})$ is the gradient of the objective on the sample with
 7 index $\omega_t \in \{1, \dots, n\}$ in the dataset computed at weights ω_t and clipping
 8 performs the operation

$$\text{clip}_c(v) = \frac{cv}{\|v\|_2 + \epsilon}$$

9 where c is a pre-specified value and it is the ℓ_2 norm of the clipped
 10 gradient. The scalar ϵ in the denominator prevents numerical issues when
 11 the gradient magnitude is small.

12 Sometimes you instead clip the per-weight gradient at values $[-c, c]$,
 13 i.e., if the gradient vector is $v \in \mathbb{R}^p$ and v_k is the gradient at the k^{th}
 14 element

$$\text{clip}_c(v) = [\min(\max(-c, v_1), c), \dots, \min(\max(-c, v_p), c)].$$

15 Orthonormal initialization of weights

16 If A is an orthonormal matrix, we have

$$A^\top A = I.$$

17 All singular values of an orthonormal matrix have an absolute value
 18 of 1. This helps when we perform repeated multiplication with the
 19 weight matrices in forward-backward propagation because the norm of
 20 the intermediate products does not change

$$\|A^k x\|_2 = \|x\|$$

21 if A is orthogonal. The weight matrices of an RNN are typically initialized
 22 as orthogonal matrices; this is easy to do by first initializing the matrix
 23 using random Gaussian entries as usual and then setting the actual weights
 24 to be the left singular vectors after computing an SVD of the matrix.

25 Moving window over the data

26 We wrote down SGD updates as sampling a random (input,target) pair
 27 from the dataset at each iteration. The data for an RNN consists of a
 28 number of trajectories/sequences. We can sample one (or a mini-batch) of
 29 such sequences and a contiguous chunk of each of those sequences as a

▲ The function `clip_grad_norm` performs gradient clipping. When you observe it closely you will realize that it is really scaling the gradient and should therefore be called gradient scaling.

❓ If the weights of an RNN are initialized as orthogonal matrices, do they remain so after multiple steps of SGD?

1 mini-batch in an RNN

$$D_{\text{mini-batch}} = \{(x_1^i, y_1^i), \dots, (x_{25}^i, y_{25}^i)\} \cup \\ \{(x_5^j, y_5^j), \dots, (x_{30}^j, y_{30}^j)\} \cup \\ \{(x_{13}^k, y_{13}^k), \dots, (x_{38}^k, y_{38}^k)\} \cup \\ \vdots$$

2 The hidden state h_0 of the RNN can be initialized to zero/randomly at the
3 beginning for all these trajectories.

4 We can however also play a neat trick while sampling mini-batches in
5 an RNN to give it the ability to handle more long-range correlations. The
6 mini-batch is treated as a moving window over the data and it is rolled
7 forward sequentially, i.e.,

$$D_{\text{mini-batch 1}} = \{(x_1^i, y_1^i), \dots, (x_{25}^i, y_{25}^i)\} \cup \\ \{(x_1^j, y_1^j), \dots, (x_{25}^j, y_{25}^j)\} \cup \\ \{(x_1^k, y_1^k), \dots, (x_{25}^k, y_{25}^k)\} \cup \dots$$

8 and the next mini-batch is chosen to be

$$D_{\text{mini-batch 2}} = \{(x_{26}^i, y_{26}^i), \dots, (x_{50}^i, y_{50}^i)\} \cup \\ \{(x_{26}^j, y_{26}^j), \dots, (x_{50}^j, y_{50}^j)\} \cup \\ \{(x_{26}^k, y_{26}^k), \dots, (x_{50}^k, y_{50}^k)\} \cup \dots$$

9 In this case, we simply copy the hidden state/statistic h_{25} of the previous
10 mini-batch as the initialization h_0 for the next mini-batch. While this
11 creates strong correlations in the consecutive mini-batches and data for
12 SGD is not sampled iid, it is a useful trick to increase the effective range of
13 temporal correlations modeled in the RNN without essentially any special
14 operations. You can see an implementation of this idea at
15 https://github.com/pytorch/examples/blob/master/word_language_model/main.py#L131
16

Roughly speaking, data that consists of sequences of length up to 25 can be trained with RNNs.

17 8.3 Long Short-Term Memory (LSTM)

18 Innovations on top of the basic RNN architecture try to improve their ability
19 to handle long-range correlations in the data. We saw that the updates to
20 the hidden state/statistic h_t is the key to doing so. The architectures called
21 LSTMs, and their simpler counterparts called GRUs, are mechanisms that
22 give us more control to update the hidden state.

8.3.1 Gated Recurrent Units (GRUs)

GRUs “gate” the hidden state, i.e., the architecture has a mechanism to control when the hidden state gets updated and when it does not. For instance, if the first symbol in our sequence is very predictive of the future of the sequence we want the RNN to learn to not update the hidden state, and similarly if there are irrelevant words in the middle of the sequence we want the hidden state to not be updated at those time-steps. A GRU also has a mechanism to “reset” the hidden state that reduces the influence of the previous hidden state on the next hidden state.

Recall that the hidden state for an RNN with a single layer is updated as

$$h_{t+1} = \sigma(w_h h_t + w_x x_{t+1}).$$

A GRU has two more variables that are called the reset variable and the zero variable respectively, each created from previous x_t, h_t using learnable weights

$$\begin{aligned} r_{t+1} &= \text{sigmoid}(w_{xr} x_t + w_{hr} h_t) \\ z_{t+1} &= \text{sigmoid}(w_{xz} x_t + w_{hz} h_t). \end{aligned} \quad (8.10)$$

The entries of r_t, z_t are between $(0, 1)$. The update to the hidden state in an RNN is modified to be

$$h_{t+1} = (1 - z_{t+1})h_t + z_{t+1} \odot \tanh(w_h (r_{t+1} \odot h_t) + w_x x_{t+1}). \quad (8.11)$$

If entries of z_{t+1} are close to 0, the old state is propagated almost unchanged to result in h_{t+1} ; information from x_{t+1} is essentially ignored in this case. If entries of z_{t+1} are close to 1, the reset gate is used to decide what the next state h_{t+1} is: if r_{t+1} is close to one, then the update is the same as that of a conventional RNN; if r_{t+1} is close to zero, then the previous hidden state does not play any role in the update and the update is only dependent on the observation x_{t+1} .

8.3.2 LSTMs

The design of an LSTM was inspired by logic gates in a computer and is a bit complicated. The original LSTM paper is an assigned reading for this lecture. LSTMs are powerful models in sequence modeling and in spite of being developed all the way back in 1997, they are among the few deep learning models that remained popular through the second AI winter and are still the workhorse of the NLP industry today.

An LSTM has three new variables on top of an RNN, these are called the “input, forget, and output” gates respectively

$$\begin{aligned} i_{t+1} &= \sigma(w_{hi} h_t + w_{xi} x_{t+1}) \\ f_{t+1} &= \sigma(w_{hf} h_t + w_{xf} x_{t+1}) \\ o_{t+1} &= \sigma(w_{ho} h_t + w_{xo} x_{t+1}) \end{aligned} \quad (8.12)$$

▲ The idea that the hidden state is the memory in sequence models is more clear in this context. In some cases we may want to update our memory after observing a particular part of the sequence, in some cases we want to keep the memory unchanged while in some cases we may wish to reinitialize the memory before observing the future data.

▲ There are many variants of a GRU which will reduce the number of operations used in the GRU. For example, Light GRU, does not use a reset gate and uses ReLU instead of tanh (denoted here as $(\cdot)_+$), along with batch-normalization. This is what most

$$\begin{aligned} z_{t+1} &= (\text{bn}(w_{xz} x_t) + w_{hz} h_t)_+ \\ h_{t+1} &= (1 - z_{t+1})h_t + z_{t+1} \odot \\ &\quad (w_h h_t + \text{bn}(w_x x_{t+1}))_+. \end{aligned}$$

▲ GRUs are very useful recurrent models because they are more general than RNNs but at the same time much simpler than other models such as LSTMs. In most cases, it is a good idea to first try to fit the data using a GRU before using more complex models.

1 where all the above weight matrices are learnable parameters. In the GRU
 2 we had the convex combination using the zero gate in (8.11) to prevent
 3 forgetting. In an LSTM we use the two gates f_t, i_t for this purpose. The
 4 hidden state of an LSTM is propagated as

$$h_{t+1} = o_{t+1} \odot c_{t+1} \quad (8.13)$$

5 where the variable

$$c_{t+1} = f_{t+1} \odot c_t + i_{t+1} \odot \tanh(w_{hc} h_t + w_{xc} x_{t+1}) \quad (8.14)$$

6 is thought of as a memory cell. Understanding crisply what an LSTM ought
 7 to learn is a bit difficult but we can think of an LSTM as parameterizing
 8 the operations of GRU; convex combination in (8.11) is replaced by a
 9 weighted combination using the input and forget gates in (8.14) while the
 10 output gate in (8.13) is identity in a GRU.

11 Just like we can handle multiple layers in an RNN, we can also
 12 have multiple layers in an GRU. Each layer gets its own gates; temporal
 13 propagation is performed using the above equations and only the hidden
 14 state h_t is propagated up to the deeper layers.

15 You will notice that a lot of non-linearities in GRUs/LSTMs are sig-
 16 moids and hyperbolic tangents. This is because these gates are interpreted
 17 as Boolean variables that the model is supposed to learn. There are two
 18 lessons to draw from this. First, if you are modeling some computation
 19 and would like to learn a Boolean variable, it is a good idea to compute a
 20 learnable function of the inputs and use a sigmoid nonlinearity. Second,
 21 vanishing gradients are a problem with LSTMs/GRUs as well, the various
 22 mechanisms (reset/zero in GRUs and input/forget/output in LSTMs) alle-
 23 viate this to an extent but do not eliminate vanishing gradients. Roughly
 24 speaking, we can use LSTMs to model sequences of up to length 50.

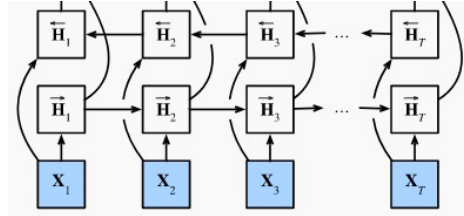
25 8.4 Bidirectional architectures

26 Until now, we have imagined that we would like to predict the future words
 27 in a sequence or design a predictor that uses a statistic of the sequence
 28 to predict the output. Our recurrent models were causal in the temporal
 29 direction, i.e., future elements of the sequence did not play a role in the
 30 outputs and updates of the model at time t . This is indeed how a lot of
 31 computation is performed, e.g., if you want to predict the next location
 32 of a vehicle in a video, you should not build a predictor that uses future
 33 frames because this model cannot be run at test time without access to the
 34 future frames. However, there are also problems in which you have access
 35 to some future observation and estimate the present state. For instance,
 36 you may fill in the following blanks totally differently depending upon the
 37 context of the future words.

38
 39 I am very _____ .
 40 I am very _____ for school.
 41 I am very _____ , I need a big dinner.

1 Bidirectional models help us distinguish between the three situations
 2 and allow predicting context-specific output. Just like we motivated
 3 recurrent models using a Kalman filter and sufficient statistics of the past
 4 sequence, we can also derive an analogy with what is called Kalman
 5 smoothing (predicting the current state given the past observations *and*
 6 the future observations).
 7

8 Building bidirectional models using RNNs is easy. We have two RNNs
 9 running in opposite directions as shown in the following picture.



10
 11 We maintain two sets of weights, one for the forward RNN and the
 12 other for the backward RNN. This gives two hidden states, one in the
 13 forward direction and another in the backward direction

$$h_{t+1}^{\text{forward}} = \sigma(w_h^{\text{forward}} h_t^{\text{forward}} + w_x^{\text{forward}} x_{t+1})$$

$$h_t^{\text{backward}} = \sigma(w_h^{\text{backward}} h_{t+1}^{\text{backward}} + w_x^{\text{backward}} x_t).$$

14 The concatenation of these two hidden states is now the sufficient statistic
 15 of the entire sequence. So the output \hat{y}_t is now a function of both these
 16 hidden states

$$\hat{y}_t = v^{\text{forward}\top} h_t^{\text{forward}} + v^{\text{backward}\top} h_t^{\text{backward}}. \quad (8.15)$$

17 Let us emphasize that these two directions have nothing to do with
 18 backpropagation. There is a backpropagation for the backward directions as
 19 well, which updates $h_{t+1}^{\text{backward}}$ using h_t^{backward} . You should do the following
 20 exercise: imagining that the loss is only computed on the predictions at
 21 time t , i.e., $\ell = \ell(y_t, \hat{y}_t)$ and think of how the backpropagation gradient
 22 flows in a bidirectional RNN.

23 Just like we have bidirectional RNNs, we can also build bidirectional
 24 GRUs and LSTMs.

25 8.5 Attention mechanism

26 The human perception system is quite limited by its sensors, we do not
 27 have eyes at the back of our heads. It is also limited by computation, the
 28 human brain consumes only about 12W of power when it works, about
 29 30% of this power is consumed by the visual system.

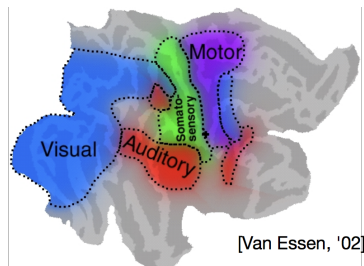


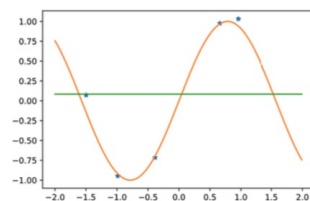
Figure 8.2: This is a picture of the human brain by a neuroscientist named David Van Essen. Around the early 90s it became clear that brains consist of different parts, each specialized to processing different kinds of data. The visual system takes up a bulk (30%) of the real estate.

1 Our perceptual system is very powerful considering the limits of this
 2 computation. We discussed reasons for this in Chapter 1, the ability to
 3 move gives us the ability to specialize the processing on different parts
 4 of the environment instead of passively processing all the incoming data
 5 from the sensors. For instance, when you are driving, you look over your
 6 shoulder only before you merge on the right, you do not really care to
 7 remember where every car in your vicinity is at any given point of time.
 8 Similarly, experiments on race car drivers reveal that even at high speeds
 9 they do not pay attention to all parts of the environment, a driver typically
 10 only cares about two variables, the heading of the car while going into a
 11 turn and the distance to the apex of the turn. When you watch TV, you are
 12 paying attention to only a small part of the TV screen. You can read more
 13 about these experiments at <http://ilab.usc.edu/surprise> and in the work of
 14 many other researchers who study such problems.

15 The human perceptual system is tuned to pay attention to only parts
 16 of the input data that is relevant. Attention in machine learning is an
 17 attempt to model this phenomenon. It turns out that since understanding
 18 which part of a long sequence is relevant to making a prediction at a
 19 particular time instant, attention is well-suited to mitigating the problems
 20 with long-range correlations in sequence data. We will not go very deep
 21 into the architectural intricacies of attention models (you can read the
 22 suggested reading material) but we provide an introduction that makes it
 23 easy to understand the papers.

24 8.5.1 Weighted regression estimate

25 Consider a regression problem where the true function is drawn in orange
 26 and the dataset is shown in blue.



1 If we wanted to predict the targets, then the green line given by

$$\hat{y} = \frac{1}{n} \sum_{i=1}^n y^i.$$

2 is the world's dumbest estimator: it predicts the same output irrespective
3 of the input x . We can do better using the Watson-Nadaraya estimator.
4 This computes the weighted combination

$$\hat{y}(x) = \sum_{i=1}^n k(x, x^i) y^i \quad (8.16)$$

5 where the kernel $k(x, x^i)$ computes some similarity between the input x^i
6 in the dataset and the test input x ; the kernel weighs the target y^i higher if
 x is close to x^i .

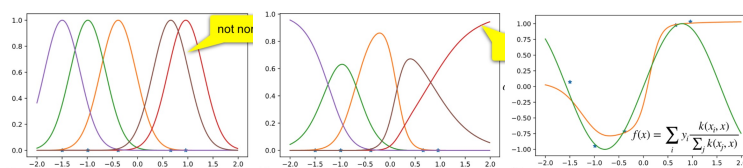


Figure 8.3: The left panel shows the Gaussian kernel $k(\cdot, x^i)$ for different inputs in the dataset. The kernel is not normalized so we cannot match the target values y^i easily using a weighted combination of the kernels. The second panel fixes this by picking a normalized kernel $k(x, x^i) := \frac{k(x, x^i)}{\sum_j k(x, x^j)}$. The estimate of the target $\hat{y}(x)$ using a weighted combination of this normalized kernel is a non-parametric estimator of the targets.

7

8 The Watson-Nadaraya estimator in Figure 8.3 is a simple interpolation
9 mechanism and it is also consistent, i.e., as the amount of data $n \rightarrow \infty$,
10 the regression error goes to zero. There are no “weights” in this model; all
11 the intricacy lies in choosing the kernel to calculate the similarity between
12 two samples.

An attention layer can be thought of as learning a particular kind of weighing function in our regression estimate.

13 8.5.2 Attention layer in deep networks

14 Let us consider a typical kind of attention that is heavily employed in deep
15 learning. It is called the dot-product attention mechanism. This takes in
16 two matrices as input: $k \in \mathbb{R}^{T \times p}$ which is called the “key” and $v \in \mathbb{R}^{T \times p}$
17 which are called “values”. Given a query vector $q \in \mathbb{R}^p$ the attention
18 module outputs

$$\sum_{i=1}^T \sigma(k_i^\top q) v_i \quad (8.17)$$

▲ We can come up with many different kernels that will work for this problem, e.g.,

$$\text{Gaussian} = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

$$\text{Box} = \mathbf{1}_{\{\|x - x'\| \leq c\}}$$

$$\text{Laplace} = \exp(-\lambda \|x - x'\|)$$

Any of these are reasonable kernels to use for the Watson-Nadaraya estimator.

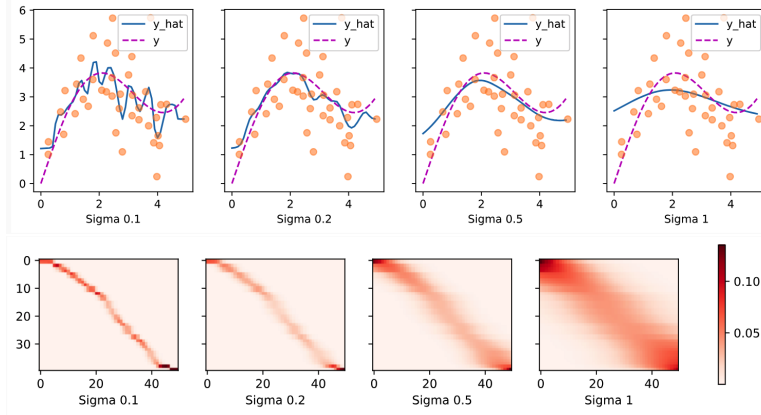


Figure 8.4: Top row: For the true function $y = 2 \sin(x) + x + \epsilon$ where ϵ is zero-mean Gaussian noise (dotted lines), we sample 40 data points from the domain of x (orange points) and fit the targets using the Watson-Nadaraya estimator using a Gaussian kernel for different values of σ . Bottom row: For each of these problems, the heatmap denotes the term $k(x^i, x^j) / (\sum_k k(x^i, x^k))$ where i, j and k range over the 40 data points (say arranged in ascending order from left to right on the real line). As the bandwidth σ increases, the “attention map” denoted by the kernel becomes more diffuse and takes into account farther and farther data points.

- 1 where k_i denotes the i^{th} row of the matrix and likewise for the values.
- 2 Observe that the summation is a weighted combination of all the values
- 3 v_i with weights given by the similarity of the query with each of the keys
- 4 k_i . Just like the Watson-Nadaraya estimator, we would like these weights
- 5 to be normalized, so we choose

$$\sigma(k_i^\top q) = \text{softmax}_i(k_i^\top q) = \frac{e^{k_i^\top q}}{\sum_j e^{k_j^\top q}};$$

- 6 the softmax normalization is performed over the time-axis i . In simple
- 7 words, the expression is a weighted combination of the values where the
- 8 kernel is computed using a simple dot product and normalization of the
- 9 kernel is performed using softmax. If a particular query vector q is similar
- 10 to one of the keys k_i , that value v_i gets up-weighted in the summation.

- 11 The expression for attention is equivalent to the Watson-Nadaraya
- 12 estimator with

$$\begin{aligned} \text{train data } x_i &\equiv k_i \text{ keys} \\ \text{test data } x &\equiv q \text{ query} \\ \text{targets } y_i &\equiv v_i \text{ values} \\ \text{kernel } k(x_i, x) &\equiv e^{k_i^\top q} \text{ exp-dot-product} \\ \sigma(k_i^\top q) &\equiv \frac{e^{k_i^\top q}}{\sum_j e^{k_j^\top q}}. \end{aligned}$$

▲ It is traditional to replace the inner-product by $\frac{k_i^\top q}{\sqrt{p}}$. Keys and queries will be parameterized by the weights of a neural network later. And they can become quite correlated to each other and result in a very large inner product. The denominator \sqrt{d} is chosen with the rationale that if we have two p -dimensional random vectors with standard Gaussian entries, then their inner product has zero mean and variance p ; we can think of this division as an attempt to preserve the magnitude of the similarity kernel in attention.

▲ This is not the only kind of attention. The additive attention operation uses

$$\sigma(k_i, q) \doteq w_v^\top \tanh(w_k^\top k + w_q^\top q);$$

in general, as we said above we can use any kernel for attention.

If the query is one of the keys k_i , this is called the self-attention operation.

1 How can we use this in a deep network? First let us consider a standard
 2 convolutional network with features $h^l \in \mathbb{R}^{m \times c}$ at the l^{th} layer; we have
 3 reshaped the width and height of the feature map into a single dimension
 4 of size m , the number of channels is c . If we set the keys, values and
 5 queries to be learnable quantities

$$\begin{aligned}\mathbb{R}^{m \times c} \ni k &= \text{relu}(w_k^\top h^l) \\ \mathbb{R}^{m \times c} \ni q &= \text{relu}(w_q^\top h^l) \\ \mathbb{R}^{m \times c} \ni v &= \text{relu}(w_v^\top h^l)\end{aligned}\tag{8.18}$$

6 then the output of the attention block would be given by a weighted
 7 summation over the features for each pixel

$$h_j^{l+1} = \sum_{i=1}^m \text{softmax}_i(k_i^\top q_j) v_i.\tag{8.19}$$

8 This is a just a more complex version of the correlation operator. It creates
 9 output features h_j^{l+1} for $j \in \{1, \dots, m\}$ that capture the similarities
 10 between queries and the keys.

11 **Handling set-valued data with attention** Note that the output of (8.19)
 12 if the keys k_i were permuted (and their values were permuted consistently).
 13 The attention operation, or a self-attention operation, is permutation
 14 invariant. This makes it very useful for modeling problems where we are
 15 interested in making predictions using a set of entities, and would like
 16 the output to not depend upon the order of the inputs, e.g., the path of an
 17 autonomous vehicle depends upon the *set* of other vehicles in its vicinity
 18 and their locations, not the order in which they are presented; the number
 19 of chairs in a room does not depend upon the order in which the camera
 20 them as it pans around the room. The attention-operation is ideally suited
 21 to model such problems.

22 [Zaheer et al. \(2017\)](#) proved that a function $f(\{x^1, \dots, x^n\})$ that
 23 operates upon a finite set ($n < \infty$) of inputs $\{x^1, \dots, x^n\}$ with $x^i \in \mathbb{R}$ is
 24 permutation invariant if and only if it can be decomposed in the form

$$f(\{x^1, \dots, x^n\}) = \rho \left(\sum_{i=1}^n \varphi(x_i) \right)$$

25 for some transformations ρ and φ . The function φ can be thought of as
 26 a feature generator that runs on each input of the set x^i ; these features
 27 are aggregated (which makes the sum invariant to permutations of inputs)
 28 before the transformation ρ acts upon it. The basic Watson-Nadaraya
 29 kernel in (8.16) is permutation invariant. The attention operation in (8.17)

▲ Draw a picture of the computation in an attention module

▲ If we want to build permutation invariance through data augmentation, we will need to augment the dataset to have each permutation. For a sequence with n elements, there are $n!$ permutations. It is much better to build permutation invariance via a clever choice to the architecture.

1 and (8.19) is also permutation invariant. This indicates that for any
 2 problem where we need a permutation-invariant representation, we can
 3 use the attention layer fruitfully.

4 **Position Encoding** For many problems we do need to consider the order
 5 of the elements in the set, e.g., for predicting the next word in a sentence,
 6 we should consider the order in which we saw the previous words. A
 7 permutation-invariant model would generate very poor English sentences
 8 (but it would do perfectly fine for languages which do not need a fixed
 9 word order such as Latin, Greek, Polish, or Sanskrit). Therefore, attention
 10 operation would lead to a poor model of sequences for which the order
 11 matters (most sequence are like this).

12 Position encoding modifies the input to retain information about the
 13 position at which the particular input arrived in the sequence. There are
 14 many ways of doing so, for example we could think of simply concatenating
 15 time to the original input to get (t, x_t) . But this makes it difficult to handle
 16 very long sequences, e.g., as t increases, the domain of the inputs to the
 17 model also increases, and if the test data has longer sequences than those
 18 in the training data then we will surely see a distribution shift in the data.
 19 It is therefore popular in sequence modeling to use Fourier features, e.g.,
 20 sinusoids, and use the input

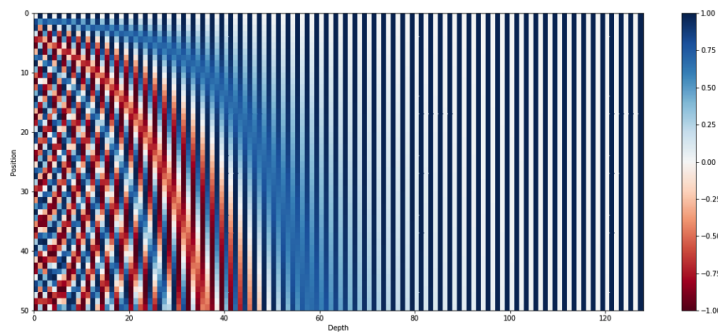
$$\tilde{x}_t = \varphi(t) + x_t, \text{ where}$$

$$\mathbb{R}^d \ni \varphi(t) \doteq [\sin(\omega_1 t), \cos(\omega_1 t), \sin(\omega_2 t), \cos(\omega_2 t), \dots, \sin(\omega_{d/2} t), \cos(\omega_{d/2} t)]^\top, \quad (8.20)$$

21 and the frequencies are chosen by the user. For example, [Vaswani et al.](#)
 22 (2017) used

$$\omega_i = 10^{-8(i-1)/d}$$

23 where $x_t \in \mathbb{R}^d$. The number of frequencies should be chosen after
 24 considering the length of the largest sequence that we wish to model. Each
 25 128-dimensional row of the following figure shows the elements of the
 26 position encoding $\varphi(t)$, each row represents a different value of position
 27 t ; depending upon the dimensionality of the input x_t , the width of this
 28 picture would be truncated to obtain the position encoding.



29
 30 It may seem peculiar that we are summing up the encoding of time
 31 $\varphi(t)$ and the original input x_t . The dimensions of the signal x_t and the

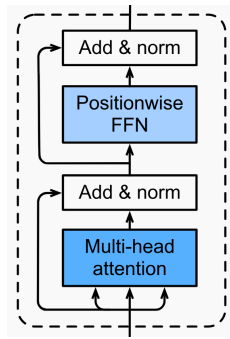


Figure 8.5: One block of the encoder of a Transformer architecture.

1 different dimensions of the position encoding $\varphi(t)$ mean very different
 2 things. We could have also used

$$(\varphi(t), x_t).$$

3 The difference between the two is a tricky implementation detail. If we
 4 sum the position encoding, then on one hand, an attention-layer in (8.18)
 5 that uses these inputs does not have to consider the time part and the input
 6 data distinctly but on the other hand, the magnitude of inputs x_t needs
 7 to be chosen carefully to ensure that the temporal information (which
 8 is magnitude 1 for each dimension and depends upon the length of the
 9 sequence. . . which is perhaps why Vaswani et al. (2017) used frequencies
 10 ω_i that also depend upon d). If we concatenate the position encoding,
 11 then on one hand the attention-layer in (8.18) needs to select its weight
 12 matrices to correctly account for the position encoding. On the other
 13 hand, we need not worry about the relative magnitude of $\varphi(t)$ anymore.
 14 In practice, most people use summation.

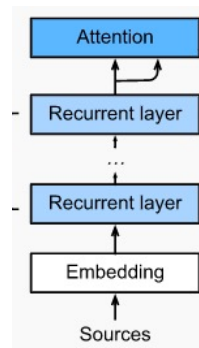
15 **Multi-head attention** Just like we have multiple channels in convolu-
 16 tional networks, we can have multiple channels in attention-based networks.
 17 What we have shown in Section 8.5.2 is a typical “encoder” block from
 18 the Transformer architecture. The multi-head attention layer implements
 19 multiple sets of keys, queries (in the more common self-attention layer, the
 20 queries are the same as the keys) and values and concatenates the output
 21 features in (8.19) for these different sets, followed by a fully-connected
 22 layer to bring back down the dimensionality of the output. A layer nor-
 23 malization layer is used to normalize these concatenated outputs; usually
 24 there is a residual connection where the input to the multi-head attention
 25 layer is added to its output. The position-wise FFN shown in this picture is
 26 simply a fully-connected layer that runs on the features of each time-step
 27 independently.

8.5.3 Attention in recurrent networks

The attention operation is very useful for sequence modeling because it *completely eliminates* the problem of vanishing/exploding gradients. For a sequence of length T , the attention layer computes the same operation as in (8.19). Observe that this expression, rewritten here with the number of features $m \doteq T$ corresponding to the time dimension and the feature size $c \doteq p$

$$h_j^{l+1} = \sum_{i=1}^T \text{softmax}(k_i^\top q_j) v_i$$

has hidden state h_j^{l+1} that depends on the hidden states of the lower layer $h_i^l, i \in \{1, \dots, T\}$. Effectively, the attention layer acts as a temporal shortcut that makes the hidden states of an RNN dependent on both past and future hidden states for the sequence. In a picture, this looks as follows.



The recurrent layers compute features in a causal fashion but the attention layer connects all the time-steps together. If you think of how backpropagation gradient flows down from the output layer via the attention, you will realize that the gradient of the loss computed at step t , say $\ell(y_t, \hat{y}_t)$ flows back to the hidden states h_2 using two paths; the first is the standard BPTT path of the recurrent layers while the second one is a more direct path of the cross-correlation operation in the attention layer. This is a huge benefit because it essentially eliminates problems with gradient vanishing and allows recurrent model very long sequences. Modifications of this attention module can easily handle sequences of a few hundred words.

What is the sufficient statistic that is built by attention? We began our discussion on recurrent models by arguing that we need to build a statistic of the past sequence that can predict the next element of the sequence, i.e., a sufficient statistic. To repeat (8.4)

$$h_{t+1} = \varphi(h_t, x_{t+1}),$$

and recurrent models such as RNNs and GRUs/LSTMs implement such an update. But as we said the updates to the sufficient statistic were made recursive simply for computational purposes. Attention predicts the output

1 directly using all the past inputs

$$\hat{y}_{t+1} = \text{func}(x_1, \dots, x_{t+1})$$

2 in a non-recursive fashion. In other words, attention-based models are
3 no different—conceptually—than the recurrent models that we have seen
4 before. Except that attention-based sequence models are quite peculiar,
5 there is no hidden state in these models. Attention-based layers are also
6 therefore fundamentally more expensive in terms of computation.

7 The upside of extra work is that in other recurrent models, the
8 same statistic h_t has to maintain information about all kinds of future
9 words/tokens etc. For example, if we are building a language model that
10 can create new text, then this same statistic h_t has to learn both the local
11 structure of nearby words and also global structure about the language,
12 context consistency between successive sentences etc. We have seen the
13 issues while doing so for RNNs or GRUs/LSTMs, e.g., vanishing gradients
14 for long sequences. Attention-based networks circumvent this by simply
15 not having the hidden state.

This discussion also suggests that one can forego the recurrent layers altogether in the above picture and simply use attention-based layers for the entire network.

16 **Attention operator can be computed completely in parallel** Note that
17 the number of features p can be quite large, say $p \approx 10^3$ and similarly
18 the length of the sequence that we would like to address with attention-
19 based models can be quite large $T \approx 10^3$. Calculating the self-attention
20 operation in (8.17) requires $\mathcal{O}(pT^2)$ amount of work. There are a lot of
21 important techniques that have been implemented over the years to hide
22 the latency of this calculation and speed-up attention. It important to
23 remember that although attention allows us to handle sequences of very
24 large lengths, the amount of computation that needs to be performed scales
25 quadratically with the sequence length. This is not as bad it seems, because
26 unlike recurrent models where have to predict the outputs sequentially,
27 the outputs of an attention-based network can be computed completely
28 in parallel. In other words, while an RNN does $\mathcal{O}(Tp^2)$ work, it needs
29 $\mathcal{O}(T)$ time to process things sequentially. An attention-based network has
30 to do $\mathcal{O}(pT^2)$ work in $\mathcal{O}(1)$ time.

31 8.6 Some applications of attention-based net- 32 works (transformers)

33 Attention-based models are ideally suited for problems where we need
34 to work with a sequence of inputs. Certainly, they can also be used for
35 problems where there is no temporal structure, e.g., for any problem where
36 we used a multi-layer perception or a convolutional network, we can also
37 use an attention-based layer. In principle,

▲ For many applications, the inputs to each attention layer are masked to ensure that the output, say \hat{y}_t is computed causally by the layer at time t , i.e., it does not depend upon future inputs such as x_{t+1} . This masking can be done by modifying the operation in (8.19) as

$$h_t = \sum_{s=1}^T \text{softmax}_s(k_s^\top q_t + m_{st}) v_s$$

where $m_{st} = -\infty$ if $s \geq t$ and zero otherwise. This is called causal attention. You can use masking to enforce many different kinds of restrictions on how the attention-based computations should be performed. Variants of causal attention need to be implemented very carefully on each layer of an attention-based network to ensure that we are not using information from the future, see a good example at (Sukthanker et al., 2022).

▲ Due to the popularity of the Transformer architecture, first built by Vaswani et al. (2017), self-attention-based networks have become essentially synonymous in the literature with “Transformers”.

1 Multi-layer perceptron \supset Self-attention-based model \supset Con-
2 volutional Network,

3 i.e., any function that can be fitted using a CNN can also be fitted by
4 an appropriate attention-based model (you can think of attention as a
5 particular nonlinear convolutional kernel with width equal to the size of
6 the input), and any function that can be fitted by a self-attention-based
7 model can also be fitted by an appropriate fully-connected network (the
8 computation performed by the attention layer of course can be fitted by
9 the MLP). Self-attention-based networks strike a good balance between
10 versatility (similar architecture can be used for text and language and
11 many other modalities, CNNs and MLPs can also be used like this but
12 it takes some creativity to do so) and ease of training (training large
13 MLPs is very difficult due to numerical issues which we will see in
14 the next Module). All this, coupled with the rise of efficient libraries
15 that implement attention well, e.g., Hugging Face transformers library
16 <https://github.com/huggingface/transformers>, there is a huge number of
17 applications that have seen good results using attention-based architectures.
18 We will next briefly survey some examples.

19 8.6.1 Pretraining on natural language

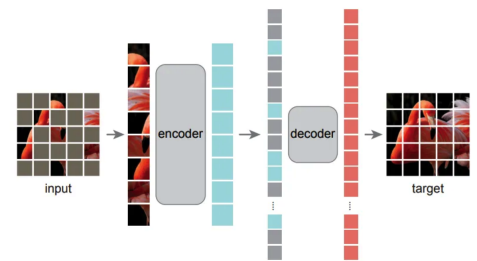
20 Very often, models are trained in two stages. First a model is trained on a
21 large source of data for some simple task, e.g., predicting the next word.
22 This first stage is called “pre-training”. Next, the final layer of this model
23 is reset, or often modified, to allow it to make predictions on some new
24 problem using a new dataset, e.g., answering questions about a piece of
25 text, which requires the model to generate new words in response. This
26 second stage is called “fine-tuning”.

27 We might be able to understand using the bias-variance tradeoff why
28 such a strategy is fruitful; the essential idea is similar to the procedure
29 called doubly robust estimation that we saw in the section on correcting
30 for covariate shift. The pretraining phase restricts the class of models to
31 the ones that can effectively solve the pretraining task, namely predicting
32 the next word. And the fine-tuning stage now needs to select a model
33 from a much smaller set. The variance of the fine-tuning procedure can
34 therefore be small even if we do not have too much data in the second
35 stage. The success of this procedure hinges upon two things: (i) whether
36 the pretraining task is broad enough that the solution of the fine-tuning
37 stage lies within the reduced set of models, and (ii) whether the pretraining
38 task is narrow enough that it meaningfully restricts the set of models to
39 create the reduced set.

40 We will next look at a few ways to pretrain representations on sequential
41 data. Consider a dataset $D = \{(x_t^i)_{t=1}^T\}_{i=1}^n$ with n sentences and T
42 words in each sentence. Let us suppose that our goal is to calculate
43 whether a new sentence $(x_t)_{t=1}^T$ is like the ones in our dataset, i.e., we
44 would like to learn a probability distribution

$$p(x_1, \dots, x_T)$$

▲ For image classification, you can imagine that in the pre-training stage we build a model to predict the RGB pixel intensities of a patch of the input image. The input and output of this architecture will have the same size



The above picture is an example of a masked autoencoder (ignore the details of the architecture for now). After pretraining, we can use the features of one of the layers of this model as inputs to a classifier layer and fine-tune an image classification model.

that gives high likelihood to sentences that look like they belong to our dataset and low likelihood to sentences that are outside. This is a complicated distribution, e.g., say a sentence has $T = 15$ words, and we have about 10,000 unique words, then this probability distribution is supported on a domain of size $15^{10^4} \approx 10^{12,000}$ which is an absurdly large number. Of course, the set of legitimate or natural sentence is much smaller and that is why we can hope to learn something meaningful using this approach.

Bidirectional Encoder Representations from Transformers (BERT)

Instead of modeling the likelihood as a joint probability over all the T words, BERT models it as

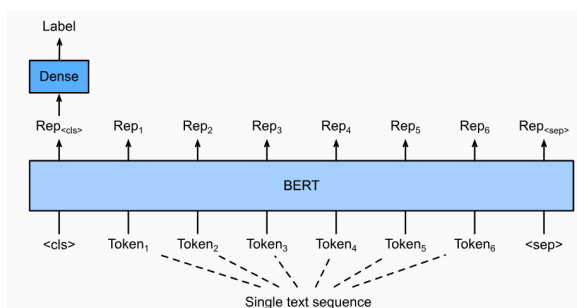
$$\ell^i = - \sum_{t=1}^T \log p_w(x_t^i | x_{-t}^i). \quad (8.21)$$

where $x_{-t}^i \equiv (x_1^i, \dots, x_{t-1}^i, x_{t+1}^i, \dots, x_T^i)$ denotes the sequence of length T with x_t^i replaced by a special masked token to indicate missing information. Each term in the summation is the log-likelihood of predicting the word x_t^i using all the words that came before it in the same sentence and all the words that came after it—hence the name bidirectional. We can also code up the term $p_w(x_t^i | x_{-t}^i)$ using masking to ignore x_t^i . The training objective of BERT is simply the maximum likelihood objective using this model

$$\ell = \frac{1}{n} \sum_{i=1}^n \ell^i.$$

The other details of the architecture, e.g., position encoding (BERT uses learned position encodings), multi-head attention, layer normalization etc. are the same as that of the Transformer architecture.

As we said above, BERT provides a good pretraining objective without any annotations. After this stage, we can use its outputs for a variety of fine-tuning tasks. For example, in the figure below, the sentence is going to be classified as grammatically correct or incorrect using annotated data.



The outputs of the individual words are not being used here, only the output corresponding to the first token (which is a special “word” that signals the beginning of a sentence) is being used to classify. In other

▲ The original BERT paper also used a loss where the model takes as input two sentences in different orders and fits a binary classifier to detect which sentence came first in the text. Let us ignore this for clarity’s sake.

▲ Once we become comfortable with writing these kinds of likelihoods for pretraining objectives, there are many alternatives to think of, e.g.,

$$- \sum_{t=1}^T \mathbb{E}_{r_t} [\log p_w(x_t, \dots, x_{t+T} | r \odot x)]$$

where the mask denoted by $r \equiv (r_t)_{t=1}^T$ for $r_t \in \{0, 1\}$ hides a random sub-sequence of consecutive words in the sentence x using a single special token. This loss, in addition to the BERT objective, was used to train a famous model named T5 by Google.

1 problems, e.g., prediction of the part of a sentence, we would use the other
2 outputs.

3 **word2vec with a contiguous bag of words** We can also build a rather
4 simplistic model of the data by writing

$$\ell^i = - \sum_{t=1}^T \log p_w(x_t^i | \underbrace{x_{t-m}^i, \dots, x_{t-1}^i, x_{t+1}^i, \dots, x_{t+m}^i}_{2m \text{ neighboring words}}). \quad (8.22)$$

5 where instead of using the entire sentence to predict x_t^i , we only use the
6 m words before and after. Such a network would not be able to model
7 long-range dependencies between words but it would be good at using
8 local context to fill in the blanks, i.e., predict x_t^i . This model called
9 word2vec was one of the first examples of highly versatile and effective
10 word embeddings (there are many uses for this, e.g., in content retrieval
11 and text search). A smaller window of $2m$ words also reduces the amount
12 of computation that we need to do for predicting the embedding of each
13 word in the sentence.

14 **Generative Pre-trained Transformer (GPT)** The BERT objective is
15 not causal, i.e., the prediction of a word depends upon both which words
16 came before it and which ones came after. Such a pretraining objective
17 cannot be used if our desired task is to generate new text. The GPT
18 objective is

$$\ell^i = - \sum_{t=1}^T \log p_w(x_t^i | x_{<t}^i). \quad (8.23)$$

19 where $x_{<t}^i \equiv (x_1^i, \dots, x_{t-1}^i)$ denotes the sequence of length $t-1$ that ends
20 just before x_t^i . Each of the terms in the summation is the log-likelihood of
21 predicting the next word x_t^i using all the words that came before it; as we
22 saw before we can easily use a mask to force GPT to ignore $x_{\geq t}^i$. GPT
23 uses sines and cosines as position encodings, multi-head attention, layer
24 normalization etc.

25 The interesting aspect of writing the probabilistic model like (8.23) is
26 that at inference time, we can draw samples

$$\hat{x}_t \sim p_w(\cdot | \hat{x}_{<t}) \quad \forall t = T_0 + 1, \dots, T,$$

27 starting from T_0 initial words of a sentence (x_1, \dots, x_{T_0}) ; this starting
28 sequence is called a “prompt”. The predicted sequence at each time-step
29 as the model runs forward is

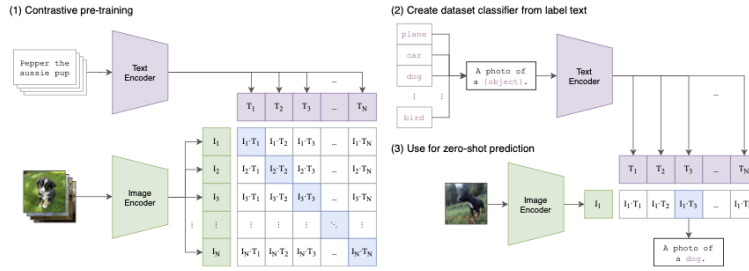
$$\hat{x}_{<t} = (x_1, \dots, x_{T_0}, \hat{x}_{T_0+1}, \dots, \hat{x}_{t-1}).$$

30 Later variants of GPT are very similar to this objective, except that GPT-3
31 was trained on about 500 billion tokens (roughly the same thing as words).

Reconstruction of the original input is a good pretraining task In general, the pretraining task can be any task that restricts the set of hypothesis sufficiently before the fine-tuning phase begins. It has been noticed that for many problems, completing masked versions of the input (e.g., predicting missing words in the text, in-painting masked patches of an image) or building invariance to transformed versions of the input (e.g., ensuring that the features of the original input image are close to those of the same image rotated, translated, masked, blurred etc.) are very good pretraining tasks. Such tasks, specifically reconstruction of masked versions of the input, force the model to learn features that are a lossless encoding of the original inputs—in our words, the features are a statistic of the original input that are sufficient for reconstruction.

8.6.2 Handling multi-modal inputs

Since the attention layer can be used for different input modalities, we have a neat way of combining them while building our model. For example, the CLIP model shown below (from (Radford et al., 2021)) computes a joint embedding from images and text.



Given a dataset of images and their captions (e.g., those derived from Instagram, or created manually from visual recognition datasets such as Imagenet to have captions such as “a photo of a dog”), CLIP pretrains the model in a simplistic fashion. At each mini-batch, the text embedding of the caption is forced to be similar to the image embedding of the corresponding image, while being far away from the image embedding of all the other images in the mini-batch. Given n input pairs (x^i, c^i) where x^i is an image and c^i is a caption, we can use an objective

$$\begin{aligned} \ell_x^i &= -\log \frac{e^{-\lambda/2 \|\varphi(x^i) - \varphi(c^i)\|^2}}{\sum_j e^{-\lambda/2 \|\varphi(x^i) - \varphi(c^j)\|^2}} \\ \ell_c^i &= -\log \frac{e^{-\lambda/2 \|\varphi(c^i) - \varphi(x^i)\|^2}}{\sum_j e^{-\lambda/2 \|\varphi(c^i) - \varphi(x^j)\|^2}} \\ \ell^i &= \frac{\ell_x^i + \ell_c^i}{2}. \end{aligned} \quad (8.24)$$

Effectively, in ℓ_x^i we are setting up a Gaussian mixture model where the centers of the Gaussians are at $\varphi(x^i)$ and we are maximizing the likelihood of the correct caption embedding $\varphi(c^i)$ being closer to this Gaussian than

⚠ These techniques, when combined with the fine-tuning phase for a particular task, e.g., image classification, text translation, generative modeling of images/text/sounds are called self-supervised learning. Most people also pretrain models on very large amounts of data, e.g., millions of images, or terabytes of text, to pretrain “foundation models” from which we can fine-tune to a very large set of tasks.

1 the others; a similar loss is used to get ℓ_c^i where the Gaussians are now
2 centered at $\varphi(c^i)$ and we are calculating the likelihood of the image
3 embeddings being close to the correct Gaussian.

4 This is a very simple objective but CLIP is very effective at a wide
5 variety of problems ranging from supervised learning (i.e., a CLIP model
6 trained using this loss can be used for image classification as shown in
7 the figure (b) and (c), you can also use a linear classifier using the CLIP
8 features).

Chapter 9

Background on Optimization, Gradient Descent

We have covered the cliff-notes of the practice of deep learning in the previous eight chapters. It is by no means a complete overview. The practice of deep learning is an enticing, mysterious, and sometimes frustrating enterprise. The more time you spend playing with code, the more you will learn about deep learning. New ideas are routinely discovered using very simple experiments that each of you is capable of running now.

As we discussed, there three main concepts in machine learning. First, the class of functions $f(x; w)$ that you use to make predictions, this is called the hypothesis class or the architecture. Second, the algorithm you use to find the best model in this class of functions that fits your data; this uses tools from optimization theory. Third is the generalization performance of your classifier. Machine Learning is about picking a good hypothesis class, finding the best model within this class and making sure that the model generalizes.

The above process is relatively well-understood for simpler models such as SVMs but the story is quite murky for deep networks. Often in practice, it is never clear which architecture you should pick for your problem (many of you have asked this question in the office hours for instance). Training a deep network involves a number of bells and whistles (some of which like Batch-Normalization and Dropout that we have seen) and if at the end of this exercise we get a high validation error, it is unclear how one should change the parts of the process to improve performance. Disentangling this vicious cycle is what “understanding deep learning” is all about.

Goal Module 2 will develop an understanding of optimization and generalization for more generic machine learning models first. It will end with an insight into understanding their interplay for deep networks.

1 Module 2 has a different flavor, it is more theoretical. Our goal is to grasp
 2 the general concepts behind these theoretical results and understand the
 3 training process of deep networks better. This will also help us train deep
 4 networks much better in practice.

5 9.1 Convexity

6 Consider a function $\ell : \mathbb{R}^p \rightarrow \mathbb{R}$ that is convex, i.e., for any w, w' that
 7 lie in the domain (which is assumed to be a convex set) of f and any
 8 $\lambda \in [0, 1]$ we have

$$\ell(\lambda w + (1 - \lambda)w') \leq \lambda \ell(w) + (1 - \lambda)\ell(w'). \quad (9.1)$$

9 A function $\ell(w)$ is concave if $-\ell(w)$ is convex. If the function f is
 10 continuous, it is enough to check this definition for a particular value of
 11 λ , say $\lambda = 1/2$ if you need to prove that a function is convex. Some
 12 examples of convex functions are

- 13 • powers w^α for $w > 0$ and $\alpha \geq 1$ or $\alpha \leq 0$ for $w \in \mathbb{R}$,
- 14 • powers of absolute values $|w|^\alpha$ for $w \in \mathbb{R}$ and $\alpha \geq 1$,
- 15 • exponential $\exp(w)$, negative logarithm $-\log(w)$ for $w \in \mathbb{R}$,
- 16 • affine functions $Aw + b$,
- 17 • quadratics $w^\top Aw + b^\top w + c$,
- 18 • norms $\|w\|_p = (\sum_{i=1}^p |w_i|^p)^{1/p}$ for $p \geq 1$, or $\|w\|_\infty = \max_k |w_k|$,
- 19 • log-sum-exp $f(w)_i = \log \sum_i \exp(w_i)$ for $w \in \mathbb{R}^p$.

20 **Strictly convex functions** Strictly convex functions have the property
 21 that for all $w \neq w'$ in the domain (which is assumed to be convex) and
 22 $\lambda \in (0, 1)$

$$\ell(\lambda w + (1 - \lambda)w') < \lambda \ell(w) + (1 - \lambda)\ell(w').$$

23 **First-order condition for convexity** If ℓ is differentiable, the definition
 24 of convexity in (9.1) is equivalent to the following first-order condition. A
 25 differentiable function ℓ with convex domain is convex iff

$$\ell(w') \geq \ell(w) + \langle \nabla \ell(w), w' - w \rangle. \quad (9.2)$$

26 for all w, w' in the domain. Note that the first-order condition is equivalent
 27 to the definition of convexity in (9.1) for differentiable functions. The
 28 proof is long but easy; you can see https://www.princeton.edu/aaa/Public/Teaching/ORF523/S16/ORF523_S16_Lec7_gh.pdf for the proof. For
 29 strictly convex functions the inequality is strict
 30

$$\ell(w') > \ell(w) + \langle \nabla \ell(w), w' - w \rangle.$$

1 **Monotonicity of the gradient for convex functions** The first-order
 2 condition for convexity gives a useful, and equivalent, characterization of
 3 the gradient. Write (9.2) for w, w' in two opposite directions

$$\begin{aligned}\ell(w) &\geq \ell(w') + \langle \nabla \ell(w'), w - w' \rangle \\ \ell(w') &\geq \ell(w) + \langle \nabla \ell(w), w' - w \rangle\end{aligned}$$

4 and add them to get

$$\langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle \geq 0. \quad (9.3)$$

5 This is called the “monotonicity of the gradient” condition for convexity.
 6 In words, it says that the change in the gradient $\nabla \ell(w) - \nabla \ell(w')$ and
 7 the change in the weights $w - w'$ are aligned, i.e., their inner product is
 8 non-negative.

▲ Try to prove that monotonicity of the gradient is equivalent to convexity.

9 **Second-order condition for convexity** If ℓ is twice-differentiable and
 10 the domain is convex, then ℓ is convex iff

$$\nabla^2 \ell(w) \succeq 0, \quad (9.4)$$

11 for all w in the domain. The symbol \succeq denotes positive semi-definiteness
 12 of the Hessian matrix $\nabla^2 \ell(w)$ whose entries are given by

$$(\nabla^2 \ell(w))_{ij} = \frac{\partial^2 \ell(w)}{\partial w_i \partial w_j}.$$

13 For strictly convex functions, the inequality in (9.4) is strict, i.e., the
 14 Hessian is positive definite.

15 As an example using the second-order condition of convexity to
 16 show that a function is convex, note that the least squares objective
 17 $\ell(w) = \frac{1}{2} \|y - Xw\|_2^2$ is convex because

$$\nabla^2 \ell(w) = X^\top X \succeq 0$$

18 which is positive semi-definite for any X .

19 **Strongly convex functions** A function is strongly convex if there exists
 20 an $m > 0$ such that

$$\ell(w) - \frac{m}{2} \|w\|_2^2 \text{ is convex.} \quad (9.5)$$

21 It is easy to see that strong convexity implies strict convexity. Since
 22 the function $\ell(w) - m/2 \|w\|_2^2$ is convex, it satisfies:

$$\begin{aligned}\ell(\lambda w + (1 - \lambda)w') - \frac{m}{2} \|\lambda w + (1 - \lambda)w'\|_2^2 \\ \leq \lambda \left(\ell(w) - \frac{m}{2} \|w\|_2^2 \right) + (1 - \lambda) \left(\ell(w') - \frac{m}{2} \|w'\|_2^2 \right).\end{aligned} \quad (9.6)$$

1 But

$$\frac{\lambda m}{2} \|w\|^2 + \frac{(1-\lambda)m}{2} \|w'\|^2 - \frac{m}{2} \|\lambda w + (1-\lambda)w'\|^2 > 0$$

2 for $\lambda \in (0, 1)$ for all $w \neq w'$ because $\|w\|^2$ is strictly convex. This shows
3 that if we have a strongly convex function ℓ it also satisfies

$$\ell(\lambda w + (1-\lambda)w') < \lambda \ell(w) + (1-\lambda)\ell(w').$$

In other words, we have

$$\text{strong convexity} \Rightarrow \text{strict convexity} \Rightarrow \text{convexity.}$$

4 We will see that strongly convex functions are easier to optimize
5 for our algorithms. It will also always be much easier to prove a result,
6 e.g., the number of iterations that we should run gradient descent for, for
7 strongly convex functions. In your homework, you will show that the
8 second-order condition for strongly convex functions reads as

$$\nabla^2 \ell(w) \succeq mI_{p \times p}.$$

9 We will use the following first-order condition for strongly convex
10 functions often. A function is m -strongly convex if and only if

$$\ell(w') \geq \ell(w) + \langle \nabla \ell(w), w' - w \rangle + \frac{m}{2} \|w' - w\|^2 \quad (9.7)$$

11 for any w, w' in the domain. This is easy to show by observing that the
12 function of $v = w' - w$

$$g(v) \equiv \ell(v + w) - \ell(w) - \langle \nabla \ell(w), v \rangle,$$

13 where w is fixed, is also m -strongly convex if ℓ is and therefore $g(v) -$
14 $m/2\|v\|^2$ is convex.

15 9.2 Introduction to Gradient Descent

16 In this chapter, we will write $\ell(w)$ to denote the training objective, i.e.,
17 if we have a classifier $f(x; w)$ and a dataset $D = \{(x^i, y^i)\}_{i=1, \dots, n}$ of n
18 samples we will denote

$$\ell(w) := \frac{1}{n} \sum_{i=1}^n \ell(w; x^i, y^i).$$

19 The objective ℓ will always be a function of the entire dataset but we
20 will keep the dependence implicit. Note that the number of samples n is
21 usually quite large in deep learning, so the summation above has a large
22 number of terms on the right-hand side.

▲ The Polyak-Lojasiewicz (PL) inequality says that

$$\frac{1}{2m} \|\nabla \ell(w)\|^2 \geq \ell(w) - \ell(w^*). \quad (9.8)$$

Functions that satisfy the PL inequality need not be convex (it simply says that the magnitude of the gradient at a point w should be small if $w \approx w^*$) but it has been studied that such functions are also easy to optimize using first-order optimization methods. It has been empirically found that the PL inequality holds for many deep networks. Prove that a strongly convex function satisfies PL inequality, you can first prove (9.7) and minimize both sides of this inequality over w' .

1 Gradient descent is a simple algorithm to minimize $\ell(w)$. Before we
2 study its properties, it will help to refresh the following few facts.

3 9.2.1 Conditions for optimality

4 **Local and global minima** A point w is a local minimum of the function
5 $\ell(w)$ for all w' in a neighborhood of w we have $\ell(w) \leq \ell(w')$. The
6 point is a global minimum of the function ℓ if this condition is true for all
7 w' in the domain, not just the ones in the neighborhood.

8 **Local minima are global minima for convex functions** This is easy to
9 see using an argument by contradiction. If w is a local minimum that is
10 not the global minimum, there exists a point w' in the domain such that
11 $\ell(w') < \ell(w)$. The function is convex, so pick a point $v = \lambda w' + (1 - \lambda)w$
12 and see that

$$\ell(v) - \ell(w) \leq \lambda(\ell(w') - \ell(w));$$

13 using the definition of convexity. Since w is only a local minimum, we
14 can pick λ to be small enough that the left hand side is non-negative. This
15 shows that $\ell(w') \geq \ell(w)$ but this means that w is a global minimum and
16 we have a contradiction.

17 **Global minimum is unique for strictly convex functions** If a function
18 is strictly convex on a convex domain the optimal solution (if it exists)
19 must be unique. Indeed, if there were two solutions w, w' that were both
20 minimizers we would have

$$\ell(w) = \ell(w') \leq \ell(w'') \quad \forall w'' \tag{9.9}$$

21 We can now apply the definition of convexity to the point $v = (w + w')/2$
22 to get

$$\ell(v) < \frac{1}{2}\ell(w) + \frac{1}{2}\ell(w') = \ell(w).$$

23 which contradicts (9.9). The least-squares objective is strictly convex, so
24 the solution is unique global minimizer of the objective.

25 **First-order optimality condition** If w is a local minimum of a continu-
26 ously differentiable function ℓ , then it satisfies

$$\nabla \ell(w) = 0. \tag{9.10}$$

27 If further ℓ is convex, then $\nabla \ell(w) = 0$ is a sufficient condition for global
28 optimality from the above discussion.

29 9.2.2 Different types of convergence

30 Let us assume that we have a continuously differentiable convex function
31 ℓ and let

$$w^* = \underset{w}{\operatorname{argmin}} \ell(w)$$

1 be the global minimizer of this function.

2 We would like to develop an iterative scheme that takes in the initial-
3 ization of the weights w^0 and updates them to obtain a sequence

$$w^{(0)}, w^{(1)}, \dots, w^{(t)}, \dots$$

4 Along this sequence we are interested in understanding the

5 1. convergence of the function value $\ell(w^{(t)})$ to the minimal value
6 $\ell(w^*)$, and

7 2. convergence of the iterates $\|w^{(t)} - w^*\|$.

8 **Descent direction** We are going to perform a sequence of updates given
9 by

$$w^{(t+1)} = w^{(t)} + \eta d^{(t)} \quad (9.11)$$

10 where $d^{(t)}$ is called the descent direction and the scalar parameter $\eta > 0$
11 is called the step-size and determines how far we travel using this descent
12 direction. Any direction such that

$$\langle \nabla \ell(w^{(t)}), d^{(t)} \rangle < 0$$

13 is a good descent direction because this leads to a reduction in the value of
14 the function $\ell(w^{(t+1)})$ after the weight update locally. There are numerous
15 ways to pick a good descent direction. Among the simplest ones is gradient
16 descent which descends along the direction of the negative gradient and
17 thereby performs the following set of updates

$$w^{(t+1)} = w^{(t)} - \eta \nabla \ell(w^{(t)}) \quad (9.12)$$

18 given an initial value $w^{(0)}$. The step-size (also called the learning rate) is
19 chosen by the user. The step-size need not always be fixed, for instance
20 you can chose it to be a function of the number of weight updates t in the
21 homework. A good step-size is one that does not overshoot the minimum
22 w^* . For instance, after having chosen a particular descent direction $d^{(t)}$
23 we can compute the best step-size to use at time t by solving

$$\eta^{(t)} = \operatorname{argmin}_{\eta \geq 0} \ell(w^{(t)} + \eta d^{(t)}).$$

24 This is known as line-search in the optimization literature. You may have
25 seen Newton's method

$$w^{(t+1)} = w^{(t)} - \left(\nabla^2 \ell(w^{(t)}) \right)^{-1} \nabla \ell(w^{(t)}). \quad (9.13)$$

26 which does not have a user-tuned step-size and further modifies the descent
27 direction to be the product of the inverse Hessian with the gradient.

▲ Draw a picture of overshooting using a large step-size.

🔗 Can you think of an algorithm for minimizing a function that does not use the gradient of the function to compute the descent direction?

9.3 Convergence rate for gradient descent

We will next understand how quickly gradient descent converges to the global minimum. There are two concrete goals of this analysis

1. to be able to pick the step-size to avoid overshooting without doing line-search, and
2. characterize how many iterations of gradient descent to run until we are guaranteed to be within some distance of the global minimum.

9.3.1 Some assumptions

Before we begin, we will make a few simplifying assumptions on the function $\ell(w)$. These are quite typical in optimization and ensure that we are not dealing with functions that are arbitrarily difficult to optimize.

1. **Lipschitz continuity/bounded gradients** We will assume that ℓ is uniformly Lipschitz continuous over the entire domain, i.e.,

$$|\ell(w) - \ell(w')| \leq B\|w - w'\|_2. \quad (9.14)$$

for some $B > 0$. You might also see this condition written as

$$\|\nabla\ell(w)\| \leq B$$

for differentiable functions; show that the second one is an implication of the first.

2. **Smoothness** We will always consider “smooth” functions with gradients that are L -Lipschitz, i.e.,

$$\|\nabla\ell(w) - \nabla\ell(w')\|_2 \leq L\|w - w'\|_2. \quad (9.15)$$

If ℓ is twice-differentiable, this is equivalent to assuming

$$\nabla^2\ell(w) \preceq L I_{p \times p}. \quad (9.16)$$

From the Cauchy-Schwarz inequality which states that

$$\langle u, v \rangle \leq \|u\| \|v\|$$

for two vectors u, v , we have the following implication of smoothness:

$$\langle \nabla\ell(w) - \nabla\ell(w'), w - w' \rangle \leq L\|w - w'\|^2. \quad (9.17)$$

9.3.2 GD for convex functions

We begin with the so-called Descent Lemma.

Lemma 9.1 (Descent Lemma). For an L -smooth function, we have

$$\ell(w') \leq \ell(w) + \langle \nabla\ell(w), w' - w \rangle + \frac{L}{2}\|w' - w\|^2. \quad (9.18)$$

1 for any two w, w' in the domain.

2 **Proof.** First, you should compare this with the first-order characterization
3 of convexity

$$\ell(w') \geq \ell(w) + \langle \nabla \ell(w), w' - w \rangle.$$

4 The two conditions can be used to sandwich the value of $\ell(w^{(t+1)})$ given
5 the value of $\ell(w^{(t)})$ in gradient descent with room for a quadratic term
6 $\frac{L}{2} \|w' - w\|^2$. This marshals the intuition as to what L -smooth really
7 means; a large value of L means that the function ℓ has a large curvature.
8 Let $v = w + \lambda(w' - w)$ and use Taylor's theorem to see that

$$\ell(w') = \ell(w) + \int_0^1 \langle \nabla \ell(v), w' - w \rangle d\lambda \quad (9.19)$$

9 Subtract $\langle \nabla \ell(w), w' - w \rangle$ from both sides to get

$$\ell(w') - \ell(w) - \langle \nabla \ell(w), w' - w \rangle = \int_0^1 \langle \nabla \ell(v) - \nabla \ell(w), w' - w \rangle d\lambda.$$

10 Observe that

$$\begin{aligned} |\ell(w') - \ell(w) - \langle \nabla \ell(w), w' - w \rangle| &= \left| \int_0^1 \langle \nabla \ell(v) - \nabla \ell(w), w' - w \rangle d\lambda \right| \\ &\leq \int_0^1 |\langle \nabla \ell(v) - \nabla \ell(w), w' - w \rangle| d\lambda \\ &\leq \int_0^1 \|\nabla \ell(v) - \nabla \ell(w)\| \|w' - w\| d\lambda \\ &\leq L \int_0^1 \lambda \|w' - w\|^2 d\lambda \\ &= \frac{L}{2} \|w' - w\|^2. \end{aligned}$$

11 This completes the proof after removing the absolute value on the left-hand
12 side. \square

13 We can use the Descent Lemma twice on two points to w, w' to
14 get (9.17). Another direct consequence of the Descent Lemma is the
15 following corollary that relates the value $\ell(w)$ at any point w in the domain
16 to that of the global minimum.

17 **Corollary 9.2.** For L -smooth convex function ℓ , if w^* is the global
18 minimizer, then

$$\frac{1}{2L} \|\nabla \ell(w)\|^2 \leq \ell(w) - \ell(w^*) \leq \frac{L}{2} \|w - w^*\|^2. \quad (9.20)$$

19 **Proof.** Since $\nabla \ell(w^*) = 0$, the right-hand side follows directly from the
20 Descent Lemma. To get the left-hand side, let us optimize the upper bound

1 in the Descent Lemma using $w' = w + \lambda v$ with $\|v\| = 1$ as follows

$$\begin{aligned}
 \ell(w^*) &= \inf_{w'} \ell(w') \\
 &\leq \inf_{w'} \left\{ \ell(w) + \langle \nabla \ell(w), w' - w \rangle + \frac{L}{2} \|w' - w\|^2 \right\} \\
 &= \inf_{\|v\|=1} \inf_{\lambda} \left\{ \ell(w) + \lambda \langle \nabla \ell(w), v \rangle + \frac{L}{2} \lambda^2 \right\} \\
 &= \inf_{\|v\|=1} \left\{ \ell(w) - \frac{1}{2L} (\langle \nabla \ell(w), v \rangle)^2 \right\} \\
 &= \ell(w) - \frac{1}{2L} \|\nabla \ell(w)\|^2.
 \end{aligned}$$

2

□

3 In other words, the gap between the function values $\ell(w) - \ell(w^*)$ is upper-
 4 bounded by the gap to the minimizer $\frac{L}{2} \|w - w^*\|^2$ and lower-bounded by
 5 the norm of the gradient $\frac{1}{2L} \|\nabla \ell(w)\|^2$.

6 **Co-coercivity of the gradient** The gradient being L -Lipschitz is equiv-
 7 alent to co-coercivity of the gradient with parameter $1/L$

$$\langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle \geq \frac{1}{L} \|\nabla \ell(w) - \nabla \ell(w')\|^2. \quad (9.21)$$

8 We can see that co-coercivity implies Lipschitz continuity of the
 9 gradients $\nabla \ell(w)$ using (9.17) and (9.21). The reverse is also true and you
 10 will show both of these sides in the homework. A sketch of the reverse
 11 goes as follows.

12 Note that Lipschitz-continuity of the gradient implies the Descent
 13 Lemma (Lemma 9.1). Now define two functions

$$\begin{aligned}
 g(u) &= \ell(u) - \langle \nabla \ell(w), u \rangle \\
 h(u) &= \ell(u) - \langle \nabla \ell(w'), u \rangle.
 \end{aligned}$$

14 Both of these have L -Lipschitz gradients. Note that $u = w$ minimizes
 15 $g(u)$ (the minimum is zero). Observe that

$$\begin{aligned}
 \ell(w') - \ell(w) - \langle \nabla \ell(w), w' - w \rangle &= g(w') - g(w) \\
 &\geq \frac{1}{2L} \|\nabla g(w')\|^2 \quad \text{from (9.20)} \\
 &= \frac{1}{2L} \|\nabla \ell(w') - \nabla \ell(w)\|^2.
 \end{aligned}$$

16 Apply the same again to h to get

$$\ell(w) - \ell(w') - \langle \nabla \ell(w'), w - w' \rangle \geq \frac{1}{2L} \|\nabla \ell(w') - \nabla \ell(w)\|^2$$

17 and add the two inequalities.

18 We can now get our first result on how gradient descent makes
 19 monotonic progress towards the solution.

▲ The condition in (9.21) is called co-coercivity because there is a related condition called coercivity for m -strongly convex functions

$$\langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle \geq m \|w - w'\|^2,$$

for all w, w' . Try to prove it. Note that this condition boils down to simple monotonicity of the gradient for $m = 0$ (which is just a convex function). This is why it is also called strong monotonicity of $\nabla \ell$.

1 **Lemma 9.3 (Monotonic progress for gradient descent).** For gradient
2 descent $w^{(t+1)} = w^{(t)} - \eta \nabla \ell(w^{(t)})$, if we pick the step-size

$$\eta \leq \frac{1}{L} \quad (9.22)$$

3 we have

$$\ell(w^{(t+1)}) \leq \ell(w^{(t)}) - \frac{\eta}{2} \left\| \nabla \ell(w^{(t)}) \right\|^2 \quad \forall t. \quad (9.23)$$

4 Further,

$$\ell(w^{(t+1)}) - \ell(w^*) \leq \frac{1}{2\eta} \left(\left\| w^{(t)} - w^* \right\|^2 - \left\| w^{(t+1)} - w^* \right\|^2 \right) \quad (9.24)$$

5 which implies

$$\left\| w^{(t+1)} - w^* \right\|^2 \leq \left\| w^{(t)} - w^* \right\|^2. \quad (9.25)$$

6 **Proof.** Substitute $\eta \leq 1/L$ in the Descent Lemma and simplify to
7 get (9.23). The second result is obtained by

$$\begin{aligned} 0 \leq \ell(w^{(t+1)}) - \ell(w^*) &\leq \ell(w^{(t)}) - \ell(w^*) - \frac{\eta}{2} \left\| \nabla \ell(w^{(t)}) \right\|^2 \\ &\leq \left\langle \nabla \ell(w^{(t)}), w^{(t)} - w^* \right\rangle - \frac{\eta}{2} \left\| \nabla \ell(w^{(t)}) \right\|^2 \\ &= \frac{1}{2\eta} \left(\left\| w^{(t)} - w^* \right\|^2 - \left\| w^{(t)} - w^* - \eta \nabla \ell(w^{(t)}) \right\|^2 \right) \\ &= \frac{1}{2\eta} \left(\left\| w^{(t)} - w^* \right\|^2 - \left\| w^{(t+1)} - w^* \right\|^2 \right). \end{aligned}$$

8 Observe that since the left-hand side is positive, the claim in (9.25) is
9 true. \square

10 We have therefore shown that if the step-size is not too large (the
11 smoothness parameter of the function determines how large the step-size
12 can be) then gradient descent always improves the value of the function
13 with each iteration (9.23). It also improves the distance of the weights to
14 the global minimum at each iteration (9.25).

15 **Lemma 9.4 (Convergence rate for gradient descent, convex function).**
16 For gradient descent $w^{(t+1)} = w^{(t)} - \eta \nabla \ell(w^{(t)})$ with step-size $\eta < 1/L$,
17 we have

$$\ell(w^{(t+1)}) - \ell(w^*) \leq \frac{1}{2t\eta} \left\| w^{(0)} - w^* \right\|^2. \quad (9.26)$$

1 **Proof.** We sum up the expression in (9.24) for all times t to get

$$\begin{aligned} \sum_{s=1}^t \ell(w^{(s)}) - \ell(w^*) &\leq \frac{1}{2\eta} \sum_{s=1}^t \left(\|w^{(s-1)} - w^*\|^2 - \|w^{(s)} - w^*\|^2 \right) \\ &= \frac{1}{2\eta} \left(\|w^{(0)} - w^*\|^2 - \|w^{(t)} - w^*\|^2 \right) \\ &\leq \frac{1}{2\eta} \|w^{(0)} - w^*\|^2. \end{aligned}$$

2 We know from (9.23) that $\ell(w^{(t)})$ is non-increasing, so we can write

$$\ell(w^{(t)}) - \ell(w^*) \leq \frac{1}{t} \sum_{s=1}^t \left(\ell(w^{(s)}) - \ell(w^*) \right) \leq \frac{1}{2t\eta} \|w^{(0)} - w^*\|^2.$$

3

□

If we want to find weights with

$$\ell(w^{(t)}) - \ell(w^*) \leq \epsilon$$

for a convex function, we need to run gradient descent for at least

$$t = \mathcal{O}(1/\epsilon)$$

iterations. This is an important result to remember.

4 9.3.3 Gradient descent for strongly convex functions

5 Things are much better if the function we are minimizing is strongly
6 convex. First we have the following lemma for strongly-convex functions
7 which involves a rewriting co-coercivity condition for strongly convex
8 functions.

9 **Lemma 9.5 (Co-coercivity for strongly convex function).** If $\ell(w)$ is
10 m -strongly convex, and L -smooth, then

$$\langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle \geq \frac{mL}{m+L} \|w - w'\|^2 + \frac{1}{m+L} \|\nabla \ell(w) - \nabla \ell(w')\|^2. \quad (9.27)$$

11 **Proof.** If the conditions of the lemma hold, then the function $g(w) =$
12 $\ell(w) - \frac{m}{2} \|w\|^2$ is convex and $(L - m)$ -smooth. The convexity of $g(w)$
13 is immediate to see from the definition of strong convexity of $\ell(w)$. Use
14 the monotonicity of the gradient of $g(w)$ to get

$$\begin{aligned} 0 &\leq \langle \nabla g(w) - \nabla g(w'), w - w' \rangle \\ &= \langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle - m \|w - w'\|^2 \\ &\leq (L - m) \|w - w'\|^2. \end{aligned}$$

1 We can now rewrite the co-coercivity condition for ∇g and simplify to
 2 get (9.27). \square

3 **Lemma 9.6 (Convergence rate of gradient descent for strongly convex**
 4 **functions).** For strongly convex functions we have pick a step-size

$$0 < \eta < \frac{2}{m + L}$$

5 to get

$$\|w^{(t+1)} - w^*\|^2 \leq \left(1 - \eta \frac{2mL}{m + L}\right) \|w^{(t)} - w^*\|^2. \quad (9.28)$$

6 which gives

$$\|w^{(t)} - w^*\|^2 \leq c^t \|w^0 - w^*\|^2 \quad (9.29)$$

7 where $c = \left(1 - \eta \frac{2mL}{m + L}\right)$.

8 **Proof.** We expand the left hand-side in (9.28) to get

$$\begin{aligned} \|w^{(t+1)} - w^*\|^2 &= \|w^{(t)} - \eta \nabla \ell(w^{(t)}) - w^*\|^2 \\ &= \|w^{(t)} - w^*\|^2 - 2\eta \langle \nabla \ell(w^{(t)}), w^{(t)} - w^* \rangle + \eta^2 \|\nabla \ell(w^{(t)})\|^2 \\ &\leq \left(1 - \eta \frac{2mL}{m + L}\right) \|w^{(t)} - w^*\|^2 + \eta \left(\eta - \frac{2}{m + L}\right) \|\nabla \ell(w^{(t)})\|^2 \\ &\leq \left(1 - \eta \frac{2mL}{m + L}\right) \|w^{(t)} - w^*\|^2. \end{aligned}$$

9 We have substituted the co-coercivity condition from (9.27) for the inner-
 10 product with $w' := w^{(t)}$ and $w := w^*$ to get the first inequality. This
 11 implies that the distance to the global minimum $\|w^{(t)} - w^*\|$ decreases
 12 multiplicatively; compare this with (9.25) where the progress is additive.
 13 The additional assumption of strong convexity therefore means that we
 14 are making very quick progress towards the global minimum. We can use
 15 this inequality repeatedly for all iterations t to get

$$\|w^{(t)} - w^*\|^2 \leq c^t \|w^0 - w^*\|^2$$

16 where $c = \left(1 - \eta \frac{2mL}{m + L}\right)$. \square

Strong convexity enables much faster progress towards the global

minimum. If we want $\|w^{(t)} - w^*\| \leq \epsilon$ we need

$$\mathcal{O}(\log(1/\epsilon))$$

iterations of gradient descent. This is *much* less than that for a convex function. Quite non-intuitively, this is called *linear* convergence because we need a constant number of iterations to reduce the gap to the optimal in half. The naming convention is a bit unusual here but you will see that if we plot $\log \|w^{(t)} - w^*\|$ (or $\log(\ell(w^{(t)}) - \ell(w^*))$) on the Y-axis and number of iterations t on the X-axis, we get a straight line for gradient descent on strongly-convex functions; you can see this from (9.29).

We say that the convergence rate of gradient descent for non-strongly convex functions is *sub-linear*. The longer we run GD for convex functions, the slower its progress.

Further, if we pick the largest step-size $\eta = 2/(m + L)$ we get

$$c = \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 < 1. \quad (9.30)$$

where $\kappa = L/m$ is the condition number of the Hessian (it is the ratio of the largest eigenvalue and the smallest eigenvalue). Larger the condition number κ , closer to 1 the multiplicative constant c and *slower* the convergence rate of gradient descent.

▲ Plot the convergence rate of gradient descent for convex and strongly-convex functions.

▲ The nomenclature is a bit non-intuitive in the optimization literature. An algorithm with

$$\lim_{t \rightarrow \infty} \frac{\ell(w^{(t+1)}) - \ell(w^*)}{\ell(w^{(t)}) - \ell(w^*)} = \rho$$

is said to be sub-linear if $\rho \in (0, 1)$, linear if $\rho = 1$ and super-linear if $\rho = 0$.

1 A few more points to note

2 1. The step-size is limited by $m + L$ but the convergence rate depends
3 on $\kappa = L/m$. Smaller the value of c , faster the convergence.

4 2. Larger the L , smaller the ideal step-size η

5 3. You can get the upper bound

$$\ell(w^{(t)}) - \ell(w^*) \leq \frac{L}{2} \|w^{(t)} - w^*\|^2 \leq \frac{c^t L}{2} \|w^0 - w^*\|^2 \quad (9.31)$$

6 using (9.20).

7 You will also see the convergence rate written in many papers as

$$\|w^{(t)} - w^*\| \leq e^{-4t/\kappa} \|w^0 - w^*\|. \quad (9.32)$$

8 You can get this inequality by using the fact that $1 + x \leq e^x$ in (9.30). We
9 can use this to pull out the dependence on κ in the convergence rate; for
10 strongly convex functions, gradient descent requires

$$\mathcal{O}(\kappa \log(1/\epsilon))$$

11 iterations to reach within an ϵ -neighborhood of the global minimum $\ell(w^*)$.

12 This suggests that smaller the condition number κ fewer the iterations

1 required.

2 We can intuitively understand why convergence of gradient descent is
 3 slower for a large condition number. A large condition number means that
 4 some directions of the objective ℓ are highly curved while some others
 5 are very flat. It is difficult to pick one single scalar step-size in such
 6 situations that makes quick progress along the flat directions but also does
 7 not overshoot the highly curved directions. You might imagine that clever
 8 schemes to change the step-size depending upon the local geometry of the
 9 function $\ell(w^{(t)})$ could help solve this issue and indeed it does, but such
 10 adaptive schemes are expensive to implement computationally. We will
 11 see some algorithms that pick different step-sizes for different weights in
 12 Chapter 11.

▲ Draw a picture of this phenomenon for a quadratic objective $\ell(w) = \langle w, Aw \rangle$ for matrices $A \succ 0$ with different condition numbers κ .

13 9.4 Limits on convergence rate of first-order 14 methods

15 It is a powerful and deep result that we cannot do better than a linear
 16 convergence rate for optimization methods that only use the gradient of
 17 the function $\ell(w)$. More precisely, for any first-order method, i.e., any
 18 method where the iterate at step t given by $w^{(t)}$ is chosen to be

$$w^{(t)} \in w^0 + \text{span} \{ \nabla \ell(w^0), \dots, \nabla \ell(w^t) \},$$

19 we have the following theorem by Yurii Nesterov.

20 **Theorem 9.7 (Nesterov's lower bound).** If $w \in \mathbb{R}^p$, for any $t \leq (p-1)/2$
 21 and every initialization of weights w^0 there exist functions $\ell(w)$ that are
 22 convex, differentiable, L -smooth with finite optimal value $\ell(w^*)$ such that
 23 any first-order method has

$$\ell(w^{(t)}) - \ell(w^*) \geq \frac{3}{32} \frac{L \|w^0 - w^*\|^2}{(t+1)^2}.$$

24 Let us read the statement of the theorem carefully. It states that
 25 *fixed* a time t and initial condition w^0 , we can *find* a convex function
 26 $\ell(w)$ such that it takes any first order method at least $\mathcal{O}(1/\sqrt{\epsilon})$ to reach
 27 an ϵ -neighborhood of the optimal value $\ell(w^*)$. The implication of this
 28 theorem is as follows. The convergence rate $\mathcal{O}(1/\epsilon)$ we obtained for
 29 convex functions is not the best rate we can get. Nesterov's lower bound
 30 suggests that there should be gradient-based algorithms that only require
 31 $\mathcal{O}(1/\sqrt{\epsilon})$ iterations. Such methods will be the topic of the next Chapter.

Chapter 10

Accelerated Gradient Descent

Reading

1. The blog-post titled “Why momentum really works?” at <https://distill.pub/2017/momentum>

In the previous chapter we saw two results that characterize how many iterations gradient descent requires to reach within an ϵ -neighborhood of the global optimum for convex functions. If the function $\ell(w)$ is convex, GD with a step-size at most $1/L$ requires $\mathcal{O}(1/\epsilon)$ iterations. If the function $\ell(w)$ is strongly convex, then the step-size can be as large as $2/(m + L)$ and GD requires $\mathcal{O}(\kappa \log(1/\epsilon))$ iterations where

$$\kappa = \frac{L}{m}$$

is the condition number of the Hessian $\nabla^2 \ell(w)$. A large value of κ means that there are some directions where the function is highly curved and others where it is relatively flat. The main point to remember is that we often do not know a good value for m, L . Since the step-size of GD depends upon the curvature of the function; if the step-size is too large then GD overshoots on the highly curved directions and if the step-size is too small then GD makes slow progress along a direction.

We will study two algorithms in this chapter which accelerate the progress of gradient descent.

10.1 Polyak’s Heavy Ball method

The most natural place to begin is to imagine gradient descent as a kinematic equation. Let $w^{(t)}$ be the iterate of GD at time t , let us associate

1 to it an auxiliary variable called the “velocity” $v^{(t)}$

$$v^{(t)} := w^{(t+1)} - w^{(t)}. \quad (10.1)$$

2 Gradient descent can then be written as

$$v^{(t)} = -\eta \nabla \ell(w^{(t)}), \quad (10.2)$$

3 which allows us to think of the term $-\nabla \ell(w^{(t)})$ as some kind of force
4 that acts on a particle to update its position from $w^{(t)}$ to $w^{(t+1)}$. This
5 particle has no inertia, so we will say that the applied force directly affects
6 its position. If the magnitude of the gradient is small in a certain direction,
7 the velocity is also small in that direction.

8 We now give our particle some inertia. Instead of the force directly
9 affecting the position we will write down Newton’s second law of motion
10 ($F = ma$) for a particle with unit mass $m = 1$ and time discretization η
11 (or equivalently, with a time discretization of 1 and a mass of η^{-1}) as

$$\begin{aligned} -\nabla \ell(w^{(t)}) &=: \frac{v^{(t+1)} - v^{(t)}}{\eta} \\ &= \frac{1}{\eta} \left(w^{(t+1)} - 2w^{(t)} + w^{(t-1)} \right) \\ \Rightarrow w^{(t+1)} &= w^{(t)} - \eta \nabla \ell(w^{(t)}) + \left(w^{(t)} - w^{(t-1)} \right). \end{aligned} \quad (10.3)$$

12 Notice the third term on the right-hand side above, it is the gap between
13 the current weights $w^{(t)}$ and the previous weights $w^{(t-1)}$, if we have

$$\left\langle w^{(t)} - w^{(t-1)}, \nabla \ell(w^{(t)}) \right\rangle < 0,$$

14 i.e., the change from the previous time-step is along the descent direction,
15 then the weights $w^{(t+1)}$ get an extra boost. If instead, the change from
16 the previous direction is not along the gradient descent direction, then
17 the third term reduces the magnitude of the gradient. The third term is
18 effectively the inertia of gradient updates. This method is therefore called
19 Polyak’s Heavy Ball method.

We give ourselves some more control over how inertia enters the update equation using a hyper-parameter ρ (which is akin to mass)

$$w^{(t+1)} = w^{(t)} - \eta \nabla \ell(w^{(t)}) + \rho \left(w^{(t)} - w^{(t-1)} \right). \quad (10.4)$$

If $\rho = 0$, we do not use any inertia and Polyak’s method boils down to gradient descent. Typically, we choose $\rho \in (0, 1)$. This inertia is called *momentum* in the optimization literature and ρ is called the momentum coefficient.

20 Polyak’s method is simple yet very powerful. In the previous chapter,
21 we showed a lower-bound of Nesterov which indicates that first-order
22 optimization algorithm (that only depends on the gradient of the objective)

cannot be faster than $\mathcal{O}(1/\sqrt{\epsilon})$. It turns out that Polyak's method converges at this rate, i.e., if we want

$$\|w^{(t)} - w^*\| \leq \epsilon$$

we need to run Polyak's Heavy Ball method for $\mathcal{O}(1/\sqrt{\epsilon})$ iterations for convex functions. If the function is strongly convex, the number of iterations comes down to

$$\mathcal{O}(\sqrt{\kappa} \log(1/\epsilon)).$$

Both of these are improvements upon their convergence rates for gradient descent. These improvements are also quite a lot, we need quadratically fewer iterations than gradient descent in Polyak's method and the only incremental cost of doing so is that we have to maintain a copy of the weights $w^{(t+1)}$ while implementing the updates in (10.4).

An alternative way to write Polyak's updates We can rewrite the updates in (10.4) using a dummy variable $u^{(t)}$ as

$$\begin{aligned} u^{(t)} &= (1 + \rho)w^{(t)} - \rho w^{(t-1)} \\ w^{(t+1)} &= u^{(t)} - \eta \nabla \ell(w^{(t)}). \end{aligned} \quad (10.5)$$

This is how these updates are implemented in PyTorch. This is convenient: effectively, the code needs to maintain only the difference $u^{(t)} = (1 + \rho)w^{(t)} - \rho w^{(t-1)}$ in a buffer $u^{(t)}$ and subtract the gradient $\nabla \ell(w^{(t)})$ from this buffer to result in the new updates. GD can be implemented with a simple change by setting $u^{(t)} := w^{(t)}$ with corresponds to $\rho = 0$. The dummy variable is initialized to $u^0 = w^0$.

A yet another way to write Polyak's updates We can also rewrite the updates in (10.5) as

$$\begin{aligned} u^{(t+1)} &= \rho u^{(t)} - \nabla \ell(w^{(t)}) \\ w^{(t+1)} &= w^{(t)} + \eta u^{(t+1)}. \end{aligned} \quad (10.6)$$

This set of updates brings out idea of momentum more clearly. The variable $u^{(t)}$ in this case is exactly the velocity $v^{(t)}$ that we have seen above except that it is updated slightly different than our expression ($F = ma$) in the first equation. The first term

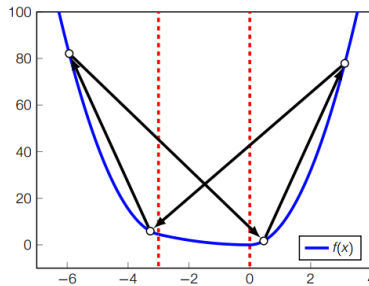
$$u^{(t+1)} = \rho u^{(t)} - \nabla \ell(w^{(t)})$$

reduces the velocity $u^{(t)}$ by a factor ρ before adding the gradient to it.

▲ Draw Polyak's updates for a two-dimensional function.

10.1.1 Polyak's method can fail to converge

The caveat with relying on the inertia of the particle to make progress is that near the global minimum, when the iterates overshoot the global minimum, the inertia is often very different from the gradient. Polyak's method can become unstable and can result in oscillations under such conditions, e.g.,



However it is a very simple method to accelerate gradient descent and works great in practice.

10.2 Nesterov's method

Nesterov's method is an advanced version of Polyak's method. Let us understand these oscillations better. We saw that incorporating a notion of inertia in Polyak's method gave us accelerated convergence; this is intuitive, if the velocity is along the descent direction the particle descends faster. The same inertia hurts towards the end because the velocity can be very different than the gradient when the particle overshoots the minimum.

A simple way to fix this is to incorporate damping (friction) into Newton's law of motion; you can read about the simple harmonic oscillator at https://en.wikipedia.org/wiki/Harmonic_oscillator. We are going to write

$$ma = F - cv.$$

where m is the mass, c is the coefficient of damping, a and v are acceleration and velocity respectively and F is the force as usual. The effective force decreases with the velocity. Doing so does not slow down the weight updates much when the gradient magnitude is large at the beginning of training. But when the gradient magnitude is small (when the particle is in the neighborhood of the global minimum), this friction prevents excessive overshooting.

With that background, let us see how Nesterov's updates for gradient descent look.

We will write a similar set up of updates as that of (10.6).

Nesterov's updates correspond to

$$\begin{aligned} u^{(t+1)} &= \rho u^{(t)} - \nabla \ell(w^{(t)} + \eta \rho u^{(t)}) \\ w^{(t+1)} &= w^{(t)} + \eta u^{(t+1)}. \end{aligned} \quad (10.7)$$

The only difference between (10.7) and (10.6) is the term in blue; effectively the gradient is computed as if the weights $w^{(t)}$ moved using the velocity $u^{(t)}$; and then this new velocity $u^{(t+1)}$ is used to obtain the new updates $w^{(t+1)}$. Nesterov's method solves the problem of instability in Polyak's method by essentially computing the gradient (the blue term) as given by the current velocity. You can see how this *slows down* the updates if the velocity is well-aligned with the gradient; we are reducing the benefit of inertia/momentum. However, doing so prevents overshooting as the iterates reach the neighborhood of the global minimum.

- 1 **An alternative way to write Nesterov's updates** We can rewrite the
2 updates in (10.7) to look like those in (10.5), to get

$$\begin{aligned} u^{(t)} &= (1 + \rho)w^{(t)} - \rho w^{(t-1)} \\ w^{(t+1)} &= u^{(t)} - \eta \nabla \ell(u^{(t)}). \end{aligned} \quad (10.8)$$

- 3 Again the blue term is the only difference between Polyak's method and
4 Nesterov's method. The term $u^{(t)}$ is conceptually a forecasted version of
5 the weights $w^{(t)}$ because notice that

$$u^{(t)} = w^{(t)} + \rho(w^{(t)} - w^{(t-1)}).$$

- 6 The new weights $w^{(t+1)}$ are now obtained by thinking of $u^{(t)}$ as the actual
7 weights. We initialize $w^{(t+1)} = w^{(t)}$ to w^0 for $t = 0$.

8 10.2.1 A model for understanding Nesterov's updates

- 9 We will set the damping coefficient (ρ) in (10.8) to a special value

$$\rho = \frac{t-1}{t+2};$$

- 10 effectively as $t \rightarrow \infty$ the friction becomes larger and larger. This simplifies
11 our updates to

$$\begin{aligned} u^{(t)} &= w^{(t)} + \frac{t-1}{t+2} (w^{(t)} - w^{(t-1)}) \\ w^{(t+1)} &= u^{(t)} - \eta \nabla \ell(u^{(t)}). \end{aligned}$$

- 12 which upon collapsing together give

$$w^{(t+1)} - w^{(t)} = \frac{t-1}{t+2} (w^{(t)} - w^{(t-1)}) - \eta \nabla \ell(u^{(t)}). \quad (10.9)$$

1 We now choose a different way of interpreting these updates. We will
2 imagine that the sequence

$$\{w^0, w^1, \dots, w^{(t)}, w^{(t+1)}, \dots\}$$

3 is the discretization of a smooth curve

$$\{W(\tau) : \tau \in [0, \infty)\}.$$

4 How is this curve $W(\tau)$ related to the original sequence? We are going to
5 study the updates under the setting

$$\tau := \sqrt{\eta} t. \quad (10.10)$$

6 Essentially the time of the discrete-time updates (10.9) increments in
7 intervals of 1, but the time of the curve $W(\tau)$ is real-number. Each
8 increment in discrete-time corresponds to $\sqrt{\eta}$ increment of the time τ for
9 the curve $W(\tau)$. This gives

$$\begin{aligned} W(\tau) &= w^{(t)} \\ W(\tau + \sqrt{\eta}) &= w^{(t+1)}. \end{aligned}$$

10 We now do a Taylor expansion for the continuous curve $X(\tau)$ to get

$$\begin{aligned} w^{(t+1)} - w^{(t)} &= W(\tau + \sqrt{\eta}) - W(\tau) \\ &= \dot{W}(\tau)\sqrt{\eta} + \frac{1}{2}\ddot{W}(\tau)\eta + \mathcal{O}(\eta). \end{aligned} \quad (10.11)$$

11 Here

$$\dot{W}(\tau) = \frac{d}{d\tau} W(\tau), \quad \ddot{W}(\tau) = \frac{d^2}{d\tau^2} W(\tau)$$

12 are the first and second derivative of the curve with respect to time and
13 $\mathcal{O}(\sqrt{\eta})$ denotes higher-order terms. Similarly

$$\begin{aligned} w^{(t)} - w^{(t-1)} &= W(\tau) - W(\tau - \sqrt{\eta}) \\ &= \dot{W}(\tau)\sqrt{\eta} - \frac{1}{2}\ddot{W}(\tau)\eta + \mathcal{O}(\eta). \end{aligned}$$

14 Note that due to our special scaling of time we have

$$\frac{t-1}{t+2} = 1 - \frac{3}{t+2} \approx 1 - \frac{3}{t} = 1 - \frac{3\sqrt{\eta}}{\tau}.$$

15 We now do a Taylor expansion of the loss term $\nabla \ell(u^{(t)})$ to get

$$\begin{aligned} \nabla \ell(u^{(t)}) &= \nabla \ell \left(w^{(t)} + \frac{t-1}{t+2} (w^{(t)} - w^{(t-1)}) \right) \\ &= \nabla \ell(w^{(t)}) + \text{higher order terms} \\ &= \nabla \ell(W(\tau)) + \mathcal{O}(\sqrt{\eta}). \end{aligned} \quad (10.12)$$

1 Substitute (10.11) and (10.12) in (10.9) and divide both side by $\sqrt{\eta}$ to get

$$\dot{W}(\tau) + \frac{1}{2}\ddot{W}(\tau)\sqrt{\eta} + \mathcal{O}(\sqrt{\eta}) = \left(1 - \frac{3\sqrt{\eta}}{\tau}\right) \left(\dot{W}(\tau) - \frac{1}{2}\ddot{W}(\tau)\sqrt{\eta} + \mathcal{O}(\sqrt{\eta})\right) - \sqrt{\eta} \nabla\ell(W(\tau)) + \mathcal{O}(\eta).$$

2 This equation is true for all values of η , so we can compare the coefficients
3 of $\sqrt{\eta}$ on both sides to get

$$\ddot{W} + \frac{3}{\tau}\dot{W} + \nabla\ell(W) = 0. \quad (10.13)$$

4 This equation looks very similar to Newton's law with friction $ma + cv =$
5 F . Again, the term $\nabla\ell(W)$ is acting as the force, the second derivative
6 \ddot{W} is the acceleration and the friction term $\frac{3}{t}\dot{W}$ increases with velocity.
7 We have shown that for a particularly chosen value of the momentum
8 coefficient, Nesterov's updates result in an ordinary differential equation
9 that looks much like that simple harmonic oscillator that most of you have
10 seen before in high-school. This approach gives an alternative, and very
11 simple, way of understanding Nesterov's updates which is nice because
12 the updates in (10.7) and (10.8) were quite non-intuitive and created by
13 Nesterov through a sheer *tour de force*.

14 **Remark 10.1.** Derive a similar ordinary differential equation for Polyak's
15 updates using the same setting of friction $(t-1)/(t+2)$ as that in (10.9).
16 You will notice that if viewed in continuous-time Polyak's updates are
17 exactly the same as Nesterov's updates. This is because the continuous-
18 time model is a more abstract point-of-view and eliminates the subtle
19 differences between the updates between the two algorithms.

20 Such continuous-time models are very useful to understand what these
21 updates actually do, e.g., we know that Nesterov's updates correspond to
22 having damping in Newton's law which is not apparent by looking at the
23 equations in (10.8). It is also very easy to obtain the convergence rate of
24 the continuous-time version; it is an ordinary differential equation and
25 we can use a lot of tools from dynamical systems, in particular Lyapunov
26 functions. It will amuse you to know that obtaining the convergence rate
27 for Nesterov's updates using the continuous-time version (10.13) takes
28 about half a page but doing the same proof in discrete-time (like Nesterov
29 did it originally) takes a few dozen pages.

30 10.2.2 How to pick the momentum parameter?

31 Nesterov's updates converge at a rate that is similar to that of Polyak's
32 updates. For convex functions, we need

$$\mathcal{O}(1/\sqrt{\epsilon})$$

33 iterations to get within the ϵ -neighborhood of the global minimum if we
34 set

$$\rho = (t-1)/(t+2)$$

1 in (10.6). If we are minimizing a strongly-convex function we can pick
2 the momentum coefficient to depend on m, L : we can set

$$\rho = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \quad (10.14)$$

3 and $\eta < 2/(m + L)$. If we do so, we need

$$\mathcal{O}(\sqrt{\kappa} \log(1/\epsilon))$$

4 weight updates to reach within an ϵ -neighborhood of the global minimum.
5 The expression in (10.14) gives some insight in how momentum accelerates
6 things. If $\kappa \approx 1$, i.e., the Hessian of the objective is well-conditioned
7 without a big diversity in the curvature in different directions, we do not
8 really need friction $\rho \approx 0$ to avoid overshooting close to the minimum;
9 progress in all directions is balanced. On the other hand, if $\kappa \gg 1$, the
10 objective is badly conditioned and the friction coefficient $\rho \approx 1$ should be
11 large to avoid overshooting near the global minimum.

12 **How to pick ρ in practice?** If we know what m, L are for a given
13 problem (you will see an example of this in HW 4), it is straightforward
14 to pick the momentum coefficient and get accelerated convergence of
15 gradient descent. If we do not know m, L , we pick some constant value of
16 ρ . For instance, $\rho = 0.9$ is popularly used in most deep learning libraries.
17 Typically, the momentum coefficient is not increased with the number
18 of weight updates using $(t - 1)/(t + 2)$. You will see some heuristics for
19 modifying the momentum coefficient in this week's recitation.

Chapter 11

Stochastic Gradient Descent

Reading

1. “Stochastic gradient descent tricks” by [Bottou \(2012\)](#). Great paper with lots of little tricks of how to use SGD in practice.
2. Up to Section 4.2 of “Optimization methods for large-scale machine learning” by [Bottou et al. \(2018\)](#). This is advanced material, you do not need to be able to follow it completely.
3. http://fa.bianp.net/teaching/2018/eecs227at/stochastic_gradient.html
4. “Adam: A Method for Stochastic Optimization” by [Kingma and Ba \(2014\)](#).
5. Stochastic Weight Averaging (SWA) by [Izmailov et al. \(2018\)](#).

Stochastic Gradient Descent (SGD) has its roots in stochastic optimization. A stochastic optimization problem looks like

$$w^* = \operatorname{argmin}_w \mathbb{E}_\xi[\ell(w, \xi)] \quad (11.1)$$

where ξ is a random variable. This is a very old and rich area, there was lots of action in it already in the 1950s, e.g., ([Kushner and Yin, 2003](#); [Robbins and Monro, 1951](#)). It is also a highly relevant problem: for instance, when a plane goes from Los Angeles to Philadelphia, the route that the plane takes depends on the local weather conditions along its path and airlines will optimize this route using a stochastic optimization problem of the above form. The variable w will be the trajectory of the plane and ξ are the weather conditions which we do not know exactly but may perhaps have estimated a distribution for them. Such problems are

1 very common in other fields like operations research, e.g., optimizing the
 2 time at which an Amazon package arrives with various disturbances such
 3 as delays in shipping, missing inventory in the warehouse etc.

4 In machine learning, we are interested in solving a slightly different
 5 problem called the finite-sum problem. Given a finite dataset $D =$
 6 $\{(x^i, y^i)\}_{i=1, \dots, n}$ we minimize

$$\ell(w) := \frac{1}{n} \sum_{i=1}^n \ell^i(w) \quad (11.2)$$

7 where we will use the shorthand

$$\ell^i(w) := \ell(w; x^i, y^i)$$

8 to denote the loss on the datum (x^i, y^i) with weights w . Essentially, the
 9 random variable ξ in (11.1) represents the samples in the training dataset;
 10 with important differences being that neither do we know anything about
 11 the distribution of the input data, nor do we have an infinite number of
 12 samples.

13 It is difficult to do gradient descent if the number of samples n is large
 14 because the gradient is a summation of a large number of terms

$$\nabla \ell(w) = \frac{1}{n} \sum_{i=1}^n \nabla \ell^i(w).$$

15 If the mini-batch size is 1, i.e., at each iteration we sample one of the
 16 training samples denoted by

$$\omega_t \in \{1, \dots, n\}$$

17 we update the weights using

$$w^{(t+1)} = w^{(t)} - \eta \nabla \ell^{\omega_t}(w^{(t)}). \quad (11.3)$$

18 For a larger mini-batch of size ℓ let us denote the samples in the mini-batch
 19 by

$$\{(x^{\omega_t^1}, y^{\omega_t^1}), \dots, (x^{\omega_t^\ell}, y^{\omega_t^\ell})\}$$

20 where each $\omega_t^k \in \{1, \dots, n\}$ is the index chosen uniformly randomly
 21 from the training dataset. We will choose these indices with replacement
 22 (analyzing SGD for sampling without replacement is quite hard). The
 23 gradient on this sampled mini-batch is denoted by

$$\nabla \ell_\ell(w) := \frac{1}{\ell} \sum_{i=1}^{\ell} \ell^{\omega_t^i}(w) \quad (11.4)$$

24 and we update the weights as usual using

$$w^{(t+1)} = w^{(t)} - \eta \nabla \ell_\ell(w^{(t)}).$$

1 If $\ell = 1$, we will denote the gradient by $\nabla \ell_\omega$ to keep the notation clear.

2 **What is an epoch in PyTorch?** We will not think of epochs when
 3 we develop the theory for SGD. An epoch is a construct introduced in
 4 deep learning libraries for bookkeeping purposes. It also ensures that if
 5 Algorithm A obtains so and so training/validation error after 100 epochs,
 6 it can be compared directly with Algorithm B which obtains the same
 7 training/validation error after, say, 120 epochs, e.g., one can say Algorithm
 8 A is faster than Algorithm B at training a network. Instead of sampling a
 9 mini-batch of data uniformly randomly with replacement, PyTorch shuffles
 10 the entire training set at the beginning of each epoch and samples the
 11 mini-batch *with replacement* during each epoch. This is reasonable but
 12 there will be some discrepancies in the performance of SGD as predicted
 13 by theory and obtained by PyTorch on deep networks, especially if the
 14 mini-batch size is large.

15 Although we will not discuss this, SGD using mini-batches sam-
 16 pled with replacement is faster than with mini-batches sampled without
 17 replacement (Recht and Ré, 2012).

18 11.1 SGD for least-squares regression

19 Let us understand SGD for one dimensional least-squares, our data and
 20 targets are $x^i, y^i \in \mathbb{R}$ and the objective is

$$\ell(w) = \frac{1}{2n} \sum_{i=1}^n (x^i w - y^i)^2 \quad (11.5)$$

21 for the weights $w \in \mathbb{R}$. Notice that the objective is a sum of n different
 22 quadratics, each quadratic is minimized by *different* weights

$$w^*(i) := \frac{y^i}{x^i};$$

23 in other words, each sample in the training dataset would like the weight
 24 to be y^i/x^i to minimize its residual and the least-squares objective which
 25 sums up their individual residuals forces them to made trade-offs. Focus
 26 on two quantities

$$w_{\min} = \min_i \{w^*(i)\}, \quad w_{\max} = \max_i \{w^*(i)\}.$$

27 Notice that the interval $(-\infty, w_{\max})$ is the region where the descent
 28 direction on any sample in the dataset moves the weights $w^{(t)}$ to the right.
 29 The interval (w_{\max}, ∞) is the region where the descent direction on any
 30 sample moves the weights to the left. If weights are initialized in the
 31 latter region, $w^0 \gg \max_i w^*(i)$, successive iterations of SGD will quickly
 32 bring the weights to

$$w^{(t)} \in (w_{\min}, w_{\max}) \quad (11.6)$$

▲ Draw the objective here for different values of w^i and understand how SGD works for this problem.

1 which we will call the “zone of confusion”. Similarly, if weights are
 2 initialized $w^o \ll w_{\min}$, they will move right until iterates reach the zone
 3 of confusion.

After $w^{(t)} \in (w_{\min}, w_{\max})$, there is no real convergence of the weights, if the learning rate η is fixed, since the samples ω_t are sampled uniformly randomly, depending upon which sample was chosen to compute the gradient the weights move to the right or the left and therefore keep shuttling back and forth in this region.

Notice that the objective in (11.5) is convex because it is the sum of convex functions so there is a unique global minimum namely

$$w^* = \frac{\sum_{i=1}^n x^i y^i}{\sum_{i=1}^n (x^i)^2}.$$

If one were to execute gradient descent on this same problem $w^{(t+1)} = w^{(t)} - \eta \nabla \ell(w^{(t)})$, we will converge to this value. But since SGD samples a different sample at each iteration, SGD never converges, it remains in this large zone (w_{\min}, w_{\max}) .

4 11.2 Convergence of SGD

5 If the learning rate is large, SGD makes quick progress outside the zone
 6 of confusion but bounces around a lot inside the zone of confusion. If the
 7 learning rate is too small, SGD is slow outside the zone of confusion but
 8 does not bounce around too much inside the zone. You can explore how
 9 the learning rate changes the dynamics of SGD at
 10 http://fa.bianp.net/teaching/2018/eecs227at/stochastic_gradient.html.

11 In this section, we will study under what conditions SGD converges to
 12 the global minimum and how the learning rate of SGD should be reduced
 13 to make it converge quickly. We will first analyze SGD with mini-batch
 14 size of 1.

15 **Strongly convex functions** The proofs for convex functions are tedious,
 16 so we will only consider strongly convex functions in this section. As
 17 usual the strong convexity parameter is m and smoothness parameter is L .
 18 One key thing to notice that these constants L, m refer to the full objective,
 19 i.e.,

$$\|\nabla \ell(w) - \nabla \ell(w')\| \leq L \|w - w'\|$$

20 and

$$\ell(w) - \frac{m}{2} \|w\|^2 \text{ is convex.}$$

21 Here $\ell(w)$ is the *full objective*

$$\ell(w) = \frac{1}{n} \sum_{i=1}^n \ell^i(w).$$

1 **What is the appropriate notion of convergence?** The key difference
 2 between updates of SGD and those of GD is that SGD updates also depend
 3 on the random variable ω_t . The iterate w_t is a *random variable* and
 4 therefore instead of simply bounding the gap $\ell(w^{(t)}) - \ell(w^*)$ we will
 5 have to obtain an upper bound for

$$\mathbb{E}_{w^{(t)}} [\ell(w^{(t)})] - \ell(w^*).$$

6 Similar to the case of SGD, let us construct a descent lemma for one
 7 iteration of SGD update.

Lemma 11.1 (Descent Lemma for SGD).

$$\begin{aligned} \mathbb{E}_{\omega_t} [\ell(w^{(t+1)}) - \ell(w^{(t)}) \mid w^{(t)}] &\leq -\eta \left\langle \nabla \ell(w^{(t)}), \mathbb{E}_{\omega_t} [\nabla \ell^{\omega_t}(w^{(t)})] \right\rangle \\ &\quad + \frac{L\eta^2}{2} \mathbb{E}_{\omega_t} \left[\|\nabla \ell^{\omega_t}(w^{(t)})\|^2 \right]. \end{aligned} \quad (11.7)$$

8 **Proof.** First, compare this with the descent lemma for gradient descent
 9 (if we substitute $w^{(t+1)} - w^{(t)} = -\eta \nabla \ell(w^{(t)})$ from Chapter 9)

$$\ell(w^{(t+1)}) - \ell(w^{(t)}) \leq -\eta \left\langle \nabla \ell(w^{(t)}), \nabla \ell(w^{(t)}) \right\rangle + \frac{L\eta^2}{2} \|\nabla \ell(w^{(t)})\|^2$$

10 The only difference now is that in the case of SGD we have

$$w^{(t+1)} - w^{(t)} = -\eta \nabla \ell^{\omega_t}(w^{(t)}).$$

11 The most important difference however is that the descent term, namely the
 12 left-hand side in (11.7) is conditioned on the random variable $w^{(t)}$. The
 13 proof of this lemma is easy, we simply substitute the expression for the
 14 weight updates of SGD and take an expectation over the index of datum
 15 sampled by SGD ω_t on both sides of the inequality. \square

16 The implication of the above lemma is that SGD updates need more refined
 17 conditions under which we can claim monotonic progress towards the
 18 global minimum. Effectively, we need to make sure that the right-hand
 19 side is negative, *always* irrespective of what the value of the random
 20 variable $w^{(t)}$ is. We would like to upper bound the right-hand side by a
 21 deterministic quantity ideally.

22 11.2.1 Typical assumptions in the analysis of SGD

23 1. **Stochastic gradients are unbiased.** Assume that the stochastic
 24 gradient is unbiased

$$\nabla \ell(w) = \mathbb{E}_{\omega} [\nabla \ell^{\omega}(w)] \quad (11.8)$$

25 for all w in the domain. This is akin to assuming that the way we
 26 sample images in the mini-batch is such that the average is always

1 pointing towards the true gradient with a similar magnitude. This is
 2 a natural condition and will only change if the sampling distribution
 3 is not uniform. This assumption allows to control the first term in
 4 the descent lemma.

5 **2. Second moment of gradient norm does not grow too quickly.**

6 We will assume that there exist scalars σ_0 and σ such that

$$\mathbb{E}_{\omega_t} \left[\|\nabla \ell^{\omega_t}(w)\|^2 \right] \leq \sigma_0 + \sigma \|\nabla \ell(w)\|^2. \quad (11.9)$$

7 This assumption allows to control the second term in the descent
 8 lemma for SGD. It assumes that the stochastic estimate of the
 9 gradient in SGD $\nabla \ell^{\omega_t}(w)$ is not too different than the full gradient
 10 $\nabla \ell(w)$. In the neighborhood of a critical point (locations where
 11 the full gradient $\nabla \ell(w) = 0$), the stochastic gradient is allowed to
 12 grow in a similar fashion as the true gradient except with a scaling
 13 factor $\sigma > 0$ and a constant σ_0 .

14 Let us see how the descent lemma changes with these additional
 15 assumptions.

16 **Lemma 11.2 (Descent Lemma for SGD with additional assumptions).**

17 If SGD gradients are unbiased and the second moment of the stochastic
 18 gradients can be bounded, we have

$$\begin{aligned} & \mathbb{E}_{\omega_t} \left[\ell(w^{(t+1)}) - \ell(w^{(t)}) \mid w^{(t)} \right] \\ & \leq -\eta \left\langle \nabla \ell(w^{(t)}), \mathbb{E}_{\omega_t} \left[\nabla \ell^{\omega_t}(w^{(t)}) \right] \right\rangle + \frac{L\eta^2}{2} \mathbb{E}_{\omega_t} \left[\left\| \nabla \ell^{\omega_t}(w^{(t)}) \right\|^2 \right] \\ & \leq -\eta \left\| \nabla \ell(w^{(t)}) \right\|^2 + \frac{L\eta^2}{2} \mathbb{E}_{\omega_t} \left[\left\| \nabla \ell^{\omega_t}(w^{(t)}) \right\|^2 \right] \\ & = -\eta \left(1 - \frac{\eta L \sigma}{2} \right) \left\| \nabla \ell(w^{(t)}) \right\|^2 + \frac{\eta^2 L \sigma_0}{2}. \end{aligned} \quad (11.10)$$

19 The proof is given in (11.10) itself. Compare this to the corresponding
 20 result we have derived for gradient descent in Chapter 9

$$\ell(w^{(t+1)}) - \ell(w^{(t)}) \leq -\frac{\eta}{2} \left\| \nabla \ell(w^{(t)}) \right\|^2.$$

21 In addition to the negative term $-\frac{\eta}{2} \left\| \nabla \ell(w^{(t)}) \right\|^2$, we have two additional
 22 positive terms

$$\frac{\eta^2 L \sigma}{2} \left\| \nabla \ell(w^{(t)}) \right\|^2 + \frac{\eta^2 L \sigma_0}{2};$$

23 this indicates that depending upon the magnitude of these terms we may
 24 not get monotonic improvement of the objective for SGD. There is no
 25 such concern for gradient descent, we get monotonic progress at all parts
 26 of the domain.

We need to pick the learning rate η in such a way that balances the the right-hand side of (11.10) and makes it negative.

11.2.2 Convergence rate of SGD for strongly-convex functions

Theorem 11.3 (Optimality gap for SGD). If we pick a step-size

$$\eta \leq \frac{1}{L\sigma}$$

for m -strongly convex and L -smooth function $\ell(w)$ then the expected optimality gap satisfies

$$\begin{aligned} & \mathbf{E}_{\omega_1, \omega_2, \dots, \omega_t} [\ell(w^{(t+1)})] - \ell(w^*) \\ & \leq \frac{\eta L \sigma_0}{2m} + (1 - \eta m)^t \left(\ell(w^0) - \ell(w^*) - \frac{\eta L \sigma_0}{2m} \right). \end{aligned} \quad (11.11)$$

We will not cover the proof of this theorem, it is a direct application of the descent lemma. See Bottou et al. (2018, Theorem 4.6) for an elaborate proof.

This theorem beautifully demonstrates the interplay between the step-size and the variance of SGD gradients. If there is no stochasticity, i.e., $\sigma_0 = 0$ and $\sigma = 1$, we get the same result as that of gradient descent, namely, the function value $\ell(w^{(t+1)})$ converges at a *linear rate* $(1 - \eta m)^t$. Some points to notice

1. The random variable $w^{(t+1)}$ depends upon all the indices $\omega_1, \omega_2, \dots, \omega_t$ that were sampled during updates of SGD and therefore the expectation in (11.11) should be over all these random variables.
2. When the stochastic gradient is noisy, we have a non-zero σ_0 we can no longer get to the global minimum, there is a first term which does not decay with time.
3. If we pick a small η , we get closer to the global minimum but go there quite slowly. On the other hand, we can pick a large η and get to a neighborhood of the global minimum quickly but we will then have a large error leftover at the end.

How can we make SGD converge and drive down the first term

in (11.11) to zero? A simple trick is to reduce the learning rate η with time. We do not want to decay the learning rate too quickly however because the second term in (11.11) is small, i.e., optimality gap is reduced quickly by its multiplicative nature, for a large value of the learning rate. A good schedule to pick is

$$\sum_{t=1}^{\infty} \eta_t = \infty, \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty. \quad (11.12)$$

1 **Heuristic for training neural networks** The two terms in the con-
 2 vergence rate of SGD explain the widely used heuristic of “divide the
 3 learning rate by some constant” if the training error seems plateaued. We
 4 are reducing the size of the ball in which SGD iterates bounce around by
 5 doing so.

6 **Theorem 11.4 (Convergence rate of SGD for decaying step-size).** For
 7 a schedule

$$\eta_t = \frac{\beta}{t + t_0} \text{ where } \beta > \frac{1}{m} \text{ and } t_0 \text{ is such that } \eta_1 < \frac{1}{L\sigma}$$

8 then the expected optimality gap satisfies

$$\mathbb{E}_{\omega_1, \omega_2, \dots, \omega_t} [\ell(w^{(t+1)})] - \ell(w^*) = \mathcal{O}\left(\frac{1}{t + t_0}\right). \quad (11.13)$$

9 We will not do the proof. If you are interested, see [Bottou et al. \(2018,](#)
 10 [Theorem 4.7\)](#). Compare this to the convergence rate of $\mathcal{O}(\kappa \log(1/\epsilon))$ for
 11 gradient descent for strongly-convex functions. Notice that we converge
 12 only at a sub-linear rate for SGD even for strongly convex loss functions.
 13 SGD is a much slower algorithm than GD.

14 **Convergence rate for mini-batch SGD** The mini-batch gradient $\nabla \ell_{\ell}(w)$
 15 is still an unbiased estimate of the full-gradient

$$\mathbb{E}_{\ell} [\nabla \ell_{\ell}(w)] = \nabla \ell(w)$$

16 but the second assumption in SGD improves a bit. Since the mini-batch
 17 gradient is averaged over ℓ samples we have

$$\mathbb{E}_{\ell} [\|\nabla \ell_{\ell}(w)\|^2] \leq \frac{\sigma_0}{\ell} + \frac{\sigma}{\ell} \|\nabla \ell(w)\|^2$$

1 if σ_0, σ were the constants in (11.9). This changes the convergence rate
2 in Theorem 11.3 to

$$\begin{aligned} & \mathbb{E}_{\omega_1, \omega_2, \dots, \omega_t} \left[\ell(w^{(t+1)}) \right] - \ell(w^*) \\ & \leq \frac{\eta L \sigma_0}{2m\ell} + (1 - \eta m)^t \left(\ell(w^0) - \ell(w^*) - \frac{\eta L \sigma_0}{2m\ell} \right). \end{aligned} \quad (11.14)$$

3 Note that the maximum learning rate in Theorem 11.3 is inversely pro-
4 portional to σ so we can also pick a larger learning rate $\eta < \frac{\ell}{L\sigma}$. If we
5 do so, the first and last terms above are not affected by the batch-size but
6 multiplicative term $(1 - \eta m)^t$ is. Since

$$(1 - \eta m)^t \leq e^{-tm\eta}$$

7 we see that increasing the learning rate by a factor of ℓ will reduce the
8 number of iterations required to reach the zone of confusion by a factor
9 of ℓ . Of course, this comes with the caveat that each iteration also
10 requires $\mathcal{O}(\ell)$ more computation to compute the gradient compared to
11 single-sample SGD.

12 11.2.3 When should one use SGD in place of GD?

13 Theorem 11.4 indicates that SGD is a very slow algorithm, GD is much
14 faster than SGD to minimize strongly convex functions. This gap also
15 exists if we do not have strong convexity: we did not prove this but SGD
16 requires $\mathcal{O}(1/\epsilon^2)$ to reach an ϵ -neighborhood of the global optimum for
17 convex functions whereas GD requires a much smaller $\mathcal{O}(1/\epsilon)$. One
18 might wonder why we should use SGD at all.

19 It is critical to remember that the objective in machine learning is a
20 sum of many terms

$$\ell(w) = \frac{1}{n} \sum_{i=1}^n \ell^i(w)$$

21 One iteration of SGD requires us to compute only $\nabla \ell^{\omega_t}(w)$ whereas one
22 update of GD requires us to compute the full gradient $\nabla \ell(w)$. One weight
23 update of GD is $\mathcal{O}(n)$ more expensive than one weight update using SGD.
24 Let us do a back-of-the-envelope computation for convex functions. If
25 we want to reach an ϵ -neighborhood of the global minimum of a convex
26 function, we need $\mathcal{O}(1/\epsilon)$ iterations of GD, which requires

$$\mathcal{O}\left(\frac{n}{\epsilon}\right)$$

27 operations. SGD needs $\mathcal{O}(1/\epsilon^2)$ iterations and therefore requires

$$\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$$

28 operations to reach the ϵ -neighborhood. This indicates that if our chosen

1 ϵ -ball is

$$\epsilon \lesssim \frac{1}{n}$$

2 GD requires fewer overall operations. But if ϵ -ball is larger than this, we
3 should use SGD because it is computationally cheaper.

4 SGD is particularly suited to machine learning compared to GD for
5 the following reason. Let $\epsilon^i = \ell^i(w^{(t)}) - \ell^i(w^*)$ be the residual on the
6 i^{th} datum in the training dataset. Observe that our ϵ -neighborhood is

$$\epsilon = \ell(w^{(t)}) - \ell(w^*) = \frac{1}{n} \sum_{i=1}^n \epsilon^i.$$

7 If ϵ^i is constant and does not depend on the number of training samples
8 n (i.e., say we are happy with the average error over the training dataset
9 being 2% even and do not seek a smaller one even if we collect more
10 data) then we should use SGD to train our model because it is cheaper.
11 This is not always the case for other problems, e.g., if you are doing
12 computational tomography (capturing multiple images from a CT-scan
13 machine and trying to reconstruct the heart/lung region in the thoracic
14 cavity), we may seek a more and more accurate answer, i.e., small ϵ if we
15 have more data.

16 11.3 Accelerating SGD using momentum

17 The convergence rate of SGD is quite bad, it is sub-linear. Roughly
18 speaking, the successive iterates of SGD are computed using different mini-
19 batches; the gradient on each such mini-batch is a noisy approximation
20 of the full-gradient on the training dataset (that of GD). This makes the
21 SGD iterates noisy and one may improve the convergence rate of SGD
22 by simply averaging the weights. This leads to a simple technique to
23 accelerate SGD which we discuss next.

24 **Polyak-Ruppert averaging** Consider the updates

$$\begin{aligned} w^{(t+1)} &= w^{(t)} - \eta_t \nabla \ell_{\tilde{b}}(w^{(t)}) \\ u^{(t)} &= \frac{w^0 + w^1 + \dots + w^t}{t}. \end{aligned} \tag{11.15}$$

25 In a series of papers, [Polyak \(1990\)](#); [Polyak and Juditsky \(1992\)](#); [Ruppert](#)
26 [\(1988\)](#) showed that the quantity

$$\mathbf{E}_{\omega_1, \dots, \omega_{t-1}} \left[\ell(u^{(t)}) \right] - \ell(w^*)$$

27 converges faster than the quantity

$$\mathbf{E}_{\omega_1, \dots, \omega_{t-1}} \left[\ell(w^{(t)}) \right] - \ell(w^*);$$

both of these still converge at rate $\mathcal{O}(1/\epsilon^2)$ but the former has a smaller constant. This is quite surprising and useful: essentially we are still performing mini-batch updates for the weights $w^{(t)}$ but instead of thinking of $w^{(t)}$ as the answer, we think of $u^{(t)}$ as the output of SGD. This averaging of iterates does not change the SGD algorithm. Computing this output requires us to remember all the past iterations w^0, \dots, w^t but we can easily approximate that step by exponential averaging of the *weights*

$$u^{(t)} = \rho u^{(t-1)} + (1 - \rho) w^{(t)};$$

exponential averaging is likely to achieve the same purpose with a much smaller memory requirement.

Further, this idea of using averaged iterates to speed up stochastic optimization algorithms is quite general and also works for algorithms other than SGD. The paper on Stochastic Weight Averaging by [Izmailov et al. \(2018\)](#) performs weight averaging (with quite different motivations) and works very well in practice.

11.3.1 Momentum methods do not accelerate SGD

We saw that momentum is very useful to accelerate the convergence of gradient descent. The power of momentum lies in making faster progress using the inertia of the particle: if the velocity and the current gradient are aligned with each other (as is the case at the beginning of training when the iterates are far from the global optimum) momentum speeds up things. Towards the end of training when gradients are typically mis-aligned with the velocity, we need friction (as in Nesterov's updates) to reduce this effect.

Observe that in SGD, the gradient is *always* incorrect; it is after all only a noisy stochastic approximation of the full gradient on the dataset. Since the velocity $w^{(t)} - w^{(t-1)}$ was computed using the previous stochastic gradient, there is no reason to believe that this velocity is accurate and will speed up SGD. Here is a very important point ([Kidambi et al., 2018](#); [Liu and Belkin, 2018](#)) that you should remember.

Momentum methods (Polyak's or Nesterov's) do not significantly accelerate SGD.

To be more precise, we saw that for Nesterov's updates in GD for strongly-convex functions we have a result of the form

$$\|w^{(t)} - w^*\| \leq e^{-t/\sqrt{\kappa}} \|w^0 - w^*\|$$

while the constant without momentum is larger, it is $e^{-t/\kappa}$. This term is directly related to the second term in [Theorem 11.4](#). The above authors come up with counterexamples to show that Nesterov's updates with SGD only improve this multiplicative term to something like $e^{-ct/\kappa}$ for some c ; in other words using Nesterov's updates with SGD only lead to a constant factor improvements in the convergence rate.

Accelerating stochastic optimization algorithms is done via the use of control variates (Le Roux et al., 2012). Broadly speaking these methods work by using the previous gradients in SGD $\{\nabla\ell^{\omega_1}(w^1), \dots, \nabla\ell^{\omega_t}(w^{(t)})\}$ to compute some surrogate for the current full gradient $\nabla\ell(w^{(t)})$ and compute the descent direction using both this surrogate full gradient and the standard SGD gradient.

Why do we use Nesterov’s method to train deep networks? It is worthwhile to think why we use Nesterov’s momentum to train deep networks: (i) we know that momentum does not help speed up training, and (ii) momentum is simply a faster way to minimize the same objective ℓ so it does not have any regularization properties that help generalization either. We do not have a definitive answer to this question yet but here is what we know.

Datasets that we use in deep learning represent quite narrow distributions (natural images of animals, household objects etc.). For instance, the two images below are essentially the same in spite of belonging to different classes.



Most weights of a deep network will have a similar gradient for these images as input, the weights for which the gradient will differ are likely to be the weights at the top few layers of the network. This entails that even if the stochastic gradients are computed on different mini-batches, they are essentially quite similar to each other, and thereby to the full-gradient. More precisely, the covariance of mini-batch gradients

$$\text{cov}(\nabla\ell_{\delta}(w), \nabla\ell_{\delta'}(w)) = \mathbb{E}_{\delta, \delta'} \left[(\nabla\ell_{\delta}(w) - \nabla\ell(w)) (\nabla\ell_{\delta'}(w) - \nabla\ell(w))^{\top} \right]$$

is a matrix with very few non-zero eigenvalues; only about 0.5% of the eigenvalues are non-zero (Chaudhari and Soatto, 2017) even for large networks. This means that the SGD gradient while training deep networks is essentially the full gradient and we should expect momentum to accelerate convergence in practice.

Line-search is not common in SGD Note that the function ℓ in machine learning is a summation over a large number of samples in the training dataset; in this case computing the gradient $\ell(w)$ involves a large sum as well

$$\nabla\ell(w) = \frac{1}{n} \sum_{i=1}^n \nabla\ell(w; x^i, y^i);$$

stochastic gradient descent computes this sum only over a subset of the data. Solving the line-search problem involves computing more gradients with respect to η which adds to the computational cost. Further, the

1 gradient $\widehat{\nabla}\ell(w)$ is only a noisy estimate of $\nabla\ell(w)$ so it wouldn't improve
2 progress much if we carefully picked the step-size.

3 11.4 The Adam optimizer

4 We will next look at a very popular optimizer for deep networks called
5 Adam. This is extremely popular and is the default optimization algorithm
6 that most people use when they build a network for a new problem. It
7 combines two key ideas from optimization, namely control variates and
8 second order updates. These ideas are very effective at accelerating
9 stochastic gradient descent but implementing them for deep networks
10 is very expensive. The key reason behind the success of Adam and its
11 widespread adoption is that the authors used the right approximations and
12 implemented them in incredibly clever ways.

Adam: A method for stochastic optimization

[DP Kingma, J Ba](#) - arXiv preprint arXiv:1412.6980, 2014 - arxiv.org

... stochastic **optimization** methods. Finally, we discuss AdaMax, a variant of **Adam** based
on ... We propose **Adam**, a method for efficient stochastic **optimization** that only requires first-order ...
☆ Save 📄 Cite Cited by 160431 Related articles All 27 versions ☞

13

14 Control variates

15 We will first discuss a technique called “control variate” from Monte Carlo
16 methods that can be used to accelerate stochastic gradient descent. The
17 key idea behind this technique is to observe that if we want to accelerate
18 the convergence rate of SGD, our goal should be to compute a more
19 accurate gradient (i.e., something close to the non-stochastic gradient)
20 using work that is not too much more than that required to compute the
21 stochastic gradient.

22 Say we have a random variable X and we would like to guess its
23 expected value $\mu = E[X]$. Note that X is an unbiased estimator of μ but
24 it might have a large variance. If we have another random variable Y with
25 known expected value $E[Y]$, then

$$\hat{X} = X + c(Y - E[Y]) \quad (11.16)$$

26 is also an unbiased estimator for μ for any value of c . The variance of \hat{X} is

$$\text{Var}(\hat{X}) = \text{Var}(X) + c^2 \text{Var}(Y) + 2c \text{Cov}(X, Y).$$

27 which is minimized for

$$c^* = -\frac{\text{Cov}(X, Y)}{\text{Var}(Y)}$$

28 to

$$\begin{aligned} \text{Var}(\hat{X}) &= \text{Var}(X) - c^{*2} \text{Var}(Y) \\ &= \left[1 - \left(\frac{\text{Cov}(X, Y)}{\text{Var}(Y)} \right)^2 \right] \text{Var}(X). \end{aligned}$$

1 By subtracting $Y - E[Y]$ from our observed random variable X , we have
 2 reduced the variance of X if the correlation between X and Y is non-zero.
 3 Most importantly, note that no matter what Y we plug into the above
 4 expression, we can never increase the variance of X ; the worst that can
 5 happen is that we pick a Y that is completely uncorrelated with X and
 6 end up achieving nothing.

7 Suppose that the mini-batch size is 1. We can maintain a buffer
 8 $\{g^\omega \equiv \nabla \ell^\omega\}_{\omega=1}^n$ that contains the stochastic gradient for each sample
 9 in the training dataset. As our optimization algorithm samples different
 10 mini-batches (i.e., different samples ω for a batch-size of 1), the values
 11 in this buffer are updated after each iteration. Given this buffer, if we
 12 sampled a datum ω for the current iteration, we can update the weights
 13 using the control variate-corrected gradient

$$\lambda \nabla \ell^\omega(w) - \left(\lambda g^\omega - \frac{1}{n} \sum_{i=1}^n g^i \right). \quad (11.17)$$

14 For $\lambda = 1$, this is an unbiased estimator of $\nabla \ell^\omega(w)$. Effectively, we have
 15 used the quantity $g^\omega - \frac{1}{n} \sum_{i=1}^n g^i$ from the buffer as a control variate to
 16 reduce the variance. This algorithm is called stochastic averaged gradients
 17 (SAGA). This algorithm converges linearly, i.e., at a rate $\mathcal{O}(\log(1/\epsilon))$ for
 18 strongly convex functions. This is remarkable because it is the same as
 19 the convergence rate of gradient descent for strongly convex functions.

20 We can also consider biased control variates, after all since we do not
 21 calculate the optimal value c^* in (11.16), we do not know which way the
 22 bias-variance tradeoff for our modified stochastic gradient (with respect
 23 to its mean the true gradient) is. One particular biased control variate is
 24 popular, and incidentally that is how the idea of accelerating SGD was
 25 essentially discovered,

$$\frac{\nabla \ell^\omega(w)}{n} - \left(\frac{g^\omega}{n} - \frac{1}{n} \sum_{i=1}^n g^i \right) \quad (11.19)$$

26 which we obtained by setting $\lambda = 1/n$ in (11.17). You see that the
 27 expected value of the term in the round brackets is no longer zero and
 28 therefore this is a biased control variate. But this algorithm, named as
 29 stochastic averaged gradients (SAG) was also shown to give a convergence
 30 rate of $\mathcal{O}(\log(1/\epsilon))$ for strongly convex functions, so it works pretty well.

Now notice that if we replace the value g^ω by the latest calculated

▲ There is also a non-buffer version of the SAGA algorithm called SVRG (stochastic variance reduced gradient). At some infrequent instants during training, we calculate the full gradient $\nabla \ell(w^{(s)})$ and use this to reduce the variance of the stochastic gradient at later times. Again suppose that the mini-batch size is 1. At iteration t if we sampled ω , the weights are updated using the quantity

$$\nabla \ell^\omega(w^{(t)}) - \left(\nabla \ell^\omega(w^{(s)}) - \nabla \ell(w^{(s)}) \right); \quad (11.18)$$

note that the round bracket has zero expected value. In other words, we store some checkpoint $w^{(s)}$ and calculate the stochastic gradient using the latest datum ω using the checkpoint as well as the current weights $w^{(t)}$. SVRG therefore requires calculating two stochastic gradients at each iteration. But it avoids maintaining a large buffer like that of SAGA and also converges linearly.

gradient $\nabla \ell^\omega(w)$ in our buffer to obtain a new buffer $\{g^i\}_{i=1}^n$, then

$$\frac{\nabla \ell^\omega(w)}{n} - \left(\frac{g^\omega}{n} - \frac{1}{n} \sum_{i=1}^n g^i \right) = \frac{1}{n} \sum_{i=1}^n g^i. \quad (11.20)$$

In other words, the SAG gradient is the average of the updated buffer. Note that the different gradients in the buffer were calculated using different weights over the course of past iterations, so their average is not equal to the full gradient.

This is the first big idea implemented in Adam: to average the past stochastic gradients. Instead of the Euclidean average as done in (11.20), Adam computes an exponential average. We can implement better control variates and use the SAGA gradient but Adam does not need to maintain a buffer. Adam has been shown to obtain a convergence rate of only $1/\epsilon^2$ for convex functions.

1 Pre-conditioning the gradient

2 Newton's method for solving a minimizing problem $\operatorname{argmin} \ell(w)$ corre-
3 sponds to weight updates

$$w^{(t+1)} = w^{(t)} - \left(\nabla^2 \ell(w^{(t)}) \right)^{-1} \nabla \ell(w^{(t)}). \quad (11.21)$$

4 Newton's method converges quadratically, i.e., $\ell(w^{(t+1)}) - \ell(w^*) \leq$
5 $c (\ell(w^{(t)}) - \ell(w^*))^2$, the sub-optimality shrinks very quickly. It is how-
6 ever very difficult to implement Newton's method for large optimization
7 problems, such as deep networks, because the Hessian $\nabla^2 \ell(w) \in \mathbb{R}^{p \times p}$ is
8 a very large matrix. Inverting such large matrices at each step of descent
9 is very expensive and furthermore, for many real problems the Hessian
10 is extremely ill-conditioned $\lambda_{\max}/\lambda_{\min} \approx 10^8$; in fact for deep networks
11 which have two times as many weights as the number of weights the
12 Hessian always has zero eigenvalues. In general, we can use any other
13 pre-conditioning matrix in place of the inverse Hessian so long as it is
14 positive semi-definite

$$w^{(t+1)} = w^{(t)} - \eta G^{-1/2} \nabla \ell(w^{(t)}); \quad (11.22)$$

15 the square root is taken to emphasize the positive definiteness. We can
16 approximate the Hessian using the so-called Gauss-Newton matrix:

$$\nabla^2 \ell(w) \approx \frac{1}{n} \sum_{i=1}^n \nabla \ell^i(w) \nabla \ell^i(w)^\top \quad (11.23)$$

17 if the gradients $\nabla \ell^i(w) \approx 0$. We will derive this in Chapter 14 in (14.4).
18 In other words, the Hessian, up to an approximation that holds towards
19 the end of training, is equal to the average of the outer products of the
20 individual sample gradients. This is still a very large matrix, so let's use

▲ One can implement Newton's method for some small-scale neural networks using Hessian-vector products.

1 only the diagonal and set the off-diagonal elements to zero:

$$\sqrt{G_{ii}} \equiv (\nabla^2 \ell(w))_{ii} = \frac{1}{n} \sum_{i=1}^n \left(\frac{d\ell^i(w)}{dw_i} \right)^2,$$

2 i.e., the diagonal elements of the Hessian under this approximation are the
3 averages of the squares of the gradients of the individual samples. We can
4 now write the pre-conditioned descent direction as

$$\left(G^{-1/2} \nabla \ell(w^{(t)}) \right)_i \approx \frac{\frac{d\ell(w^{(t)})}{dw_i}}{\sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{d\ell^i(w)}{dw_i} \right)^2}}. \quad (11.24)$$

5 When implementing this descent direction using stochastic gradients, we
6 can calculate the denominator using a run-time average of the square of
7 the gradients

$$\frac{1}{t} \sum_{s=1}^t \nabla_i \ell^{\omega_s}(w^{(s)})$$

Adam implements a pre-conditioned version of the gradient descent equation using a (square root of the) diagonal approximation of the Hessian as the pre-conditioning matrix. The new descent direction is (11.24) where again the denominator is calculated using exponential averaging instead of the Euclidean average.

The updates of Adam combine control variates and pre-conditioning and are given by

$$\begin{aligned} m^{(t+1)} &= (1 - \beta_1^t)^{-1} \left(\beta_1 m^{(t)} + (1 - \beta_1) \nabla \ell^{\omega_t}(w^{(t)}) \right) \\ v^{(t+1)} &= (1 - \beta_2^t)^{-1} \left(\beta_2 v^{(t)} + (1 - \beta_2) \text{diag} \left(\nabla \ell^{\omega_t}(w^{(t)}) \nabla^\top \ell^{\omega_t}(w^{(t)}) \right) \right) \\ w^{(t+1)} &= w^{(t)} - \eta \frac{m^{(t+1)}}{\sqrt{v^{(t+1)} + \epsilon}}. \end{aligned} \quad (11.25)$$

8 The factors of $(1 - \beta_1^t)^{-1}$ come from debiasing the exponential average,
9 and all three equations are computed element-wise. The default values of
10 parameters are $\beta_1 = 0.99$ and $\beta_2 = 0.999$; tweaking β_2 is usually quite
11 useful in practice. It is useful to remember that running Adam requires
12 three times the amount of memory to store the weights as that of SGD.

13 11.5 Understanding SGD as a Markov Chain

14 The preceding development tells us how SGD works and how many
15 iterations of SGD we need to get within an ϵ -neighborhood of the global

1 minimum for convex functions. Things are not this easy to understand for
 2 non-convex functions; essentially if we have two minima u^*, v^*

$$\nabla\ell(u^*) = \nabla\ell(v^*) = 0$$

3 depending upon where GD/SGD are initialized they can converge to
 4 different places. In this section, we will look at an alternative way of un-
 5 derstanding how SGD works for non-convex functions. The development
 6 here will be much more abstract than the preceding section because we
 7 want to capture the overall properties of SGD.

8 11.5.1 Gradient flow

9 Let us first talk about gradient descent. Just like we constructed a model
 10 for Nesterov's updates using a differential equation, we will first construct
 11 a model for gradient descent using a differential equation. The updates
 12 are given by

$$w^{(t+1)} - w^{(t)} = -\eta\nabla\ell(w^{(t)}).$$

13 If we again imagine a continuously differentiable curve $W(\tau)$ as a model
 14 for these discrete-time updates and time

$$d\tau := \eta$$

15 we can write a differential equation of the form

$$\frac{dW}{d\tau} = \dot{W}(\tau) = -\nabla\ell(W(\tau)); \quad W(0) = w^0. \quad (11.26)$$

16 This is called gradient flow. If we wanted to execute gradient flow on a
 17 computer, we can do so using Euler discretization

$$\dot{W}(\tau) \approx \frac{W(\tau + \Delta\tau) - W(\tau)}{\Delta\tau} = -\nabla\ell(W(\tau)).$$

18 for any value of the time-step $\Delta\tau$. If the time-step $\Delta\tau = \eta$ we get exactly
 19 gradient descent. More precisely, gradient flow is the limit of gradient
 20 descent as the learning rate $\eta \rightarrow 0$. It is important to always remember
 21 that gradient flow is a model for GD, not GD itself. Our goal in the
 22 remainder of the section is to develop a similar model for SGD.

23 11.5.2 Markov chains

24 Consider the Whack-The-Mole game: a mole has burrowed a network of
 25 three holes w^1, w^2, w^3 into the ground. It keeps going in and out of the
 26 holes and we are interested in finding which hole it will show up next so
 27 that we can give it a nice whack.

▲ A non-convex function with two local minima. The one on the left is the global minimum but gradient descent may not always reach here.

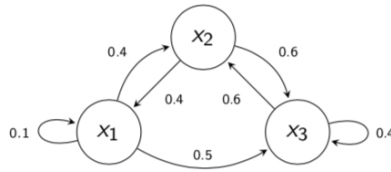


- Three holes:

$$X = \{x_1, x_2, x_3\}.$$

- Transition probabilities:

$$T = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.4 & 0 & 0.6 \\ 0 & 0.6 & 0.4 \end{bmatrix}$$



1

2 This is an example of a Markov chain. There is a transition matrix P
 3 which determines the probability P_{ij} of the mole resurfacing on a given
 4 hole w^j given that it resurfaced at hole w^i the last time. The matrix P^t is
 5 the t -step transition matrix

$$P_{ij}^t = \mathbb{P}(w^{(t)} = w^j \mid w^{(0)} = w^i).$$

6 If there exist times t, t' such the both the probabilities

$$\mathbb{P}(w^{(t)} = w^j \mid w^{(0)} = w^i) \quad \mathbb{P}(w^{(t')} = w^i \mid w^{(0)} = w^j)$$

7 are non-zero the two states w^i and w^j are said to “communicate”

$$w^i \leftrightarrow w^j$$

8 The set of states in the Markov chain that *all* communicate with each
 9 other are an equivalence class. This means that the Markov chain can
 10 visit any state from any other state in this equivalence class with non-zero
 11 probability, we just might have to wait for a really long time if $P_{ij}^t \approx 0$
 12 for two states w^i, w^j . If all the states in the Markov chain belong to
 13 the same equivalence class, it is called irreducible. A related concept
 14 is that of “positive recurrence”, i.e., if the Markov chain was at a state
 15 w at some time, it comes back to the same state after some finite time.
 16 Since the process is Markov it forgets that is just came back to the same
 17 state and therefore positive recurrence also means that if we consider an
 18 infinitely long trajectory of a Markov chain, the chain visits the same state
 19 infinitely many times along this trajectory. You can see the animations at
 20 <https://setosa.io/ev/markov-chains> to build more intuition.

21 **Invariant distribution of a Markov chain** The probability of being in
 22 a state w^i at time $t + 1$ can be written as

$$\mathbb{P}(w^{(t+1)} = w^i) = \sum_{j=1}^N \mathbb{P}(w^{(t+1)} = w^i \mid w^{(t)} = w^j) \mathbb{P}(w^{(t)} = w^j).$$

23 This equation governs how the probabilities $\mathbb{P}(w^{(t)} = w^i)$ change with
 24 time t . Let’s do the calculations for the Whack-The-Mole example. Say
 25 the mole was at hole w^1 at the beginning. So the probability distribution

1 of its presence

$$\pi^{(t)} = \begin{bmatrix} \mathbf{P}(w^{(t)} = w^1) \\ \mathbf{P}(w^{(t)} = w^2) \\ \mathbf{P}(w^{(t)} = w^3) \end{bmatrix}$$

2 is such that

$$\pi^1 = [1, 0, 0]^\top.$$

3 We can now write the above formula as

$$\pi^{(t+1)} = P^\top \pi^{(t)}$$

4 and compute the distribution $\pi^{(t)}$ for all times

$$\begin{aligned} \pi^2 &= P^\top \pi^1 = [0.1, 0.4, 0.5]^\top; \\ \pi^3 &= P^\top \pi^2 = [0.17, 0.34, 0.49]^\top; \\ \pi^4 &= P^\top \pi^3 = [0.153, 0.362, 0.485]^\top; \\ &\vdots \\ \pi^\infty &= \lim_{t \rightarrow \infty} P^t \pi^1 \\ &= [0.158, 0.355, 0.487]^\top. \end{aligned}$$

5 If such a distribution π^∞ exists, the Markov chain is said to have “equilib-
6 riated” or reached an invariant distribution. The numbers $\mathbf{P}(w^{(t+1)} = w^i)$
7 stop changing with time. We can compute this invariant distribution by
8 writing

$$\pi^\infty = P^\top \pi^\infty.$$

9 Does such a limiting invariant distribution π^∞ always exist? It turns out
10 that if a Markov chain has a finite number of states then the invariant
11 distribution π^∞ always exists; this is easy to show yourself. If the
12 Markov chain is irreducible and aperiodic, then the invariant distribution
13 is also unique. We can also compute the π^∞ given a transition matrix
14 P : the invariant distribution is the (right-)eigenvector of the matrix P^\top
15 corresponding to the eigenvalue 1.

16 **Periodicity of a Markov chain** A state of a Markov chain is periodic
17 with period k if the probability of coming back to the same state is zero
18 for times that *are not* integral multiples of k and the probability of coming
19 back to the same state is non-zero for all times that *are* integral multiples
20 of k . To take a simple example, every number of a clock is a periodic
21 state; the Markov chain comes back to that state at regular intervals. If we
22 cannot find such a time k for a given state, then the state is aperiodic. It is
23 easy to see that if there exists an aperiodic state in one communicating
24 class, then all the other states in that class also have to be aperiodic. It is
25 useful to remember that if a particular state has a non-zero probability of
26 self-transition, then the state is aperiodic.

27 **Example 11.5.** Consider a Markov chain on two states where the transition

1 matrix is given by

$$P = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \end{bmatrix}.$$

2 This is an irreducible Markov chain because you can hop between any two
 3 states with non-zero probability within one step. It is also recurrent: this
 4 is intuitive because say the Markov chain was in state 1, it is easy for it
 5 to come back to this state after a few hops. After the chain comes back
 6 to state 1, the Markov property means the chain forgets all the past steps
 7 and will again come back to state 1. The expected number of times the
 8 Markov chain comes back to state 1 is infinite. Each of the two states has
 9 a non-zero probability of self-transition, so both of them are aperiodic.

10 We are therefore guaranteed that a unique invariant distribution exists
 11 for this Markov chain. In this case it is

$$\begin{aligned} \pi^1 &= 0.5\pi^1 + 0.4\pi^2 \\ \pi^2 &= 0.5\pi^1 + 0.6\pi^2. \end{aligned}$$

12 Note that the constraint for π being a probability distribution, i.e., $\pi^1 + \pi^2 =$
 13 1 is automatically satisfied by the two equations. We can solve for π^1, π^2
 14 to get

$$\pi^1 = 4/9 \quad \pi^2 = 5/9.$$

15 **Time spent at a particular state by the Markov chain** We can observe
 16 a long trajectory of a Markov chain and compute the number of times the
 17 chain is in a particular state w^i . This is directly proportional to $\pi^\infty(w^i)$.
 18 In other words, if the invariant distribution gives small probability to a
 19 particular state, if we stop the Markov chain at an arbitrary time during its
 20 trajectory, we are very unlikely to find the Markov chain at this state.

21 11.5.3 A Markov chain model of SGD

22 The updates of SGD with mini-batch size ℓ are given by

$$w^{(t+1)} - w^{(t)} = -\eta \nabla \ell_\ell(w^{(t)}).$$

23 Notice that conditional on the iterate $w^{(t)}$, the next iterate $w^{(t+1)}$ is
 24 independent of $w^{(t-1)}$, all these three quantities are random variables
 25 because they depend on the input data $\omega_0, \dots, \omega_t$ sampled by SGD in the
 26 previous time-steps. You should never make the mistake of saying that
 27 gradient descent is a Markov chain; there is no randomness in the iterates
 28 of GD.

29 **Transition probability of SGD** What is the transition probability

$$P(w^{(t+1)} | w^{(t)})$$

30 for SGD? If we take the conditional expectation on both sides

$$\mathbb{E} \left[w^{(t+1)} - w^{(t)} | w^{(t)} \right] = -\eta \mathbb{E} \left[\nabla \ell(w^{(t)}) \right] = -\eta \nabla \ell(w^{(t)});$$

1 in other words, on-average the change in weights at $w^{(t)}$ is proportional
 2 to the full gradient $\nabla\ell(w^{(t)})$. Notice that the change in weights exactly
 3 the same for GD; this should not be surprising after all, if the gradient of
 4 SGD is unbiased then SGD is GD “on-average”.

5 **Variance of SGD weight updates** The variance is computed as follows

$$\begin{aligned}\text{Var}_{\mathfrak{b}}\left(w^{(t+1)} - w^{(t)} \mid w^{(t)}\right) &= \eta^2 \text{Var}_{\mathfrak{b}}\left(\nabla\ell_{\mathfrak{b}}(w^{(t)}) \mid w^{(t)}\right) \\ &= \eta^2 \mathbb{E}_{\mathfrak{b}}\left[\left(\nabla\ell_{\mathfrak{b}}(w^{(t)}) - \nabla\ell(w^{(t)})\right)\left(\nabla\ell_{\mathfrak{b}}(w^{(t)}) - \nabla\ell(w^{(t)})\right)^{\top}\right]\end{aligned}$$

6 Notice that the variance of the weight updates in SGD is proportional
 7 to the square of the learning rate. We have seen this before, larger the
 8 learning rate more noisy the weight update as compared to the update
 9 using the full-gradient $\eta\nabla\ell(w^{(t)})$. The variance is a large matrix $\in \mathbb{R}^{p \times p}$;
 10 this matrix depends on the current weight $w^{(t)}$.

11 If we are sampling the data inside a mini-batch with replacement the
 12 stochastic gradients are independent for different samples ω^1 and ω^2 in
 13 the mini-batch

$$\nabla\ell^{\omega^1}(w) \perp\!\!\!\perp \nabla\ell^{\omega^2}(w).$$

14 In other words

$$\mathbb{E}_{\omega^1, \omega^2} \left[\left(\nabla\ell^{\omega^1}(w^{(t)}) - \nabla\ell(w^{(t)}) \right) \left(\nabla\ell^{\omega^2}(w^{(t)}) - \nabla\ell(w^{(t)}) \right)^{\top} \right] = 0.$$

15 You can use this to show that

$$\begin{aligned}\text{Var}_{\mathfrak{b}}\left(w^{(t+1)} - w^{(t)} \mid w^{(t)}\right) &= \eta^2 \text{Var}_{\omega^1, \dots, \omega^{\mathfrak{b}}}\left(\frac{1}{\mathfrak{b}} \sum_{i=1}^{\mathfrak{b}} \nabla\ell^{\omega^i}(w^{(t)})\right) \\ &= \frac{\eta^2}{\mathfrak{b}^2} \sum_{i=1}^{\mathfrak{b}} \text{Var}_{\omega^i}\left(\nabla\ell^{\omega^i}(w^{(t)})\right) \\ &= \frac{\eta^2}{\mathfrak{b}} \text{Var}_{\omega}\left(\nabla\ell^{\omega}(w^{(t)})\right).\end{aligned}\tag{11.27}$$

16 The last step follows because we are sampling inputs ω^i uniformly randomly
 17 and therefore gradients $\nabla\ell^{\omega^i}(w^{(t)})$ are not just independent but also
 18 identically distributed. In other words, a mini-batch size of \mathfrak{b} reduces the
 19 variance by a factor of \mathfrak{b} .

20 **SGD is like GD with Gaussian noise** We now *model* the transition
 21 probability $\text{P}(w^{(t+1)} \mid w^{(t)})$ as a Gaussian distribution. Let us denote by
 22 W^t, W^{t+1} etc. the updates of this model. We now have

$$W^{(t+1)} = W^{(t)} + \xi^t$$

1 where ξ^t is Gaussian noise

$$\xi^t \sim N\left(-\eta \nabla \ell(w^{(t)}), \frac{\eta^2}{\beta} \text{Var}_\omega\left(\nabla \ell^\omega(w^{(t)})\right)\right).$$

2 In other words, on-average SGD updates weights like gradient descent, by
3 a term $-\eta \nabla \ell(w^{(t)})$ but SGD's updates also have a variance.

4 Such equations are called stochastic difference equations and they
5 are quite difficult to understand compared to non-stochastic difference
6 equations (what we see in gradient descent). So we will make a drastic
7 simplification in our model. We will say that the variance of the mini-batch
8 gradients is identity. Our model for SGD is

$$W^{(t+1)} = W^{(t)} - \eta \nabla \ell(W^{(t)}) + \frac{\eta}{\sqrt{\beta}} \xi^t \quad (11.28)$$

9 where we have zero-mean unit-variance Gaussian noise $\xi^t \sim N(0, I_{p \times p})$.

10 **Remark 11.6.** The above model for SGD is a Markov chain except that
11 the states in the Markov chain is infinite; the number of states in the
12 Whack-The-Mole example were finite. It is easy to see that the above
13 model is not exactly SGD: (i) we assumed the the transition probability
14 was a Gaussian which need not be the case while training a deep network,
15 (ii) we further assumed that the Gaussian noise does not depend on $w^{(t)}$
16 and has identity covariance. You can implement the above model on a
17 computer, first you compute the *full gradient* $\nabla \ell(w^{(t)})$ and then sample
18 Gaussian noise ξ^t to update the weights to $W^{(t+1)}$. This is obviously not
19 equivalent to SGD which updates weights using the stochastic gradient
20 $\nabla \ell_\beta(w^{(t)})$.

21 11.5.4 The Gibbs distribution

22 In a Markov chain we were interested in the invariant distribution because
23 that gives us a way to understand where the chain spends most of its time.
24 We can compute the invariant distribution for our model of SGD. It is
25 a very powerful result (which we will not do) and leads to the so-called
26 Gibbs distribution. The probability density of the invariant distribution is
27 given by

$$\rho^\infty(w) = \frac{1}{Z(\beta)} e^{-\beta \ell(w)}. \quad (11.29)$$

28 The quantity

$$\beta = \frac{2\beta}{\eta} \quad (11.30)$$

29 and $Z(\beta)$ is a normalizing constant for probability density

$$Z(\beta) = \int_{\mathbb{R}^p} e^{-\beta \ell(w)} dw.$$

30 Let us list a few properties of the Gibbs distribution that are apparent
31 simply by looking at the above expression.

- 1 1. The invariant distribution is reached asymptotically and is the
 2 limiting distribution of the weights. For instance the sum of the
 3 weights along an infinitely long trajectory converges to the mean of
 4 the Gibbs distribution

$$\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=1}^T W^t = \int_w w \rho^\infty(w) dw. \quad (11.31)$$

5 Similarly, the second moment of the weights along a long trajectory
 6 of SGD converges to the second moment of the Gibbs distribution;
 7 and same for the variance.

$$\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t'=1}^T \sum_{t=1}^T (W^{t'}) (W^t)^\top = \int_{w,w'} w w'^\top \rho^\infty(w) \rho^\infty(w') dw dw'. \quad (11.32)$$

- 8 2. The probability that the iterates of SGD are found at a location w
 9 is proportional to $e^{-\beta \ell(w)}$. If the training loss $\ell(w)$ is high, this
 10 probability is low and if the training loss is low, the probability is
 11 high. The Gibbs distribution therefore shows that if we let SGD run
 12 until it equilibrates we have a high chance of finding the iterates
 13 that have a small training loss. This observation is powerful because
 14 it does not require us to assume that $\ell(w)$ is convex. However this
 15 statement does require the assumption that the steps-size η of SGD
 16 does not go to zero; after all SGD iterates *stop* if $\eta = 0$.
- 17 3. The quantity $1/\beta$ is quite common in physics where it is called the
 18 “temperature”. This temperature $\beta^{-1} = \frac{\eta}{2\ell}$ fundamentally governs
 19 how the Gibbs distribution looks. Higher the temperature, more
 20 the noise in the iterates and vice-versa. If the learning rate η is
 21 large or the batch-size ℓ is small, it is easy for *our model of SGD*
 22 to jump over hills. This is the reason why the Gibbs distribution will
 23 be spread around the entire domain at high temperature. On the
 24 other hand, if temperature is very small, the Gibbs distribution puts
 25 a large probability mass in places where the training loss is small
 26 and the probability of finding iterates at other places in the domain
 27 diminishes. In particular, if $\beta \rightarrow \infty$, the Gibbs distribution only
 28 puts non-zero probability on the global minima of the loss function
 29 $\ell(w)$.
- 30 4. Written in another way, if we want the Gibbs distribution to remain
 31 the same we should ensure that

$$\beta^{-1} = \frac{\eta}{2\ell} \text{ is a constant.}$$

32 If you increased the batch-size by two times, you should also
 33 double the learning rate if you desire that the solutions of SGD are
 34 qualitatively similar.

- 35 5. We have achieved something remarkable by looking at the Gibbs
 36 distribution. We have discovered an algorithm to find the global

1 minimum of a non-convex loss function.

- 2 • Start from some initial condition w^0 ;
- 3 • Take lots of steps of SGD with learning rate η until SGD
- 4 reaches its invariant distribution, i.e., until it equilibrates;
- 5 • Reduce the step-size η and repeat the previous step

6 This is only a formal algorithm but in theory it will converge to the
7 global minimum of a non-convex function $\ell(w)$ if the number of
8 steps is very large. The catch of course is that at each step we have
9 to wait until SGD equilibrates. For many problems, it may take an
10 inordinately long amount of time for SGD to equilibrate.

It is very important to remember that when we train a deep network we are executing one run of SGD. The invariant distribution is an abstract concept that does not really exist on your computer. We have constructed this model to help us understand how updates of SGD behave.

🔗 How much time does it take SGD to equilibrate for a convex loss function?

11 11.5.5 Convergence of a Markov chain to its invariant distribution

13 For gradient descent and SGD, we had quantities like $\|w^{(t)} - w^*\|$ or
14 $\ell(w^{(t)}) - \ell(w^*)$ that let us measure the progress towards the global
15 minimum. For a non-convex problem, there may not exist a unique global
16 minimum, or there may be multiple local minima in the domain where the
17 gradient vanishes. We discussed in the preceding section how the invariant
18 distribution of SGD is achieved even if the loss $\ell(w)$ is non-convex. In
19 this section, we will see a simple tool to measure progress towards this
20 distribution.

21 Let us define a quantity called the Kullback-Leibler (KL) divergence
22 between two probability distributions. For two probability distributions
23 $p(w)$ and $q(w)$ supported on a discrete set $w \in W$, the KL-divergence is
24 given by

$$\text{KL}(p \parallel q) = \sum_{w \in W} p(w) \log \frac{p(w)}{q(w)}. \quad (11.33)$$

25 This formula is well-defined only if for all w where $q(w) = 0$, we also
26 have $p(w) = 0$. The KL-divergence is a measure of the distance between
27 two distances, it is zero if and only if $p(w) = q(w)$ for all $w \in W$. It
28 is always positive (you can show this easily using Jensen's inequality).
29 However, the KL-divergence is not a metric because it is not symmetric

$$\text{KL}(p \parallel q) \neq \text{KL}(q \parallel p) = \sum_{w \in W} q(w) \log \frac{q(w)}{p(w)}.$$

1 For probability densities, the KL-divergence

$$\text{KL}(p \parallel q) = \int_w p(w) \log \frac{p(w)}{q(w)} dw \quad (11.34)$$

2 is defined analogously and has the same properties.

3 We will now show a very powerful result: the KL-divergence of
 4 the state distribution of a Markov chain decreases monotonically as the
 5 Markov chain converges to its invariant distribution. Although, this result
 6 is true for SGD as well, we will only prove it for a Markov chain with finite
 7 states. Let the initial distribution of the Markov chain be π^0 , its transition
 8 matrix be P and its invariant distribution be π^∞ . We will assume that the
 9 Markov chain is such that the invariant distribution exists (it is irreducible
 10 and recurrent).

11 Let us also assume that a reverse transition matrix

$$P_{ij}^{\text{rev}} = \mathbb{P}(w^{(t)} = w^i | w^{(t+1)} = w^j),$$

12 exists; such Markov chains are called reversible. For any states w, w' this
 13 transition matrix satisfies the definition of conditional probability

$$\mathbb{P}(w^{(t+1)} = w' | w^{(t)} = w) \mathbb{P}(w^{(t)} = w) = \mathbb{P}(w^{(t)} = w | w^{(t+1)} = w') \mathbb{P}(w^{(t+1)} = w').$$

14 In our notation, this becomes

$$P_{ww'}^{\text{rev}} = \frac{P_{w'w} \pi(w')}{\pi(w)} = \frac{P_{w'w} \pi(w')}{\sum_{w'} P_{w'w} \pi(w')}.$$

15 **Lemma 11.7.** For a reversible Markov chain with an invariant distribution
 16 π^∞ , the quantity $\text{KL}(\pi^\infty \parallel \pi^t)$ decreases monotonically:

$$\text{KL}(\pi^\infty \parallel \pi^{t+1}) \leq \text{KL}(\pi^\infty \parallel \pi^t). \quad (11.35)$$

1 **Proof.** The proof is a simple calculation.

$$\begin{aligned}
\text{KL}(\pi^\infty \parallel \pi^{t+1}) &= \sum_w \pi^\infty(w) \log \frac{\pi^\infty(w)}{\pi^{t+1}(w)} \\
&= \sum_w \pi^\infty(w) \log \frac{\pi^\infty(w)}{\sum_{w'} P_{w'w} \pi^t(w')} \\
&= - \sum_w \pi^\infty(w) \log \frac{\sum_{w'} P_{w'w} \pi^t(w')}{\pi^\infty(w)} \\
&= - \sum_w \pi^\infty(w) \log \left(\sum_{w'} P_{ww'}^{\text{rev}} \frac{\pi^t(w')}{\pi^\infty(w')} \right) \quad (\text{substitute definition of } P^{\text{rev}} \text{ for distribution } \pi^\infty) \\
&\leq - \sum_w \pi^\infty(w) \sum_{w'} P_{ww'}^{\text{rev}} \log \frac{\pi^t(w')}{\pi^\infty(w')} \quad (\text{Jensen's inequality}) \\
&= \sum_{w'} \sum_x P_{ww'}^{\text{rev}} \pi^\infty(w) \log \frac{\pi^\infty(w')}{\pi^t(w')} \quad (\text{flip the negative sign, exchange sum}) \\
&= \sum_{w'} \pi^\infty(w') \log \frac{\pi^\infty(w')}{\pi^t(w')} \\
&= \text{KL}(\pi^\infty \parallel \pi^t).
\end{aligned}$$

2 The distance to the invariant distribution π^∞ decreases at each step of the
3 Markov chain. A similar statement is true for the reverse KL divergence:

$$\text{KL}(\pi^{t+1} \parallel \pi^\infty) \leq \text{KL}(\pi^t \parallel \pi^\infty).$$

4

□

5 The above result is also true for SGD which, as we discussed, can
6 be modeled as a Markov chain with infinite states. It gives us some
7 very important intuition. Just like gradient descent makes monotonic
8 progress towards the global minimum w^* , a Markov chain (or SGD) makes
9 monotonic progress towards its invariant distribution. The big difference
10 between them is that while we required that the loss function $\ell(w)$ is
11 convex for gradient descent to guarantee this monotonic progress, the loss
12 need not be convex for the case of the Markov chain model of SGD.

13 This result *does not* mean that SGD makes monotonic progress towards
14 the global minimum $w^* = \text{argmin}_w \ell(w)$. We choose to look at SGD not
15 as one particle undergoing (stochastic) gradient descent updates but rather
16 as a Markov chain. The probability distribution of states of this Markov
17 chain is then a legitimate object (the distribution π^t is the distribution of
18 weights W^t obtained after many independent run of SGD from different
19 initializations). Although π^t is *not* meaningful across *one* run of SGD,
20 we can use it to get an abstract understanding of how SGD also makes
21 monotonic progress as it converges if we imagine many *independent* runs
22 of SGD occurring simultaneously.

Chapter 12

Shape of the energy landscape of neural networks

Reading

1. Goodfellow Chapter 13
2. “Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima” by [Baldi and Hornik \(1989\)](#)
3. “Entropy-SGD: Biasing gradient descent into wide valleys” by [Chaudhari et al. \(2016\)](#)

In this chapter, we will try to understand the shape of the objective for training neural networks. We would like to characterize the difficulty of training neural networks. We know that the objective is not convex and training a network is difficult because of it. But how non-convex is the objective? The questions we want to answer here are of the following form.

1. How many global minima exist?
2. How many local minima and saddle points exist?
3. What is the loss at the local minima or saddle points?

If we train with gradient descent or stochastic gradient descent, what loss can we expect to obtain even if we don't reach the global minimum?

4. What is the local geometry of the loss function?
5. What is the global topology of the loss function?

This will help understand how SGD seems to train deep networks so efficiently and why we often get very good generalization error after training. As a pre-cursor to how the picture of the energy landscape of a neural network looks like, here's one picture from [Li et al. \(2018\)](#):

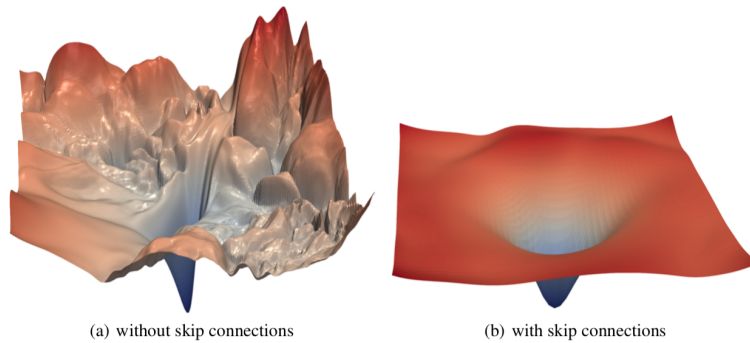


Figure 12.1: A picture of the training loss. The picture on the left was created by sampling two directions randomly out of the millions of weights for a residual network without skip-connections and computing the training loss by discretization of this two-dimensional space. The picture on the right is a similar picture for the resnet with skip-connections intact. In this picture, we see that while the training loss is very complex on the left-hand side with lots of local minima and saddle points, the loss is much more benign on the right-hand side.

12.1 Introduction

Let us introduce a few quantities that will help characterize the energy landscape. We will consider the case when the function $\ell(w)$ is twice-differentiable.

Global minima are all points in the set

$$\{w : \ell(w) \leq \ell(w') \text{ for all } w'\}.$$

Note that there may exist many different locations all with the same loss $\ell(w)$, they would all be global minima in this case. Local minima are all points in the set

$$\{w : \nabla \ell(w) = 0, \nabla^2 \ell(w) \succeq 0\}.$$

i.e., all points w where the Hessian $\nabla^2 \ell(w)$ is positive semi-definite. Note that the two conditions (i) first-order stationarity $\nabla \ell(w) = 0$ and (ii) positive semi-definiteness of the Hessian $\nabla^2 \ell(w) \succeq 0$ also have to be satisfied for all global minima. Critical points are all locations which satisfy only first order stationarity

$$\{w : \nabla \ell(w) = 0\}.$$

Saddle points are critical points but which are neither local minima nor local maxima

$$\{w : \nabla \ell(w) = 0, \nabla^2 \ell(w) \text{ is neither positive nor negative semi-definite}\}.$$

Non-convex functions, in general, can have all these different kinds of locations in the energy landscape and this makes minimizing the objective

🔍 Draw the Gibbs distribution of SGD if $\ell(w)$ has multiple global minima.

🔍 Draw the Gibbs distribution of SGD if $\ell(w)$ has multiple global minima and multiple local minima.

1 difficult. Our goal in this chapter is to learn theoretical and empirical
 2 results that help paint a mental picture of what the energy landscape looks
 3 like.

4 12.2 Deep Linear Networks

5 Let us consider the simplest case of linear neural networks first. We will
 6 have a two-layer neural network which takes in inputs x^i and aims to
 7 predict targets y^i . For simplicity, we will consider the case when both

$$x^i, y^i \in \mathbb{R}^d.$$

8 and use the regression loss

$$\ell(A, B) = \frac{1}{2n} \sum_{i=1}^n \|y^i - AB x^i\|_2^2 \quad (12.1)$$

9 We use the standard trick of appending a 1 to the input x^i so that we don't
 10 have to carry around biases in our equations.

11 The matrices A, B are the weights of the neural network with

$$A \in \mathbb{R}^{d \times p}, B \in \mathbb{R}^{p \times d}.$$

12 We will consider the case when $p \leq d$. We are interested in finding A and
 13 B and will develop some results from Baldi & Hornik's paper.

14 **Least squares solution** A simple calculation reveals that for a single-
 15 layer network the solution of the problem

$$L^* = \operatorname{argmin}_L \frac{1}{2n} \sum_{i=1}^n \|y^i - Lx^i\|_2^2$$

16 is

$$L^* = \Sigma_{yx} \Sigma_{xx}^{-1} \quad (12.2)$$

17 where

$$\begin{aligned} \Sigma_{yx} &= \sum_i y^i x^{i\top} \\ \Sigma_{xx} &= \sum_i x^i x^{i\top}. \end{aligned}$$

18 The matrices Σ_{yx} and Σ_{xx} are the data covariance matrices.

19 **Projection of a vector onto a matrix** It will be useful to define a
 20 projection matrix. Say we have a vector v that we want to project on the
 21 span of the columns of a full-rank matrix

$$M = [m_1 \quad m_2 \quad \dots \quad m_n].$$

1 If this projection is $\hat{v} \in \text{span}\{m_1, \dots, m_n\}$, we know that it has to satisfy

$$(v - \hat{v}) \perp m_k \text{ for all } k \leq n \quad \Rightarrow \quad m_k^\top (v - \hat{v}) = 0 \text{ for all } k \leq n.$$

2 The vector \hat{v} is also obtained by a combination of the columns of M , so
3 there exists a vector c which allows us to write

$$\hat{v} = Mc.$$

4 These together imply

$$c = (M^\top M)^{-1} M^\top \hat{v}$$

5 and finally

$$\hat{v} = \underbrace{M(M^\top M)^{-1} M^\top}_{\text{projection matrix}} v \\ =: P_M v.$$

6 where the matrix P_M is called the projection matrix corresponding to the
7 matrix M .

8 **Back to deep linear networks** We know from the homework problem
9 that there is no unique solution to the problem

$$A^*, B^* = \underset{A, B}{\operatorname{argmin}} \frac{1}{2n} \sum_{i=1}^n \|y^i - AB x^i\|_2^2.$$

10 If A^*, B^* are solutions, so are $A^* P, P^{-1} B^*$ for any invertible matrix P .
11 We also showed in the homework that the objective is not convex. But if
12 we fix either A or B and only optimize over the other, the loss is convex.
13 Notice that the rank of AB is at most p .

14 **Fact 12.1 (Critical points of B if A is fixed).** For any A , the function
15 $\ell(A, B)$ is convex in B and has a minimum at

$$(A^\top A) \hat{B}(A) \Sigma_{xx} = A^\top \Sigma_{yx}.$$

16 If Σ_{xx} is invertible and A is full-rank, then we can write

$$\hat{B}(A) = (A^\top A)^{-1} A^\top \Sigma_{yx} \Sigma_{xx}^{-1}. \quad (12.3)$$

17 Note that these are all locations when the gradient

$$\frac{\partial \ell}{\partial B} = 0.$$

18 **Fact 12.2 (Critical points of A if B is fixed).** We have an analogous
19 version of the previous fact for A : if B is fixed, the loss is convex in A ,
20 for full-rank Σ_{xx} and B , then for $\frac{\partial \ell}{\partial A} = 0$, we should have

$$AB \Sigma_{xx} B^\top = \Sigma_{yx} B^\top. \quad (12.4)$$

▲ Note that $P_M^2 = P_M$, i.e., if we project the vector twice onto the column space of M , the second projection does nothing. Also, any projection matrix P is symmetric. To see this, consider two vectors v, w and the dot products

$$\langle Pv, w \rangle, \text{ and } \langle v, Pw \rangle.$$

In both cases, one of the vectors lies completely in the column space of M and therefore the dot product ignores any component that is orthogonal to the column space of M . This means

$$\langle Pv, w \rangle = \langle v, Pw \rangle = \langle Pv, Pw \rangle.$$

We can now rewrite the first equality to obtain

$$(Pv)^\top w = v^\top (Pw) \\ \Rightarrow v^\top P^\top w = v^\top Pw$$

and since this is true for any two vectors v, w , we have that $P = P^\top$.

1 Or in other words,

$$\hat{A}(B) = \Sigma_{yx} B^\top (B \Sigma_{xx} B^\top)^{-1}. \quad (12.5)$$

2 **Fact 12.3 (Critical points of (A, B)).** We now solve the equations (12.3)
3 and (12.5) to get a critical point, i.e., the gradient of the objective in both
4 A and B is zero. First

$$W = AB = P_A \Sigma_{yx} \Sigma_{xx}^{-1}. \quad (12.6)$$

5 from (12.3). Next, multiply (12.4) on the right by A^\top to get

$$W \Sigma_{xx} = W^\top AB \Sigma_{xx} B^\top A^\top = \Sigma_{yx} B^\top A^\top = \Sigma_{yx} W^\top.$$

6 Now we substitute the value of W from (12.6) to get the condition that A
7 should satisfy

$$P_A \Sigma = \Sigma P_A = P_A \Sigma P_A.$$

8 where

$$\Sigma = \Sigma_{yx} \Sigma_{xx}^{-1} \Sigma_{xy}.$$

9 **Fact 12.4 (If W is a critical point, then it can be written as a projection**
10 **of the least squares solution $\Sigma_{yx} \Sigma_{xx}^{-1}$ on the subspace spanned by**
11 **some p eigenvectors of Σ).** This is an important fact. Let us say we have
12 a full-rank Σ with distinct eigenvalues $\lambda_1 > \dots > \lambda_d$. Let u_{i_k} be the
13 eigenvector associated with the i_k^{th} eigenvalue of Σ , i.e.,

$$\mathbb{R}^{d \times d} \Sigma = U \Lambda U^\top.$$

14 Given any index set of p eigenvalues

$$\mathcal{I} = \{i_1, \dots, i_p\} \text{ with } 1 \leq i_k \leq d \text{ for all } k.$$

15 we can define a matrix of rank p

$$U_{\mathcal{I}} = [u_{i_1} \quad u_{i_2} \quad \dots \quad u_{i_p}]$$

16 formed by the orthonormal eigenvectors of Σ associated with the eigen-
17 values $\lambda_{i_1}, \dots, \lambda_{i_p}$ of the index set.

18 One can then show that the matrices A and B are critical points if and
19 only if there is a set \mathcal{I} and an invertible matrix $C \in \mathbb{R}^{p \times p}$ such that

$$\begin{aligned} A &= U_{\mathcal{I}} C \\ B &= C^{-1} U_{\mathcal{I}}^\top \Sigma_{yx} \Sigma_{xx}^{-1}. \end{aligned} \quad (12.7)$$

20 You can find the proof in the Appendix of Baldi & Hornik's paper. Because
21 $U_{\mathcal{I}}$ is a matrix of orthonormal vectors we also have

$$P_{U_{\mathcal{I}}} = U_{\mathcal{I}} U_{\mathcal{I}}^\top$$

▲ Proving (12.4) is slightly involved and you can read the Appendix of the original paper for the proof. It relies upon a clever rewriting of the regression objective using the identity

$$\text{vec}(PQR^\top) = (R \otimes P) \text{vec}(Q)$$

where the Kronecker product of two matrices $R \otimes P$ is obtained by replacing each entry R_{ij} of the matrix R by the matrix $R_{ij}P$. Using this, we can write our original objective in (12.1) as

$$\begin{aligned} & \frac{1}{2n} \sum_i \|y^i - ABx^i\|^2 \\ &= \frac{1}{2n} \|\text{vec } Y - (X^\top B^\top \otimes I) \text{vec } A\|^2 \end{aligned}$$

where X is a matrix with x^i as the i^{th} column. Now we can use our standard formula for the solution of linear regression to solve for the vector $\text{vec } A$ in terms of the other known quantities.

1 and therefore

$$W = P_{U_{\mathcal{I}}} \Sigma_{yx} \Sigma_{xx}^{-1}$$

2 which is the same form for W as (12.6) in Fact 3 and L^* in (12.2). In
3 other words, the solution $W = AB$ in a two-layer linear network is given
4 by our original least squares regression matrix followed by an orthogonal
5 projection onto the subspace spanned by p eigenvectors of Σ .

6 **Fact 12.5 (If W is the global minimum for a two-layer network, then
7 it is a projection of the solution for a single-layer network onto the
8 subspace spanned by the top p eigenvectors of Σ).** You can further
9 show that the objective

$$\ell(A, B) = \text{trace}(\Sigma_{yy}) - \sum_{i_k \in \mathcal{I}} \lambda_{i_k}. \quad (12.8)$$

10 at a critical point (A, B) . The first term is a constant with respect to
11 the parameters of the network A, B . The second term is a sum of the
12 eigenvalues of the matrix Σ at indices that we picked in our set $U_{\mathcal{I}}$. What is
13 the index set that minimizes this loss? It is simply the largest p eigenvalues
14 of Σ . This is also a unique value for the loss because we have assumed
15 that all the eigenvalues are distinct. This also solidifies the connection
16 of this model with Principal Component Analysis (PCA), the matrix W
17 is projecting on the sub-space spanned by the top p eigenvectors in the
18 auto-associative case.

19 **Fact 12.6 (There are exponentially many saddle points for a two-layer
20 network).** There are a total of $\binom{d}{p}$ possible index sets \mathcal{I} . One of them as
21 we saw above corresponds to a global minimum. It can be shown that all
22 the others are saddle points. Note that there are exponentially many saddle
23 points. This is an important fact to remember: there are exponentially
24 many saddle points in a hierarchical architecture.

25 Smaller the number of neurons in the hidden layer p (also the upper
26 bound for the rank of the weight matrices), fewer are the number of saddle
27 points but this also creates a dimensionality bottleneck in the feature space.
28 If p is too small as compared to d we lose large amounts of information
29 necessary to classify the image and the network need not work well.

30 **Fact 12.7 (There are no local minima in a deep linear network; all
31 minima are global minima).** The proof of Fact 12.6 also shows that any
32 index set $\mathcal{I} \neq \{1, \dots, p\}$ cannot be a local minimum (see the Appendix
33 of the paper). There are no local minima for a deep linear network, only
34 global minima and saddle points. This is often summarized as “linear
35 networks have no bad local minima”.

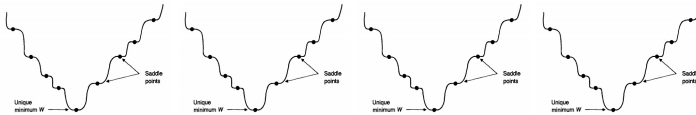
36 **Fact 12.8 (The global minimum is not unique).** This is perhaps the most
37 important point of this chapter. The loss at the global minimum is unique,
38 not the global minimum itself. Any full-rank square matrix $C \in \mathbb{R}^{p \times p}$ of
39 our choice gives a pair of solutions (A, B) . How many such solutions are
40 there? There are lots and lots of such solutions, in fact, given any solution
41 with a particular C if we can perturb the C without losing rank (quite easy

🔗 Based on the previous two facts, what can you say about the solution W if $p \geq d$ and Σ is invertible? Since the two-layer network simply projects on the p eigenvalues of Σ , if $p \geq d$ and Σ is invertible, the solution already lies in the column-space of Σ and therefore $W = L^*$.

1 to do by, say, changing the eigenvalues slightly) we get another solution of
2 a linear network.

3 **Fact 12.9 (All the previous results are true for multi-layer linear**
4 **networks).** The same results are true for deep linear networks (Kawaguchi,
5 2016). These results also hold if $\dim(y_i) = 1$, i.e., for the regression case.

We used a simple two-layer linear network to obtain an essentially complete understanding of how the loss function looks like. A schematic looks as follows.



There are lots of locations where the global minimum of the function is achieved. There are lots of saddle points in the energy landscape. The Gibbs distribution for this energy landscape has a lot of modes, one each at the global minima.

6 How does weight-decay

$$\Omega(A, B) = \lambda \left(\|A\|_F^2 + \|B\|_F^2 \right)$$

7 change the energy landscape of deep linear networks? It changes the
8 number of global minima, only the ones that have the smallest ℓ_2 norm
9 remain in the energy landscape. It also reduces the number of saddle
10 points because the Hessian at saddle points has an extra additive term that
11 involves λ .

12 12.3 Extending the picture to deep networks

13 Let us think carefully about the non-uniqueness of the solution for a
14 two-layer network. We know that all critical points are of the form

$$A = U_{\mathcal{I}} C,$$

$$B = C^{-1} U_{\mathcal{I}}^{\top} \Sigma_{yx} \Sigma_{xx}^{-1}.$$

15 The gradient at these critical points is zero. Given a particular C , we can
16 perturb it slightly and obtain a new critical point (a new saddle point, or a
17 new global minimum) and this keeps the objective unchanged. Effectively,
18 we have a connected set of global minima and saddle points for a deep
19 linear networks.

20 If one were to try to visualize this energy landscape and extend the
21 picture heuristically to deep networks with nonlinearities, we can think of
22 the global minimum as looking like the basin of the Colorado river.



1

2 The important point to remember from this picture is that all the points
3 at the basin of the river are solutions that obtain a good training loss.
4 Although gradient-based algorithms (GD/SGD etc.) do not allow us to
5 travel along the river (the gradient is zero along it), if the river basin snakes
6 around in the entire domain, no matter where the network is initialized,
7 we always have a global minimum close to the initialization. Essentially,
8 the objective of deep networks is not convex, but current results indicate
9 that it is quite benign. And this is perhaps the reason why it is so easy to
10 train them.

Chapter 13

Generalization performance of machine learning models

This chapter gives a preview of generalization performance of deep networks. We will take a more abstract view of learning algorithms here and focus only on binary classification. We will first introduce a “learning model”, i.e., a formal description of what learning means. The topics we will discuss stem from the work of two people: [Leslie Valiant](#) who developed the most popular learning model called Probably Approximately Correct Learning (PAC-learning) and [Vladimir Vapnik](#) who is a Russian statistician who developed a theory (called the VC-theory) that provided a definitive answer on the class of hypotheses that were learnable under the PAC model.

13.1 The PAC-Learning model

Our goal in machine learning is to use the training data in order to construct a model that generalizes well, i.e., has good performance outside of the training data. Formally, we search over a hypothesis space \mathcal{F} , e.g., a specific neural net architecture, using the available data to find a good hypothesis $f \in \mathcal{F}$. As we motivated in Chapter 2, without further assumptions, we cannot guarantee that this hypothesis works well on test data. We therefore assume two things in this chapter:

1. Nature provides independent and identically distributed samples $x \in \mathcal{X}$ from some (unknown to the learner) distribution P .
2. Nature labels these samples with $c(x)$ which is again unknown to the learner.

Both training and test data are samples from Nature’s distribution P . We will also assume that even if the true labeler $c(x)$ is unknown to us, we know that it belongs to a chosen hypothesis class $c(x) \in \mathcal{C}$ and is

deterministic, i.e., Bayes error is zero. Changing this assumption does not change the crux of this theory.

Consider a learning algorithm, denoted by L . Given a dataset $D = \{(x^i, c(x^i))\}_{i=1}^n$ and a hypothesis class \mathcal{C} , the population risk (for classification) of the hypothesis output by this learning algorithm is

$$R(f) = \mathbb{E}_{x \sim P} [\mathbf{1}_{\{f(x) \neq c(x)\}}]$$

Let us assume that the learning algorithm is deterministic for now, i.e., given a training dataset D it returns a unique answer f . Let us assume that the hypothesis class that the learner searches over, named \mathcal{F} is the same as the hypothesis class \mathcal{C} . What do we want from this algorithm?

We expect that it works well for all hypotheses Nature could use to label data $c \in \mathcal{C}$ and all datasets D drawn from P . The PAC-Learning model postulates the following desiderata upon the learning algorithm.

1. We are okay with an answer f with error

$$R(f) \in [0, 1/2)$$

because we only have access to finitely many training data. This is the “approximately correct” part of the PAC-Learning. The error should decrease as n increases.

2. The dataset D is a random variable. This implies that the hypothesis output by the learning algorithm $f(D)$ is also a random variable. The above statement therefore should hold with some large probability over draws of the dataset D . In other words, there can be a small probability that a non-representative dataset D is drawn and we do not expect the learner to output a good hypothesis with $R(f) < 1/2$. However the probability of such failure, let us call it $\delta \in [0, 1/2)$, should also become smaller if more data is provided. This is the “probably” part of PAC-Learning.

We can now use these two postulates to give a definition of what it means to be a good learning algorithm.

Definition 13.1 (PAC-learnable hypothesis class). A hypothesis class \mathcal{C} is PAC-learnable if there exists an algorithm L such that for every true labeling function $c \in \mathcal{C}$, for every $\epsilon, \delta \in [0, 1/2)$, if L is given access to $n(\epsilon, \delta)$ i.i.d. training data from P and their corresponding labels c then it outputs a hypothesis $h_D \in \mathcal{C}$ such that

$$\mathbb{P}_D (R(f) < \epsilon) \geq 1 - \delta.$$

We want the learner to be statistically efficient, i.e., as our desiderata ϵ, δ get smaller, we should expect $n(\epsilon, \delta)$ to not grow too quickly. One classical setting under which we analyze learning is the case when $n(\epsilon, \delta)$ is a polynomial function of $1/\epsilon$ and $1/\delta$.

Sample complexity and computational complexity The minimum number of samples $n(\epsilon, \delta)$ required to learn a hypothesis class \mathcal{C} is

1 called the sample complexity of \mathcal{C} . One is also typically interested in
 2 the computational complexity of finding f , e.g., to avoid a brute-force
 3 algorithm L that searches over the entire hypothesis class $\mathcal{F} = \mathcal{C}$; we will
 4 not worry about it here.

It is important to notice that PAC-learning assumes nothing about *how* the learner L is going to use the data D to create a hypothesis $f(D)$, e.g., whether it runs SGD or some variant, or what surrogate loss it uses, or even whether it performs Empirical Risk Minimization. In this sense, the above learning model is very abstract and we should expect only qualitative answers from this theory.

5 **Example 13.2 (Learning Monotone Boolean Formulae).** Let $x =$
 6 $[x_1, \dots, x_d] \in \{0, 1\}^d$ be a datum and let the true label $c(x)$ be the
 7 conjunction of the entries of x , e.g.,

$$c(x) = x_1 \wedge x_3 \wedge x_4.$$

8 To take a few examples, $c(10011) = 0$ and $c(11110) = 1$. Such formulae
 9 are called monotone because no literals show up as negated in the formula.

10 We can have the hypothesis class \mathcal{F} to be the set of all possible
 11 conjunctions of d variables x_1, \dots, x_d . Each literal x_i can be in the
 12 conjunction or not, so the total number of hypotheses in \mathcal{F} is 2^d .¹
 13 Observe that since this is exponential in d , an algorithm L that brute-force
 14 searches over \mathcal{F} will have a large computational complexity. Also observe
 15 that since the true hypothesis $c \in \mathcal{F}$, there exists an answer f that the
 16 algorithm L can output that achieves zero training error, i.e.,

$$\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{\{f(x^i) \neq c(x^i)\}} = 0.$$

17 But for a fixed amount of data n , there is some probability that the
 18 minimizing hypothesis f has zero training error but large population risk.
 19 As the number of data n is large, we expect this event to be less and less
 20 probable.

21 Consider an algorithm L that does the following. It starts with the
 22 hypothesis

$$f^0(x) = x_1 \wedge x_2 \wedge \dots \wedge x_d$$

23 with all literals and for every datum with a label 1, it deletes all literals x_i
 24 from the hypothesis f^0 that were not a part of that datum; this makes sense
 25 because if the deleted literals were zero in some input, f and c would
 26 predict different outputs. Remember that since $c(x) \in \mathcal{F}$, we cannot have
 27 a datum with input 1111...1 and output 0.

¹Actually the total number of conjunctions is $2^d + 1$ because for the null-conjunction (without any literals) we can have the constant $c(x) = 0$ or $c(x) = 1$ for all x . We should therefore explicitly make sure $c(111\dots 11) = 0$ is not in the true labeling function. But we ignore this corner case, and silently assume that only the hypothesis $c(x) = 1 \forall x$ is in our class \mathcal{C} .

1 What kind of errors does this algorithm make? If some literal x_i was
 2 deleted, it is because it had the value $x_i = 0$ on a positively labeled sample.
 3 So we only output a wrong hypothesis if more literals are present in our
 4 hypothesis than those in $c(x)$. If we think about this carefully, the output
 5 $f(x)$ can only make an error on data that are labeled 1 by $c(x)$, never on
 6 the ones labeled zero. Our algorithm therefore only has false negatives.

7 We now see why having more samples diminishes the probability of
 8 this event happening. Let $p_i = \mathbb{P}_{x \sim P} [c(x) = 1, x_i = 0 \text{ in } x]$. Therefore

$$R(h) \leq \sum_{x_i \in f} p_i$$

9 If some p_i is small, then it does not contribute much to the error. If
 10 some p_i is large then we make sure to see enough samples so that we
 11 remove that x_i from f . After all, it only takes one appearance of this
 12 event to delete this x_i , and the event has probability p_i which is large.
 13 Rigorously, if all x_i in f have $p_i < \epsilon/d$ then $R(h) < \epsilon$. On the other
 14 hand, if some x_i has $p_i > \epsilon/d$ then the probability of having this x_i
 15 in f is the probability that the event of p_i never happens in the draw
 16 of n samples. But this new probability is smaller than $1 - \epsilon/d$. And
 17 the event will never happen in n i.i.d. draws with probability at most
 18 $(1 - \epsilon/d)^n \leq e^{-n\epsilon/d}$. Using the union bound, since there are at most d
 19 literals in f , the probability that there is at least one such “bad event” is at
 20 most $de^{-n\epsilon/d}$.

21 If this bad event never happens the population risk is less than ϵ . Of
 22 course, such a bad event happening would be devastating. For some
 23 distributions it could lead the error up to 1. However, in our PAC-learning
 24 setting we can accept this as long as it happens rarely with probability at
 25 most δ . And therefore we can say that if

$$de^{-n\epsilon/d} < \delta \Rightarrow n \geq d\epsilon^{-1} \log(d/\delta)$$

26 then we are guaranteed to meet the PAC criteria: of error less than ϵ with
 27 probability at least $1 - \delta$.

28 Note that both the sample complexity and computational complexity
 29 are polynomial in this example. We have thus shown that the class
 30 of Monotone Boolean Formulae is (ϵ, δ) -PAC learnable. The sample
 31 complexity n is linear in the number of dimensions d of the data.

32 13.2 Concentration of Measure

33 Two very important results from probability theory that we will use are
 34 the Union Bound and the Chernoff Bound.

13.2.1 Union Bound (or Boole's Inequality)

For any countable set of events, $\{A_1, \dots, A_n, \dots\}$,

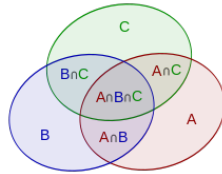
$$P\left(\bigcup_i A_i\right) \leq \sum_i P[A_i].$$

This is a rather loose, but useful, upper bound and is (mostly) embedded in the assumptions of what we call a "probability measure" in probability theory (σ -subadditivity). This essentially means that it can be used without any extra assumptions in practice.

By the inclusion-exclusion principle for finite set of events $\{A_1, \dots, A_n\}$,

$$P\left(\bigcup_{i=1}^n A_i\right) = \sum_{i=1}^n P(A_i) - \sum_{1 \leq i < j \leq n} P(A_i, A_j) + \dots + (-1)^{n-1} P(A_1, A_2, \dots, A_n)$$

We can get better approximations of the union, if we use the first $k \leq n$ terms above. If we stop at odd k , we get an upper bound. If we stop at even k we get a lower bound. The error of the approximation is decreasing with k .



▲ If we want a better approximation of the probability of the union of multiple events and we know more about the problem at hand we can use what are called Bonferroni inequalities.

❓ Where did we use the union bound in the proof for the PAC-learnability of the class of monotone Boolean functions?

❓ Try to prove that

$$P\left(\bigcap_{i=1}^n A_i\right) \geq 1 - \sum_{i=1}^n P(A_i^c)$$

13.2.2 Chernoff Bound

Let A_1, \dots, A_n be a sequence of i.i.d. random variables. We focus on the case of Bernoulli random variables where $P(A_i = 1) = p$. We would like to estimate p from samples. One way to do this is to compute the empirical average

$$\hat{p}(n) = \frac{1}{n} \sum_{i=1}^n A_i$$

and estimate how close it is to the true p . We know that as $n \rightarrow \infty$

Weak Law For all $\epsilon > 0$ we have

$$\lim_{n \rightarrow \infty} P(|\hat{p}(n) - p| \leq \epsilon) = 1.$$

This is also known as convergence in probability.

Strong Law In this case, we also have almost sure convergence, i.e.,

$$P\left(\lim_{n \rightarrow \infty} \hat{p}(n) = p\right) = 1.$$

1 **Central Limit Theorem** As $n \rightarrow \infty$, the quantity $\sqrt{n}(\hat{p} - p)$ is
 2 distributed as a Normal distribution with mean zero and variance $p(1 - p)$.
 3 Notice that as opposed to the law of large numbers, the central limit
 4 theorem also gives us a rate of convergence, i.e., how many samples n are
 5 necessary if want the difference to be close to a Normal distribution. If
 6 we set $\sigma^2 = p(1 - p)$ we can rewrite the Central Limit Theorem as

$$P(|\hat{p}(n) - p| > \epsilon) \leq 2e^{-n\epsilon^2/(2\sigma^2)}.$$

7

8 **Chernoff Bound** Since $\sigma^2 = p(1 - p) < 1/4$ we have from CLT that

$$P\left(\left|\frac{1}{n} \sum_i A_i - p\right| > \epsilon\right) \leq 2e^{-2n\epsilon^2}.$$

9 An easy way to remember the Chernoff bound is that if we want the
 10 average of n random variables to be ϵ -close to their expected value with
 11 probability at least $1 - \delta$, then we need

$$n = \Omega(\epsilon^{-2} \log(1/\delta))$$

12 samples.

13 Concentration of measure is a beautiful area of probability theory and
 14 similar results can be obtained for other distributions, other functions than
 15 averaging of random variables A_1, \dots, A_n etc. Popular inequalities are
 16 Markov's Inequality, Chebyshev's Inequality and Chernoff Bounds (and
 17 Hoeffding's Inequality as an important special case). They are written in
 18 terms of increasing tightness, but also of increasing assumptions of what we
 19 need to know in order compute them. You can read a very good introduction
 20 to this topic at <https://terrytao.wordpress.com/2010/01/03/254a-notes-1-concentration-of-measure/>.
 21

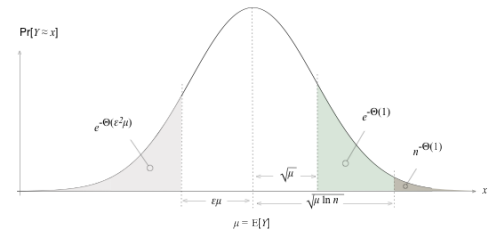
22 13.3 Uniform convergence

23 We now lift the assumption that Nature's labeling function $c \in \mathcal{C}$. After
 24 all, even if there exists such a true deterministic c we can never be sure that
 25 it is inside \mathcal{F} , say the class of neural networks of a specific architecture
 26 that we are using. This model is called the Agnostic PAC-Learning model.

27 We will stay within the confinements of Empirical Risk Minimization
 28 where we are provided with some samples where we output the hypothesis
 29 with the smallest training error

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{\{f(x^i) \neq y^i\}} \quad \text{minimizing this gives } f_{\text{ERM}} \in \mathcal{F}.$$

▲ This picture makes it easy to remember concentration inequalities for an n -dimensional Gaussian random variable Y .



❓ Do you see any patterns in the Chernoff bound with sample complexity in PAC-learning?

1 The population risk is

$$R(f) = \mathbb{E}_{(x,y) \sim P} [\mathbf{1}_{\{f_{\text{ERM}}(x) \neq y\}}] \quad \text{minimizing this gives } f^* \in \mathcal{F}.$$

2 Observe that f^* is *not* the Bayes optimal predictor that we saw in the
 3 bias-variance tradeoff. This is because we are now restricted to the
 4 hypothesis class \mathcal{F} while there was no such restriction before.

Our goal while computing a *generalization bound* is to ask the following question: if we obtain an ERM hypothesis f_{ERM} with a good training error, then does this also mean that the population risk of the best hypothesis within the class, i.e., f^* , is small?

5 The above question is central, answering it in the affirmative ensures
 6 that we are using a correct hypothesis class (say neural architecture) and
 7 that the error on the training dataset is a good indicator of the performance
 8 on the entire distribution. This involves the following two steps.

9 1. First, we need to make sure that the difference

$$\left| \hat{R}(f_{\text{ERM}}) - R(f_{\text{ERM}}) \right| \rightarrow 0, \quad n \rightarrow \infty.$$

10 This is easy, it is akin to the concentration of measure we saw in the
 11 previous section.

12 2. Second, we need to ensure that

$$\hat{R}(f_{\text{ERM}}) \approx R(f^*)$$

13 with high probability for every training dataset of n samples using
 14 which f_{ERM} is computed. If this is true, it tells us something about
 15 the ERM procedure itself, i.e., it tells us whether minimizing the
 16 empirical risk $\hat{R}(f)$ is a good thing if we want to build a classifier
 17 that works well on the population.

18 This is difficult to do, after all f_{ERM} and f^* are totally different
 19 hypothesis. Vladimir Vapnik set up a powerful approach to do this.
 20 He showed that a *sufficient* condition to achieve the above is that the
 21 empirical risk and population risk are similar for all hypotheses in \mathcal{F} .

22 **This framework/assumption is known as uniform convergence.**

23 Let us now develop the two points above. Since data are drawn iid, we
 24 can use the Chernoff bound to get that

$$\forall f \in \mathcal{F}, \mathbb{P} \left(\left| \hat{R}(f) - R(f) \right| > \epsilon \right) \leq 2e^{-2n\epsilon^2}.$$

25 If the hypothesis class is finite \mathcal{F} , we can use the union bound to show

1 that for *any* hypothesis, the training error and population risk are close.

$$\begin{aligned} & \mathbf{P}\left(\exists f \in \mathcal{F} : \left|\hat{R}(f) - R(f)\right| > \epsilon\right) \\ & \leq \sum_{f \in \mathcal{F}} \mathbf{P}\left(\left|\hat{R}(f) - R(f)\right| > \epsilon\right) \\ & \leq |\mathcal{F}| 2e^{-2n\epsilon^2}. \end{aligned}$$

2 If we want this above probability of a bad event to be less than δ we
3 therefore need

$$n \geq \frac{1}{2\epsilon^2} \log \frac{2|\mathcal{F}|}{\delta} \quad (13.1)$$

4 training samples.

5 Suppose we had a classifier f with 2% gap ($\epsilon = 0.02$) between
6 the training error $\hat{R}(f)$ and the validation error (which is a proxy for
7 the population risk $R(f)$), if we want to reduce this gap by half to 1%
8 ($\epsilon = 0.01$), we need 4 times as many training data. We could also reduce
9 this gap by fitting a classifier with small $|\mathcal{F}|$ but in this case, both the
10 training and validation error might increase even if their gap decreases.

11 Next, we need a relation between the population risk of f_{ERM} and the
12 best possible predictor f^* in our hypothesis class \mathcal{F} . Observe that

$$\begin{aligned} R(f_{\text{ERM}}) & \leq \hat{R}(f_{\text{ERM}}) + \epsilon && \text{(Chernoff bound on } f_{\text{ERM}}) \\ & \leq \hat{R}(f^*) + \epsilon && (f_{\text{ERM}} \text{ has the smallest training error)} \\ & \leq R(f^*) + 2\epsilon && \text{(Chernoff bound on } f^*). \end{aligned}$$

13 The two Chernoff bound inequalities hold with probability at least $1 - \delta$
14 so the final inequality

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\epsilon$$

15 holds with probability at least $1 - 2\delta$. Substitute this in (13.1) to get

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\sqrt{\frac{1}{2n} \log \frac{4|\mathcal{F}|}{\delta}} \quad (13.2)$$

16 with probability $1 - \delta$. A result of this kind is called a Vapnik-Chernovenkis
17 (VC) bound or a PAC bound. Notice how this bound changed from the
18 Monotone Boolean function example: we need $\mathcal{O}(1/\epsilon)$ times more
19 samples in (13.1).

20 Let us consider our monotone Boolean formulae example again. Since
21 $|\mathcal{F}| = 2^d$, if the input dimension is $d = 1000$ and we set $\delta = 10^{-3}$, the
22 VC-bound predicts the following (we should imagine running ERM to
23 pick the best hypothesis f_{ERM} , not the elimination algorithm we discussed
24 in the section on monotone Boolean formulae):

25 1. With $n = 1000$ data, we have $R(f_{\text{ERM}}) \leq R(f^*) + 1.42$. This is
26 vacuous/non-informative since the population risk is an expectation
27 of indicator variables and should therefore be less than 1.

28 2. With $n = 10^5$, we have $R(f_{\text{ERM}}) \leq R(f^*) + 0.45$. This is

informative: it means that the population risk of the classifier obtained by ERM is within 44% of the population risk of the best classifier f^* in that class. Of course it is only meaningful if f^* generalizes well, i.e., if $R(f^*)$ is small. This will happen if the hypothesis class \mathcal{F} is large enough.

3. With $n = 10^6$, we have $R(f_{\text{ERM}}) \leq R(f^*) + 0.04$.

13.4 Vapnik-Chervonenkis (VC) dimension

In the above section, the concept/hypothesis class was assumed to be finite $|\mathcal{C}| < \infty$. The union bound of course breaks if this is not the case. Notice that once we pick a neural architecture (hypothesis class), the number of possible models (hypotheses), each with different weight vectors, is infinite. Observe that in the monotone Boolean formulae example, the algorithm L was using the training data to eliminate hypothesis from \mathcal{C} , this is not going to work if \mathcal{C} is not finite. It is therefore a natural question whether we can still learn a hypothesis class with infinitely-many candidate hypotheses with a finite number of training data.

Vladimir Vapnik and Alexey Chervonenkis (Vapnik, 2013) developed the so-called VC-theory to answer the above question. Technically, VC-theory transcends PAC-Learning but we will discuss only one aspect of it within the confinements of the PAC framework. VC-theory assigns a “complexity” to each hypothesis $f \in \mathcal{C}$.

Shattering a set of inputs We say that the set of inputs $D = \{x^1, \dots, x^n\}$ is *shattered* by the concept class \mathcal{C} , if we can achieve every possible labeling out of the 2^n labellings using some concept $c \in \mathcal{C}$. The size of the largest set D that can be shattered by \mathcal{C} is called the VC-dimension of the class \mathcal{C} . It is a measure of the complexity/expressiveness of the class; it counts how many different classifiers the class can express.

If we find a configuration of n inputs such that when we assign *any* labels to these data, we can still find a hypothesis in \mathcal{C} that can realize this labeling, then

$$\text{VC}(\mathcal{C}) \geq n.$$

On the other hand, if for every possible configuration of $n + 1$ inputs, we can always find a labeling such that no hypothesis in \mathcal{C} can realize this labeling, then

$$n \leq \text{VC}(\mathcal{C}).$$

If we find some n for which both of the above statements are true, then

$$\text{VC}(\mathcal{C}) = n.$$

Some examples.

- d -dim Linear Threshold Functions: $\text{VC-dim} = d + 1$.

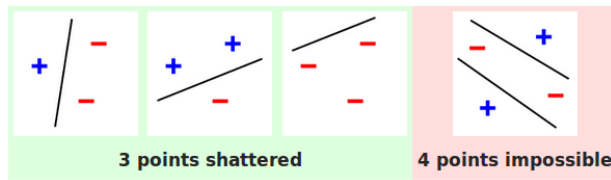


Figure 13.1: $d=2$: See that for the lower bound, we found some configuration of the 3 points, such that a linear threshold function always separates the points consistently with the labels; for any possible labeling. 3 such labellings are shown, convince yourselves that it can be done for all 8 cases. Observe that we cannot do the same for 4 points. In the figure above one such unrealizable configuration is given (With the “XOR” labeling). To prove the upper bound we need to talk about ANY configuration though. See that the only other case for 4 points, is that one point is inside the convex hull generated from the other 3. Find the labeling that cannot be obtained with linear classifiers in this case.

- 1 • 2 dimensional axis aligned rectangles: VC-dim = 4 (exercise)
- 2 • Monotone Boolean Formulae: VC-dim = d (exercise).
- 3 • If the hypothesis class is finite, then

$$\text{VC}(\mathcal{F}) \leq \log |\mathcal{F}|.$$

- 4 • If $x \in \mathbb{R}$ and our concept class includes classifiers of the form

$$\text{sign}(\sin(wx))$$

- 5 where w is a learned parameter, then

$$\text{VC} = \infty.$$

- 6 • For a neural network with p weights and sign activation function

$$\text{VC} = \mathcal{O}(p \log p).$$

It is a deep result that if the VC-dimension of concept class is finite $V = \text{VC}(\mathcal{F}) < \infty$, then this class has the uniform convergence property (for any $f \in \mathcal{F}$, the empirical and population error are close). Therefore, we can learn this concept class agnostically (without worrying about whether Nature’s labeling function c is in our hypothesis class \mathcal{F} or not) in the PAC framework with

$$n = \Omega \left(\frac{V}{\epsilon^2} \log \frac{V}{\epsilon} + \frac{1}{\epsilon^2} \log \frac{1}{\delta} \right)$$

training data. If a hypothesis class has infinite VC-dimension, then it is not PAC-learnable and it also does not have the uniform convergence property.

The above result written in another form looks as follows. For a (finite or infinite) hypothesis class \mathcal{F} with finite VC-dimension $V = \text{VC}(\mathcal{F})$

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\sqrt{\frac{1}{n} (2V - \log \delta)} \quad (13.3)$$

with probability at least $1 - \delta$. This is an important expression to remember: the number of samples n required to learn a concept class scales linearly with the VC-dimension V . A more refined version of this same bound looks like

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\sqrt{\frac{1}{n} \left(V \left(\log \frac{2n}{V} + 1 \right) + \log \frac{4}{\delta} \right)}; \quad (13.4)$$

but such expressions should essentially be understood to be saying the same thing, namely that the number of samples required to learn scales linearly with the VC-dimension.

Bounds on the VC-dimension of deep neural networks For general classifiers, it is typically difficult to compute the VC dimension. One instead finds upper and lower bounds for the VC dimension to be used in inequalities of the form (13.4). Bounds on the VC-dimension of deep network architectures are available (Bartlett et al., 2019). With p weights and L layers, an essentially tight VC-dimension looks like

$$\Omega \left(p L \log \frac{p}{L} \right) = \text{VC}(\mathcal{F}) = \mathcal{O}(p L \log p)$$

for deep networks with ReLU nonlinearities.

This bound is not entirely useful in the VC-theory however. For instance, the ALL-CNN network you used in your homework with $p \approx 10^6$ and $L = 10$ has $\text{VC} \approx 10^8$. If we use the coarse VC-bound in (13.3) with $n = 50,000$ samples, we have

$$R(f_{\text{ERM}}) \leq R(f^*) + 40$$

which is a vacuous generalization bound. However, remember that this is simply an *upper bound* on the generalization error of ERM. It is clear from empirical results in the literature (including your homework problems) that deep networks indeed generalize well to new data outside the training set and that means $R(f_{\text{ERM}})$ is small.

The gap in applying VC-theory to deep networks therefore likely stems from the need for uniform convergence: we may not need that the empirical and population risk are close for *all* hypotheses in the class. If we only have uniform convergence within a small subset $F \subset \mathcal{F}$ and if $\text{VC}(F) \ll \text{VC}(\mathcal{F})$ and if the training algorithms like SGD always find ERM minimizers $f_{\text{ERM}} \in F$, then VC-theory/PAC-Learning do predict that deep networks will generalize well. Of equivalently, instead of considering concept classes \mathcal{C} that are learnable with polynomially-many samples n , we should consider simpler concept classes that require fewer

▲ You have noticed this in one of your homework problems. When data is sampled from a small part of the domain, even if the true labeling function is very complicated, we can build a hypothesis that generalizes well. But this hypothesis may not generalize if data is sampled from outside this domain.

- ¹ samples to learn. Understanding this is the subject of a large body of
- ² ongoing research.

Chapter 14

Sloppy Models

Reading

1. Sections 1 and 2 of “Geometry of nonlinear least squares with applications to sloppy models and optimization” ([Transtrum et al., 2011](#))

In the previous chapter we have seen some classical ideas on how to capture the *size* of the hypothesis space, e.g., using a quantity called the VC-dimension. This lets us estimate the number of samples required to learn data from a given concept class. In this chapter, our goal will be to obtain an understanding of the *shape* of the hypothesis class, e.g., its geometry (which models are close by to which models and which ones are far away), its topology (are there some models that are identical to others), etc. The ideas that we are going to discuss form a part of a field called “Information Geometry” (developed by Shun’ichi Amari https://en.wikipedia.org/wiki/Shun%27ichi_Amari). It is a very rich field that combines ideas from geometry and information theory to understand learning. We will not go into a lot of mathematical details in this chapter. But you will see that these ideas give a very visual understanding of both optimization and generalization of deep networks (and also other machine learning models).

14.1 Model manifold of nonlinear regression

Consider a dataset $\{(x^i, y^{*i})\}_{i=1}^n$ where true outputs $y^{*i} \in \mathbb{R}$ and inputs $x^i \in \mathbb{R}^d$. We will fit this dataset using a nonlinear function and assume that the underlying probabilistic model is

$$y^* = f(x; w) + \xi$$

1 where $\xi = N(0, \sigma^2)$ is Gaussian noise for some scalar $\sigma > 0$. This
 2 setup is identical to the one we did in Lecture 2 for maximum likelihood
 3 regression. The weights of this model are $w \in \mathbb{R}^p$. The residual of the fit,
 4 i.e., the error incurred at each datapoint is given by

$$r_i(w) = \frac{y^{*i} - f(x^i; w)}{\sigma}. \quad (14.1)$$

5 Since all our samples are independent and identically distributed, the
 6 residuals r_i are normally distributed with zero mean and unit variance
 7 (this is because we divided by σ). In other words, the likelihood of our
 8 dataset under the model with weights w is

$$P(r_1, \dots, r_n; w) = \frac{1}{(2\pi)^{n/2}} \exp\left(-\frac{1}{2} \sum_{i=1}^n r_i(w)^2\right).$$

9 It should not be surprising at this point that it is this likelihood that we
 10 maximize when we fit a model using maximum likelihood estimation or
 11 perform nonlinear least squares regression.

12 **Data space** Let us create a shorthand for the vector of residuals of each
 13 input datum

$$\vec{r} = [r_1, r_2, \dots, r_n].$$

14 We can similarly create a short-hand for the vector of the true outputs and
 15 the predicted outputs

$$\begin{aligned} \mathbb{R}^n \ni \vec{y}^* &= [y^{*1}, y^{*2}, \dots, y^{*n}] \\ \mathbb{R}^n \ni \vec{y}(w) &= [\hat{y}^1, \hat{y}^2, \dots, \hat{y}^n]. \end{aligned}$$

16 Notice that $\vec{y}(w)$ is a function of the weights of the model. The key idea
 17 of this chapter (and information geometry) is to realize that the above
 18 quantities are simply vectors in \mathbb{R}^n . We can therefore plot them in this
 19 space and understand distances between them. For example, the “truth”
 20 \vec{y}^* can be written as

$$\vec{y}^* = \vec{y}(w) + \sigma \vec{r}.$$

21 We will give this space a name: it is called the “data space”.

22 **Model manifold** A manifold is a mathematical object which locally
 23 looks like Euclidean space. A good example to keep in mind is the surface
 24 of the Earth: at each point to us walking on it the Earth is locally flat, but
 25 the Earth can have a more complex shape than what is evident to us on it
 26 surface (a sphere). It will also be useful to keep in mind that the Earth is an
 27 object in 3-dimensional Euclidean space but it is a 2-dimensional manifold.
 28 Because of the constraint that points on the surface are equidistant from
 29 the center, every point on the surface can be described by two variables:
 30 the latitudes and longitudes.

Let M be the manifold swept by $\vec{y}(w)$ for different values of w

$$M = \{\vec{y}(w) : w \in \mathbb{R}^p\}.$$

We will give this a name: it is the “model manifold” of our model $f(\cdot; w)$. The model manifold is embedded in the data space so its dimensionality is at most n . Notice that the truth \vec{y}^* need not lie on the manifold M .

Fisher Information Metric It is useful to define a metric that helps us understand how far away two points on the manifold are. Any point that lies on the manifold also lies in the data space. And therefore we can use some reasonable way to measure distances in the data space in order to talk about the distances on the manifold. Since we are performing least squares regression, let us define squared distances between two points in the n -dimensional data space as the sum of the squares of the coordinates (ℓ_2 norm). Distance from the truth \vec{y}^* is

$$C(w) = \frac{1}{2n} \sum_i (\hat{y}^i(w) - y^{*i})^2 = \frac{1}{2} \sum_i r_i(w)^2;$$

this is the standard least squares regression objective.

Armed with this new language, we can now say: fitting a model is equivalent to finding the closest point to \vec{y}^* on the model manifold M .

For two nearby points on the manifold $\vec{y}(w)$ and $\vec{y}(w')$ with $w' = w + dw$, this corresponds to

$$\begin{aligned} d(w, w') &= \frac{1}{2n} \sum_i (\hat{y}^i(w) - \hat{y}^i(w'))^2 \\ &\approx \frac{1}{2n} \sum_i \partial_{w'} (\hat{y}^i(w) - \hat{y}^i(w'))^2 \Big|_{w'=w} dw + \frac{1}{2} dw^\top g(w) dw. \end{aligned} \quad (14.2)$$

where we took the Taylor series approximation of $(\cdot)^2$ and defined

$$\mathbb{R}^{p \times p} \ni g(w) = \frac{1}{2n} \sum_i \partial_{w'}^2 (\hat{y}^i(w) - \hat{y}^i(w'))^2 \Big|_{w'=w}. \quad (14.3)$$

Now notice that since $\hat{y}^i(w) - \hat{y}^i(w') = 0$ when $w = w'$, the first derivative term in the Taylor series is zero. We therefore have, up to second order,

$$\begin{aligned} \text{dist}(w, w + dw) &= \frac{1}{2} dw^\top g(w) dw \\ &= \frac{1}{2} dw^\top J^\top J dw. \end{aligned} \quad (14.4)$$

▲ Picture of the model manifold and data space

where the Jacobian

$$\mathbb{R}^{n \times p} \ni J_{ik} = \frac{dr_i}{dw_k}$$

is the derivative of the i^{th} sample's residual with respect to the k^{th} weight w_k in the model. **The matrix $g(w)$ is called the Fisher Information Metric (FIM)** because it gives us a way to measure the distance between two *infinitesimally nearby* points on the model manifold. It is important to realize that the FIM $g(w)$ is a function of the weights that it is calculated at. At the global minimum when $r_i \approx 0$ for all i , the FIM is equal to the Hessian (observe this in the adjoining derivation).

Remark 14.1. The distance in the data space $C(w)$ is like our standard Euclidean distance in 3-dimensional space. But we know that the shortest path between two points on the surface of the Earth is not the straight line that joins them (which does not lie on the surface but cuts through it) but instead along the great circle that joins the two points (this is the path that airplanes usually fly along, and longitudes are defined using). The great circle path is “shortest” because among all continuous paths that join two points w, w' on the surface of the Earth (not necessarily nearby), the great circle path has the shortest value of

$$\int_0^1 \sqrt{dw^\top g(w) dw}.$$

Such paths are called “geodesics”. They are the analogue of straight lines in Euclidean space for manifolds.

Optimization involves initializing the weights w at some point $w^{(0)}$ which corresponds to some point $\vec{y}(w^{(0)})$ on the model manifold M and finding the point on the manifold w^* that is closest to \vec{y}^* as measured by the cost $C(w^*) = \frac{1}{2n} \sum_i r_i(w^*)^2$. The trajectory of the weights during optimization corresponds to a trajectory on the model manifold. Among all trajectories, it would potentially help to take the shortest trajectory, shortest as measured by the FIM.

Generalization corresponds to making statements about the width of the manifold in the $(n + 1)^{\text{th}}$ dimension given a particular point $\vec{y}(w)$ in the n -dimensional manifold. If the width is small, then the model has a small variance (i.e., its predictions do not vary much on the new datum). If the true $\vec{y}^* \in \mathbb{R}^{n+1}$ is also close to $\vec{y}(w) \in \mathbb{R}^{n+1}$ then the model also has a small bias and only then it generalizes well. Instead of thinking of adding an extra dimension, we can also think of taking slices of our n -dimensional data space (i.e., projecting it into lower dimensions) and build a similar mental picture: if projecting into different (cardinal) subspaces usually ends up eliminate axes along which the manifold was thin, then the model generalizes well.

▲ Deriving (14.4) is not hard. But to make it easier, you can also imagine that \vec{y}^* lies on the manifold and we are working in a neighborhood of some weight w^* that gives \vec{y}^* . In this case,

$$\begin{aligned} (g(w))_{kl} &= \partial_{w_k} \partial_{w_l} \frac{1}{2n} \sum_i r_i^2 \\ &= \frac{1}{n} \sum_i (\partial_{w_k} r_i) \partial_{w_l} r_i \\ &\quad + \sum_i r_i \partial_{w_k} \partial_{w_l} r_i \\ &\approx \frac{1}{n} \sum_i (\partial_{w_k} r_i) \partial_{w_l} r_i \\ &= (J^\top J)_{kl}. \end{aligned}$$

The approximation follows from noticing that if we have a model that fits the data well, the residuals $r_i \approx 0$ for all samples i and therefore the second term is small.

14.2 Understanding optimization for sloppy models

The FIM is a fundamental object in the study of probability distributions. We will list some of its properties below that will shed light upon how the geometric structure of the model manifold and the FIM allows us to understand some key phenomena in optimization and generalization.

FIM does not depend upon the ground-truth labels The FIM depends upon the inputs $\{x^i\}$ and the model that fits the data $f(x; w)$. It does not depend upon the ground-truth targets. Notice that it depends upon the derivatives of the residuals in (14.3) (not the residuals...which are functions of the ground-truth labels).

FIM for classification problems Although we have defined all quantities in the case of nonlinear regression, we can define the FIM, the model manifold and all relevant quantities for any probabilistic model. For classification, if our network predicts $p_w(y | x)$ where y takes C distinct values, the FIM is defined as

$$(g(w))_{kl} = \frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C \frac{\partial_{w_k} p_w(y | x^i) \partial_{w_l} p_w(y | x^i)}{p_w(y | x^i)}.$$

Notice again that it does not depend upon the ground-truth labels. In this sense, it is very different from the Hessian of the cross-entropy loss

$$(\nabla^2 \ell(w))_{kl} = -\frac{1}{n} \sum_i \partial_{w_k} \partial_{w_l} \log p_w(y^{*i} | x^i).$$

The FIM $\in \mathbb{R}^{p \times p} g(w) = J^\top J$ is a positive semi-definite matrix. The Hessian has both positive and negative eigenvalues in general. The two are equal at the global minimum of the cross-entropy loss.

FIM characterizes how the weight space maps into the data space Since $d(w, w + dw) = \frac{1}{2} dw^\top g(w) dw$, the matrix $g(w)$ defines how the weight space gets mapped to the data space. More precisely, a unit ball in the data space centered around $\vec{y}(w)$, i.e., the set

$$A = \left\{ \vec{y}(w') : \frac{1}{2n} \|\vec{y}(w) - \vec{y}(w')\|^2 \leq 1 \right\}$$

corresponds to an ellipse

$$B = \left\{ w' : \frac{1}{2} (w' - w)^\top g(w) (w' - w) \leq 1 \right\}$$

in the weight space. The matrix $g(w)$ therefore controls how changes in the weights $w - w'$ reflect in the changes in the outputs of the model on all the samples $\vec{y}(w) - \vec{y}(w')$. Suppose we write the singular value

1 decomposition of the FIM as

$$g(w) = U\Sigma^2U^\top.$$

2 where the singular values are sorted in decreasing order of their magnitude
 3 along the diagonal of a diagonal matrix Σ^2 and columns of U are the
 4 singular vectors. Changes in weights $w - w'$ along singular vectors that
 5 have small singular values will have a small value of $(w - w')^\top g(w)(w -$
 6 $w')$ and therefore will not result in very different predictions $\vec{y}(w)$ and
 7 $\vec{y}(w')$.

8 The volume of the ellipse B in the weight space is proportional to

$$\sqrt{\det g(w)}.$$

9 For many problems, the determinant of the FIM is very very small.
 10 Such phenomena have been studied under the name “sloppy models”.
 11 We have seen one example of this phenomenon. For the 1-dimensional
 12 polynomial regression problem in the midterm exam, we were doing linear
 13 least squares regression with $y = Aw$ where

$$A = \begin{bmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^{d-1} \\ 1 & a_2 & a_2^2 & \cdots & a_2^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^{d-1} \end{bmatrix}$$

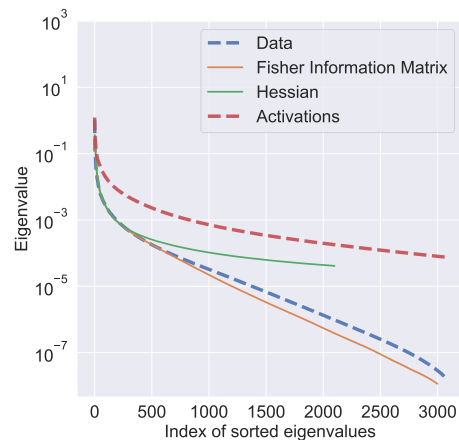
14 was the Vandermonde matrix. It is easy to check that the FIM for linear
 15 least squares problem is $g(w) = A^\top A$. The determinant of such FIMs is
 16 very small

$$\det g(w) \approx \epsilon^{n(n-1)} \approx 0 \text{ for large } n.$$

17 Here ϵ is the maximum distance between two data points in the dataset.
 18 Enormous volumes in the weight space correspond to tiny volumes in the
 19 data space for such problems.

20 Deep networks are sloppy

21



22

This pattern persists even for deep networks. For a wide residual network trained on CIFAR-10, Yang et al. (2022) computed the first 3000 eigenvalues of the FIM (orange) and compared them to the top 2000 eigenvalues of the Hessian (green) and the eigenvalues of the input correlation matrix $\frac{1}{n} \sum_i x^i x^{i\top}$. Notice that eigenvalues of the FIM drop by about 8 orders of magnitude within the first 3000 entries; the network has many more weights $p \sim 3 \times 10^6$ and it is expected that the orange curve keeps decreasing all the way to zero; we know that some eigenvalues have to be zero because the network has many more parameters than the number of data points. The FIM of this network will therefore be extremely small. All neural networks trained on typical datasets seem to have sloppy FIMs.

Optimization for sloppy models is slow because the condition number is large Notice that in the above plot, the eigenvalues of the Hessian also drop quickly (by about 4 orders of magnitude). If we take a quadratic approximation of the loss near the final point in the above figure, the objective can be written as

$$\ell'(w) = \frac{1}{2}(w - w^*)^\top \nabla_w^2 \ell(w)|_{w=w^*} (w - w^*).$$

If the Hessian is sloppy, the contours of this objective (even if it convex and quadratic) are very elongated ellipses. Roughly, the largest axis of the ellipse is about 100 times longer than the 2000th largest axis. This entails that the best learning rate along the direction of the largest axis is 100 times smaller. As we have discussed in the chapters on optimization, this makes it difficult to pick a good value for the learning rate. We know that the number of steps required even if we use the best learning rate is proportional to $\sqrt{\kappa}$, this is very large for sloppy models.

But why is optimization for deep networks so effective then? The above point is less of a problem that it may seem because the eigenvectors of $g(w)$ corresponding to the smallest eigenvalues (which are the “sloppiest” subspaces) are exactly the directions along which we need not change the weights much. These changes will not result in large changes to the predictions and therefore the loss. In the picture of the eigenvalues above, there is a very large number of such sloppy directions and a very small number of “stiff” non-sloppy directions. If the model can make accurate predictions after making progress along the stiff directions (we do see this in practice, e.g., the loss decreases very quickly in the beginning and very slowly towards the end), then we can stop at some reasonable point after training for a short duration and expect accurate predictions. This is the reason we train deep networks for so few epochs—this is not sufficient to fit the data perfectly (you will notice that the loss is never zero) but it is presumably good enough to fit most of non-sloppy dimensions well.

▲ If you think about it, you can convince yourself that lengths of the axes of the ellipse are proportional to the square root of the eigenvalues of the Hessian.

▲ A good visual description of the optimization landscape can be obtained by noticing that the ratio of the length of a human hair to its width is also about 100. So the ellipses that correspond to the quadratic objective roughly force us to travel down the length of the hair without falling off (although there are hills on the sides for our optimization problem...).

▲ You will notice that we have not characterized how the FIM/Hessian changes with weights w —which would be necessary to say things like “optimization does not change the sloppiest subspace much”. It is often the case for neural networks that the FIM/Hessian do not change much with the weights w .

14.3 Understanding generalization for sloppy models

Cramer-Rao bound Such a pathologically ill-conditioned FIM is not necessarily an issue. Whether a model fits the data depends upon whether the manifold M has a point that is close to \hat{y}^* or not; it does not depend upon the FIM. If the FIM has a very small determinant, it simply means that there are many weights in the weight space that lead to similar predictions on the samples. This is noticed all the time for large models such as deep networks, e.g., if you train the same network twice you will get a similar training and generalization error but the weights of the network will be totally different. We have seen this as there being many equivalent solutions $A^* = U_I C$ and $B^* = C^{-1} U_I^\top L^*$ for any non-singular matrix C for a two-layer linear neural network.

But there are many problems where one is interested in estimating parameters as opposed to simply making predictions using the fitted model, e.g., finding the foci of the ellipses of the orbits of the planets in the solar system using observations in the sky. If such models are sloppy, then we would not be able to estimate the parameters of the model precisely because many parameters would map to the same point in the data space. Sloppy models were discovered by the authors of the paper listed as reading material when they noticed this while fitting some models to data from biology.

A key result in statistics called the Cramer-Rao bound states that the variance of *any* unbiased estimator \hat{w} using n samples is at least as large as the inverse of the determinant of FIM

$$n \text{Cov}(\hat{w}) \succeq g^{-1}(w). \quad (14.5)$$

For sloppy models $\det g^{-1}(w) = 1/\det g(w)$ is very very large. This entails that any procedure to estimate the parameters of the model (which would be weights of the network in our case) in an unbiased way will have a huge covariance. In simple words, if the model is sloppy then accurate prediction is not necessarily hard, but parameter estimation is very hard.

PAC-bounds for sloppy models We could obtain very good generalization if the right hand-side of the Cramer-Rao bound were small—for sloppy models it is not. But we also know that we need not fit exactly the same model as the one that generated the data (Nature’s model) in order to generalize well (we would know how to check this anyway).

In [Yang et al. \(2022\)](#), it was shown that for sloppy models we can obtain generalization bounds that are not vacuous. The reason for this is as follows. Recall that when we complained about the vacuousness of generalization bounds in the previous chapter, we argued that the VC-dimension of deep networks is so large because they have a large number of parameters. For sloppy models (see the eigenspectrum in the picture above), very few combinations of the weights play a role in making predictions (these would be the number of large singular in the FIM/Hessian and the singular vectors

▲ Can you guess when the problem of finding the foci of the ellipses using observations of them in the sky will have a sloppy FIM?

1 would give the specific combinations of the weights). Therefore, even if
2 the model has a large number of weights, if the model is sloppy, its weights
3 are under-determined by the training data—the precise value of most of
4 the weights is immaterial to the model making accurate predictions. We
5 can therefore, roughly speaking, calculate the PAC generalization bounds
6 only in the non-sloppy subspace and obtain a much more accurate picture
7 of the generalization error.

Chapter 15

Variational Inference

Reading

1. Sections 1-2 of “Variational Inference: A Review for Statisticians” by [Blei et al. \(2017\)](#).
2. Sections 1-5 of “Auto-Encoding Variational Bayes” by [Kingma and Welling \(2013\)](#)
3. Chapter 2 of Durk Kingma’s thesis: <https://pure.uva.nl/ws/files/17891313/Thesis.pdf>.
4. Bishop Chapter 11.5-11.6
5. Bishop Chapter 10-10.3
6. Lots of great intuition at <http://ruishu.io/2018/03/14/vae/>

We have been primarily concerned with models for classification and regression as yet in this course. The task there is to match the target (a class identity or a real-valued outcome). We now change tracks to consider generative modeling, these are models that are trained to synthesize new data. Effectively, the task here is not *match* a target datum, but given a training dataset of images/text, create a model that outputs similar images/text at test time. We will first take a look at variational methods and generative modeling using these methods in this chapter and do implicit generative models such as Generative Adversarial Networks in the next chapter.

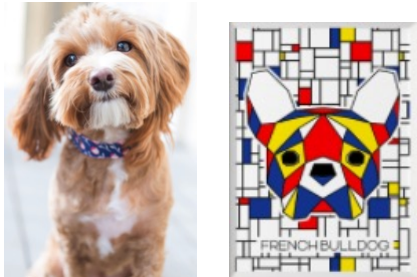
15.1 The model

Imagine how you would draw the image of a dog x on paper. First, you would decide in your mind, its breed, its age, the color of its fur etc. Let

1 us call these quantities “latent factors”. Latent factors can also include
 2 things that are not specific to the dog, e.g., the background of your painting
 3 (grass, house, beach etc.), the weather on that day (cloudy, sunny etc.), the
 4 viewpoint (zoomed in/far away). We will denote all such quantities by

$z :=$ latent factors.

5 Having decided upon all these factors, you realize your painting x . The
 6 painting x is not unique given latent factors z , e.g., two people can start
 7 off with the same latent factors and draw two totally different pictures.



8
 9 We therefore model the generative process as a obtaining samples
 10 from a probability distribution

$$p(x | z).$$

11 Given a latent factor z and an image x , the quantity $p(x | z)$ denotes the
 12 likelihood of the sample. Given the painting image x , we do not know
 13 what the latent factors are. For instance, it is not easy to say whether the
 14 following image is that of a cat or a dog.



15

In other words, the latent factors of data x are not known to us if we do not take part in the generative process. Nature is in charge of generating the data and our goal here is to guess the parameters of this generative model to be able to synthesize new samples that look as if Nature generated them.

16 There can be lots of latent factors z . So let us control this complexity
 17 and assume that we know a prior over the latent factors

$$\text{prior } p(z)$$

18 that models our belief of how likely a factor “dog with color blue” is in
 19 Nature.

Let us imagine Nature's generative model as running in two steps

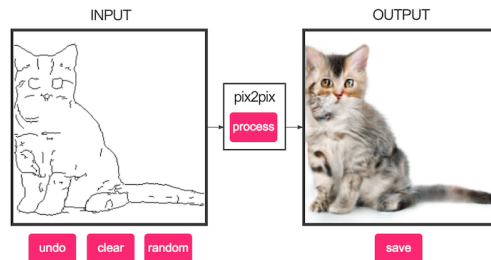
1. First, sample a latent factor z from some distribution, and then
2. sample a datum $x \sim p(x | z)$.

The central point to appreciate is that we know neither Nature's distribution for sampling latents z nor its generative model $p(x | z)$. We will need to fit both these quantities using a training dataset of images/text.

- 1 The purpose of doing so can be many-fold, e.g., we may want to
- 2 generate new data to amplify the size of our training set, given a part of
- 3 the input image (say due to occlusions, or image corruption) we may want
- 4 to complete the rest of it.



5



6

- 7 Most such applications require the knowledge of the latent factors that
- 8 generated the data. Therefore, formally, we are interested in computing
- 9 the posterior distribution of the latents and Nature's distribution of the
- 10 latents

$$\text{posterior } p(z | x)$$

$$\text{prior } p(z)$$

- 11 using samples in a training dataset $D = \{x^i\}_{i=1}^n$. Notice that we do not
- 12 need labels for this problem, effectively labels $y^i = x^i$ itself because our
- 13 generative model should of course be very good at generating samples
- 14 from the training data.

15 15.2 Some technical basics

16 15.2.1 Variational calculus

- 17 We will first take a brief look at what is called variational calculus.

1 A function is something takes in a variable as input and returns the
 2 value of the function as the output, e.g., $\mathbb{R} \ni f(x) = 5x^2$ for $x \in \mathbb{R}$.
 3 Similarly, a *functional* is an object that takes in a *function* as an input and
 4 returns a real number as the output. An example of this is entropy

$$\mathbb{R} \ni H[p] = - \int p(x) \log p(x) \, dx$$

5 which takes in a probability density p as the input and returns a real number.
 6 Entropy is therefore a *functional*. Just like standard calculus where we
 7 take derivatives/minimize over functions, we can also take derivatives of
 8 the functional.

9 The functional derivative $\frac{\delta H[p]}{\delta p}(x)$ is defined in a funny way as

$$\int \frac{\delta H[p]}{\delta p}(x) \varphi(x) \, dx = \lim_{\epsilon \rightarrow 0} \frac{H[p + \epsilon\varphi] - H[p]}{\epsilon}$$

10 for any arbitrary function φ . Essentially, you perturb the argument to the
 11 functional p by some epsilon and see how much the functional changes.
 12 The change in the functional is measured using the test function φ by
 13 integrating its changes $\frac{\delta H(p)}{\delta p}(x)$ at each point x in the domain. There may
 14 be certain conditions that the perturbation φ needs to satisfy depending
 15 upon the problem, e.g., since $p + \epsilon\varphi$ should also be legitimate probability
 16 density, the functional derivative above should only consider test functions
 17 φ such that

$$\forall \epsilon \int (p(x) + \epsilon\varphi(x)) \, dx = 1 \Rightarrow \int \varphi(x) \, dx = 0.$$

18 The KL-divergence between two probability densities,

$$\text{KL}(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} \, dx,$$

19 is another such functional; it has two arguments p and q .

Variational optimization is concerned with minimizing functionals.

20 For instance, while a problem looks like

$$w^* = \underset{w \in \mathbb{R}^p}{\operatorname{argmin}} \ell(w)$$

21 in standard optimization, a variational optimization problem with KL-
 22 divergence as the loss given a fixed density p looks like

$$q^* = \underset{q \in \mathcal{Q}}{\operatorname{argmin}} \text{KL}(q \parallel p). \quad (15.1)$$

23 The variable of optimization is the probability density q and we will denote
 24 the domain of the variable by \mathcal{Q} . Since we want q to be a legitimate

1 probability density, we should choose

$$Q \subseteq \mathcal{P}(\mathcal{X})$$

2 where $\mathcal{P}(\mathcal{X})$ denotes the set of all probability densities on some domain
3 \mathcal{X} .

4 **Picking the domain and objective in variational optimization** Picking
5 a good domain Q to minimize over is important. It is similar to the notion
6 of the a hypothesis class in machine learning. If Q is too big, it is
7 difficult to solve the optimization problem but we obtain a better value to
8 $\text{KL}(q||p)$. If Q is too small, the optimization problem may be easy but
9 we may not match the desired distribution p very well. Imagine if p is
10 a mixture of two Gaussians and we pick Q to be a family of uni-modal
11 Gaussian distributions. Since the KL-divergence is zero if and only if the
12 two distributions are equal, we are never going to be able to minimize
13 it completely. On the other hand, if we pick Q to be the family of
14 distributions with 2 or more Gaussian modes, then we can perfectly match
15 p . Essentially, the crux of variational inference boils down to picking a
16 good family of distributions Q that makes solving (15.1) easy.

17 **What functional should we use to measure the distance between q and**
18 **p ?** The KL-divergence is popular and easy to use in practice but there
19 are many others. For example, when we studied the Gibbs distribution
20 we briefly talked about something called “Wasserstein metric”: if one
21 imagines a mountain of dirt given by distribution q and another mountain
22 of dirt p , the Wasserstein distance $W_2(q, p)$ is the amount of work done
23 in transporting the dirt from q to p ; it is also called the “[earth mover’s](#)
24 [distance](#)”. The Wasserstein metric is as legitimate a distance between
25 two distributions, just like the Kullback-Leibler divergence.

26 15.2.2 Laplace approximation

27 Laplace approximation is a very useful trick to solve variational optimiza-
28 tion problems approximately. Here is how it works. Suppose we have to
29 estimate an expectation of our random variable $\varphi(w)$

$$\mathbb{E}_{w \sim e^{-n\ell(w)}} [\varphi(w)] = \int e^{-n\ell(w)} \varphi(w) dw$$

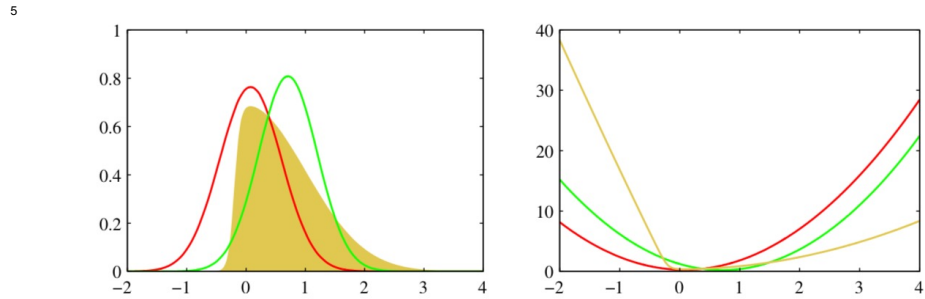
30 over draws $w \sim$ from some probability distribution $e^{-n\ell(w)}$ for some large
31 value of n . The above integral takes many values, some have small $\ell(w)$
32 and some have large $\ell(w)$. The values of w where $\ell(w)$ is small are the
33 ones that have the highest $e^{-n\ell(w)}$, especially as $n \rightarrow \infty$, and therefore
34 the ones that count for most in the integral. The Laplace approximation
35 is a trick to estimate the integral for large n . It replaces the integral by

1 taking a Taylor series expansion of the exponent as follows.

$$\begin{aligned} \int e^{-n\ell(w)} \varphi(w) \, dw &\approx \int \varphi(w) e^{-n(\ell(w^*) + \frac{1}{2}(w-w^*)^\top \nabla^2 \ell(w^*)(w-w^*))} \, dw \\ &= e^{-n\ell(w^*)} \int \varphi(w) e^{-\frac{n}{2}(w-w^*)^\top \nabla^2 \ell(w^*)(w-w^*)} \, dw \end{aligned} \quad (15.2)$$

2 where $w^* = \operatorname{argmin} \ell(w)$ is the global minimum of $\ell(w)$. The integral is
3 now with respect to a Gaussian distribution and can be done more easily.

4 How does a Laplace approximation look? Let us look at an example.



5 **Figure 10.1** Illustration of the variational approximation for the example considered earlier in Figure 4.14. The
6 left-hand plot shows the original distribution (yellow) along with the Laplace (red) and variational (green) approx-
7 imations, and the right-hand plot shows the negative logarithms of the corresponding curves.

8 Although the Laplace approximation trick is reasonable only for very
9 large values of n , it is a quick way to estimate what the correct domain
10 of the a variational optimization problem should be. For example, if we
11 are approximating a probability distribution with a Gaussian family, the
12 Laplace approximation tells us what the mean of the family should be
13 and we can only consider the variance as the variable in a variational
14 optimization problem.

14 15.2.3 Digging deeper into KL-divergence

15 Let us take an example to understand KL-divergence better.

16 Figure 15.1 compares two forms of KL-divergence. The green contours
17 represent equi-probability lines (1,2,3 standard deviations) for a two-
18 dimensional correlated Gaussian $p(z_1, z_2)$. Red contours represent similar
19 equi-probability lines for the variational approximation of this distribution
20 using an uncorrelated Gaussian distribution

$$q(z) = q_1(z_1)q_2(z_2)$$

21 where both q_1, q_2 are one-dimensional Gaussians. The variational family
22 $q \in \mathcal{Q}$ thus consists of factored uncorrelated Gaussians and we are trying
23 to find the best member of this family that approximates the *correlated*
24 true distribution $p(z)$.

25 Left panel (a) in Figure 15.1 shows the result using the forward
26 KL-divergence minimization

$$q^* = \operatorname{argmin}_{q \in \mathcal{Q}} \operatorname{KL}(q \parallel p).$$

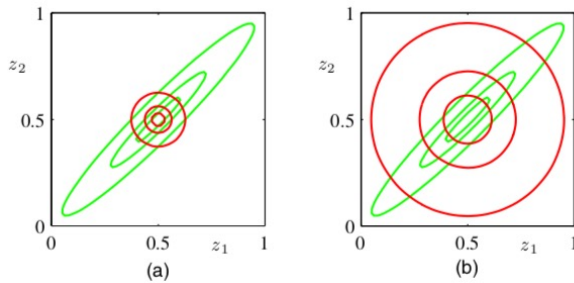


Figure 15.1: Comparison between the variational approximation of a correlated Gaussian using forward and reverse KL divergence and a factored Gaussian family.

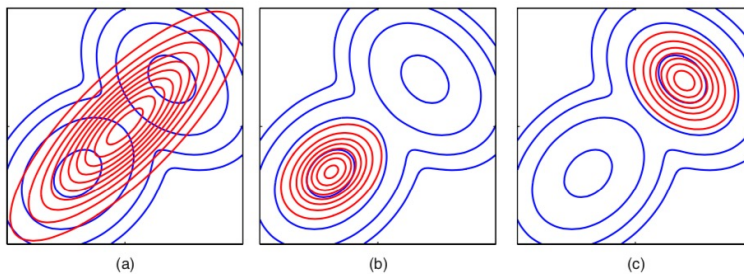


Figure 15.2: Approximating a multi-modal distribution using a uni-modal variational family.

- 1 while the right panel (b) shows the result for the reverse KL-divergence
2 minimization

$$q^* = \operatorname{argmin}_{q \in \mathcal{Q}} \operatorname{KL}(p \parallel q).$$

- 3 We see that both these forms capture the mean of the true distribution
4 $p(z)$ correctly. The variance of the two approximations is quite different
5 depending upon which form we employ.

- 6 We next consider the case when a multi-modal probability distribution
7 $p(z)$ is approximated using a unimodal Gaussian distribution. Both these
8 examples are very often seen in practice, the distribution of true data/latent
9 factors is often correlated and multi-modal. We have seen one instance of
10 this: the distribution of weights of a deep network in the Gibbs distribution
11 is multi-modal because of multiple global minima.

- 12 The distribution p is bi-modal and the variational problem is no longer
13 convex in this case; depending upon the initial condition using q , one may
14 get different solutions shown in panels (a), (b) or (c). You should also
15 think about the fact that the solution in panel (a) could be the solution of
16 optimizing the reverse KL divergence; in contrast, the solutions in panels
17 (b) and (c) have to be the ones obtained from optimizing the forward KL
18 divergence.

- 19 KL-divergence is not the only distance used in variational inference
20 and there are many many other ones. You should think of these different

🔗 Use the expression of the KL-divergence to convince yourself why the forward KL under-estimates the variance while the reverse KL over-estimates the variance in Figure 15.1.

ways to measure distances between probability distributions in variational inference as different surrogate losses; which one we use is highly problem dependent although the forward KL-divergence $\text{KL}(q \parallel p)$ is the most common.

15.3 Evidence Lower Bound (ELBO)

We now go back to the generative model.

We will formalize our goal in generative modeling as computing Nature's posterior distribution of latent factors

$$p(z \mid x).$$

We have access to a training dataset $D = \{(x^i)\}_{i=1}^n$. We do not know (i) what form Nature's posterior distribution takes, e.g., Gaussian, multi-modal distribution etc. and (ii) we do not know the true latent factors z that Nature uses. So we are going to approximate the true posterior using some variational family of our choice

$$\mathcal{Q} \ni q^*(z \mid x) \approx p(z \mid x).$$

This is the basic idea of variational inference: to approximate a complex distribution $p(z \mid x)$ using a member of from a simpler family of our choosing \mathcal{Q} . In practice, this variational family \mathcal{Q} will be parameterized by a deep network.

With this background, the mathematical process of executing the above program is quite simple. We will simply minimize the KL-divergence

$$q^*(z \mid x) = \underset{q \in \mathcal{Q}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n \text{KL}(q(z \mid x^i) \parallel p(z \mid x^i)). \quad (15.3)$$

We next rewrite this KL-divergence above in a special form.

$$\begin{aligned} 0 &\leq \text{KL}(q(z \mid x^i) \parallel p(z \mid x^i)) \\ &= \mathbf{E}_{z \sim q(z \mid x^i)} \left[\log \frac{q(z \mid x^i)}{p(z \mid x^i)} \right] \\ &= - \mathbf{E}_{z \sim q(z \mid x^i)} [\log p(z \mid x^i)] + \mathbf{E}_{z \sim q(z \mid x^i)} [\log q(z \mid x^i)] \\ &= - \mathbf{E}_{z \sim q(z \mid x^i)} [\log p(z, x^i) - \log p(x^i)] + \mathbf{E}_{z \sim q(z \mid x^i)} [\log q(z \mid x^i)] \\ &= \log p(x^i) - \mathbf{E}_{z \sim q(z \mid x^i)} [\log p(z, x^i)] + \mathbf{E}_{z \sim q(z \mid x^i)} [\log q(z \mid x^i)]. \\ \Rightarrow \log p(x^i) &\geq \mathbf{E}_{z \sim q(z \mid x^i)} [\log p(z, x^i)] - \mathbf{E}_{z \sim q(z \mid x^i)} [\log q(z \mid x^i)] \end{aligned}$$

This is quite interesting. The left-hand side of this inequality is the log-likelihood of the data under Nature's distribution, i.e., it is fixed and

1 independent of what we do. The left-hand side is also called the “evidence”
 2 in statistics (which is a bit ironic because we can never know the evidence).
 3 The right-hand side

$$\text{ELBO}(q, x^i) := \mathbb{E}_{z \sim q(z|x^i)} [\log p(z, x^i)] - \mathbb{E}_{z \sim q(z|x^i)} [\log q(z | x^i)]. \quad (15.4)$$

4 is a lower bound on the evidence and therefore called the Evidence Lower
 5 Bound (ELBO).

Next comes a key step: a good generative model should be such that the evidence of the training data, i.e., the log-likelihood of this data under Nature’s distribution, should be large under the model. We therefore want to maximize the ELBO on our training data

$$q^*(z | x) = \operatorname{argmax}_{q \in \mathcal{Q}} \frac{1}{n} \sum_{i=1}^n \text{ELBO}(q, x^i). \quad (15.5)$$

to find the posterior distribution of the latent factors $q^*(z)$. Maximizing ELBO is equivalent to minimizing the average KL-divergence $\text{KL}(q(z | x^i) || p(z | x^i))$ over all training samples.

6 We will again solve the optimization problem in (15.5) using stochastic
 7 gradient descent. Before we study how to do that, let us consider what
 8 model we have developed so far. The solution to this problem

$$q^*(z | x) \approx p(z | x)$$

9 approximates Nature’s posterior distribution. If we maximize ELBO well,
 10 given an input x , samples $z \sim q^*(z | x)$ are likely to be the latent factors
 11 that Nature could have chosen while rendering this image. But we still do
 12 not know how to synthesize an image x for these latent factors. We now
 13 rewrite ELBO in a different form to understand this.

$$\begin{aligned} \text{ELBO}(q, x^i) &= \mathbb{E}_{z \sim q(z|x^i)} [\log p(z, x^i)] - \mathbb{E}_{z \sim q(z|x^i)} [\log q(z | x^i)] \\ &= \mathbb{E}_{z \sim q(z|x^i)} [\log p(x^i | z) + \log p(z)] - \mathbb{E}_{z \sim q(z|x^i)} [\log q(z | x^i)] \\ &= \mathbb{E}_{z \sim q(z|x^i)} [\log p(x^i | z)] - \text{KL}(q(z | x^i) || p(z)). \end{aligned}$$

14 This form of ELBO

$$\text{ELBO}(q, x^i) = \mathbb{E}_{z \sim q(z|x^i)} [\log p(x^i | z)] - \text{KL}(q(z | x^i) || p(z)) \quad (15.6)$$

15 is very interesting. The first term is Nature’s log-likelihood of datum x^i
 16 given the latent factor z sampled from *our* candidate posterior q . The
 17 second term is the discrepancy between our variational approximation
 18 of the posterior $q^*(z | x^i) \approx p(z | x^i)$ and Nature’s true marginal
 19 distribution over latent factors $p(z)$. This alternative form of ELBO is

1 conceptually very similar to what we do in standard classification, e.g.,

$$\operatorname{argmin}_w \left\{ \ell(w) + \frac{\alpha}{2} \|w\|^2 \right\}.$$

2 We would like our $q(z | x^i)$ to be close to Nature's prior distribution $p(z)$
 3 but at the same time be such that samples from $q(z | x^i)$ have a high
 4 log-likelihood $p(x^i | z)$ of synthesizing images in the training set. The
 5 KL-term is therefore a regularizer for the first data-fitting term.

6 15.3.1 Parameterizing ELBO

7 What variational family \mathcal{Q} should we choose? Say we parametrized each
 8 distribution $q(z | x^i)$ by its mean and diagonal of the covariance.

$$\mathbb{R}^m \ni z \sim q(z | x^i) = N(\mu(x^i), \sigma^2(x^i)I) \in \mathcal{Q}(x^i)$$

9 where $\mu(x^i), \sigma^2(x^i) \in \mathbb{R}^m$. The ELBO in (15.6) is totally independent
 10 for each x^i in the training dataset, so all $i \in \{1, \dots, n\}$ we can solve for

$$\mu^*(x^i), \sigma^2(x^i) = \operatorname{argmax}_{\mu, \sigma^2} \text{ELBO}(N(\mu(x^i), \sigma^2(x^i)I), x^i).$$

11 But this is not a good idea: the parameters μ, σ^2 are distinct for each
 12 input x^i and effectively they are being trained using a dataset of only input
 13 image x^i .

Amortized variational inference is a clever trick that ties together the variational families $\mathcal{Q}(x^i)$. We will be using a deep network with parameters $u \in \mathbb{R}^p$ that takes x^i as the input and gives $\mu(x^i; u), \sigma^2(x^i; u)$ as the outputs

$$\text{Encoder} : x^i \xrightarrow[\text{parameters } u]{} \mu(x^i; u), \sigma^2(x^i; u).$$

The variational family $\mathcal{Q}(x^i)$ that we are considering is therefore the set of distributions expressed by this deep network with p parameters. The family $\mathcal{Q}(x^i)$ is still distinct for each datum x^i but they are all tied together by the same weights u .

Encoder. We will call this deep network the encoder because it takes in an input x^i and encodes it into $\mu(x^i; u), \sigma^2(x^i; u)$ which parameterize the distribution of the latent factors.

14 **Decoder.** Observe that although we have now parameterized the distri-
 15 bution $q(z | x^i)$ using a deep network with weights u , we still do not know
 16 how to model the term $p(x^i | z)$. After all, this is Nature's log-likelihood.

17 We have a dataset $\{(x^i, z^i)\}_{i=1}^n$ that consists of the images x^i and
 18 their corresponding latents z^i sampled from our encoder. We are going to

1 model Nature's rendering process $p(x | z)$ using a deep network. This is
 2 a program that we have done many times in the past, e.g., we model the
 3 targets in classification y^i as samples from the softmax distribution with
 4 images x^i as the input and train the weights using maximum-likelihood
 5 (as you may recall, this is equivalent to the cross-entropy loss).

6 We can repeat that program here: we are going to learn a deep network

$$\text{Decoder : } p_v(x^i | z) \approx p(x^i | z).$$

7 with parameters $v \in \mathbb{R}^p$ that models Nature's likelihood $p(x^i | z)$.

8 **Different possible decoders for MNIST** Depending upon the type of
 9 data x^i , we will code up the deep network in different ways. For instance,
 10 if each pixel of $x^i \in \mathbb{R}^{28 \times 28}$ is grayscale $[0, 255]$ like it is in MNIST, the
 11 output of the decoder is a multinomial with size $28 \times 28 \times 256$.

12 If we take the training dataset as binarized MNIST (if pixel jk is less
 13 than 128 set it to 0, else set it to 1), then the output of the decoder has size
 14 $28 \times 28 \times 2$ and we can fit this using a logistic distribution at each pixel

$$p_v(x^i | z) = \prod_{j,k=1}^{28} \underbrace{p_v(x_{jk}^i | z)}_{\text{logistic distribution for pixel } x_{jk}^i \in \{0,1\}}$$

15 The log-likelihood term in (15.6) will then correspond to the logistic loss
 16 as discussed in the Homework.

17 **Using a mean-field prior $p(z)$.** We do not know what the prior distri-
 18 bution $p(z)$ in (15.6) is. We will choose a simple prior

$$p(z) = \prod_{j=1}^m p_j(z_j) \quad (15.7)$$

19 where $p_i(z_i)$ is the distribution of the i^{th} latent factor z_i . Such distributions
 20 are called mean-field priors (where the distribution of a vector $z \in \mathbb{R}^m$ is
 21 modeled as independent distributions on its components). We will further
 22 choose each distribution

$$p_j(z_j) = N(0, 1)$$

23 to be a zero-mean standard Gaussian distribution. This is a Gaussian
 24 mean-field prior. Just like the choice of a regularizer is critical in machine
 25 learning for obtaining good generalization, the choice of a prior is critical
 26 in variational inference for synthesizing good images from the generative
 27 model.

28 15.4 Gradient of the ELBO

29 We now have all the ingredients in place for training a variational generative
 30 model. Let us summarize our setup.

▲ The distribution of labels y^i in classification was one-hot vectors, so the softmax layer created a multinomial distribution on the classes.

1. Encoder parameters u are weights of a deep network that takes in x^i as input and outputs parameters $\mu(x^i), \sigma^2(x^i)$ of the latent distribution. We have tacitly assumed the latent posterior $p(z | x^i)$ to be a Gaussian here; if you have a problem where you wish to have a different latent, e.g., all the latent genes that could have caused a particular cancer, then you want to output the parameters of that distribution from the encoder.
2. The decoder models the likelihood $p_v(x^i | z)$ using parameters v .
3. The prior $p(z)$ will be a mean-field Gaussian distribution. The prior has no parameters in our case, although you may see research papers where the prior also has its own parameters. A popular choice is to use

$$\text{ELBO}_\beta(q, x^i) = \mathbb{E}_{z \sim q(z|x^i)} [\log p(x^i | z)] - \beta^{-1} \text{KL}(q(z | x^i) || p(z))$$

in place of the standard ELBO. The hyper-parameter $\beta > 0$ gives more control over the strength of the prior; this is of course akin to picking the weight-decay coefficient.

▲ The concept of variational inference and ELBO are much more general than generative models or the encoder-decoder structure that we have developed. Go through the assigned reading material to learn more.

The ELBO when rewritten in terms of the encoder and decoder parameters looks as follows.

$$\text{ELBO}(u, v; x^i) = \mathbb{E}_{z \sim q_u(z|x^i)} [\log p_v(x^i | z)] - \text{KL}(q_u(z | x^i) || p(z)). \quad (15.8)$$

Our goal is to fit the weights u, v using

$$u^*, v^* = \underset{u, v \in \mathbb{R}^p}{\text{argmax}} \frac{1}{n} \sum_{i=1}^n \text{ELBO}(u, v; x^i). \quad (15.9)$$

The number of parameters of the encoder and decoder can be different but for clarity we imagine them to be the same.

(15.9) is an optimization problem and in this section, we will see how to compute the gradient of the objective so that we can solve the problem using SGD.

15.4.1 The Reparameterization Trick

Focus on the gradient with respect to u of the first term of ELBO

$$\nabla_u \mathbb{E}_{z \sim q(z|x^i)} [\varphi(z)].$$

We have written $\log p_v(x^i | z) = \varphi(z)$ to keep the notation clear; we do not care about the exact form of the integrand in this section.

1 If we draw a computational graph for the forward propagation of this
2 term, it looks as follows

$$u, x^i \rightarrow \text{sample } z \text{ from } q_u(z | x^i) \rightarrow \varphi(z).$$

3 The intermediate sampling step is troublesome, we do not really know
4 how to use the chain rule of calculus across sampling, i.e., given

$$\overline{\varphi(z)} := \frac{d}{d_u} \varphi(z)$$

5 we need to compute $\bar{u} = d\ell/d_u u$. We only know how to apply the chain
6 rule for *deterministic operations* of the form

$$u, x^i \rightarrow z = \text{some deterministic function } g(u, x^i) \rightarrow \varphi(z),$$

7 in which case we use the standard backprop across the function g .

The Reparameterization Trick enables us to obtain backpropagation gradients across sampling operations via a creative use of the Laplace approximation of the distribution $q_u(z | x^i)$.

8 We known from the Laplace approximation that we can compute an
9 expectation over z using a Gaussian centered at the global maximum of
10 the distribution $q_u(z | x^i)$ with variance equal to the inverse Hessian at
11 that maximum. Motivated by this, the Reparameterization Trick *rewrites*
12 the random variable z as

$$z = \mu(x^i; u) + \sigma(x^i; u) \odot \epsilon$$

13 where

$$\epsilon \sim N(0, I_{m \times m})$$

14 is a sample from a standard multi-variate Gaussian distribution and the
15 notation \odot denotes element-wise product. Effectively, we imagine that
16 the encoder outputs

$$\begin{aligned} \mu(x^i; u) &= \underset{z}{\operatorname{argmax}} q_u(z | x^i) \\ \sigma^2(x^i; u) &= \operatorname{diag} \left([\nabla_z^2 q_u(z | x^i)]^{-1} \right). \end{aligned}$$

17 Just like the integral in (15.2) was performed over the Gaussian, the
18 integral over z can be rewritten as an integral over ϵ

$$\begin{aligned} \nabla_u \mathbf{E}_{z \sim q_u(z | x^i)} [\varphi(z)] &= \nabla_u \mathbf{E}_{\epsilon \sim N(0, I)} [\varphi(\mu(x^i; u) + \sigma(x^i; u) \odot \epsilon)] \\ &= \mathbf{E}_{\epsilon \sim N(0, I)} [\nabla_u \varphi(\mu(x^i; u) + \sigma(x^i; u) \odot \epsilon)] \\ &\approx \frac{1}{N} \sum_{j=1}^N \nabla_u \varphi(\mu(x^i; u) + \sigma(x^i; u) \odot e^j), \text{ where } e^j \sim N(0, I). \end{aligned}$$

1 We can take the gradient operator inside the expectation in this case because
 2 ϵ no longer depends on the weights u . The term $\nabla_u \varphi(\mu(x^i; u) + \sigma(x^i; u) \odot \epsilon^j)$
 3 is a deterministic operation given a sample z^j and can be computed using
 4 standard backpropagation.

5 15.4.2 Score-function estimator of the gradient

6 Let us look at an alternative way to compute the same gradient.

$$\begin{aligned}
 \nabla_u \mathbb{E}_{z \sim q_u(z|x^i)}[\varphi(z)] &= \nabla_u \int \varphi(z) q_u(z|x^i) dz \\
 &= \int \varphi(z) \nabla_u q_u(z|x^i) dz \\
 &= \int \varphi(z) \frac{\nabla_u q_u(z|x^i)}{q_u(z|x^i)} q_u(z|x^i) dz \\
 &= \int \varphi(z) \nabla_u \log q_u(z|x^i) q_u(z|x^i) dz \\
 &= \mathbb{E}_{z \sim q_u(z|x^i)}[\varphi(z) \nabla_u \log q_u(z|x^i)] \\
 &\approx \frac{1}{N} \sum_{j=1}^N \varphi(z^j) \nabla_u \log q_u(z^j|x^i), \text{ with } z^j \sim q_u(z|x^i).
 \end{aligned}
 \tag{15.10}$$

7 The term

$$\frac{\nabla_u q_u(z|x^i)}{q_u(z|x^i)} = \nabla_u \log q_u(z|x^i) \tag{15.11}$$

8 is called the score function of a probability distribution q_u . The above
 9 calculation is quite beautiful: calculating the gradient of the expectation
 10 of any quantity $\varphi(z)$ is equal to the expectation of the same quantity
 11 weighted by the score function

$$\nabla_u \mathbb{E}_{z \sim q_u}[\varphi(z)] = \mathbb{E}_{z \sim q_u}[\varphi(z) \nabla_u \log q_u].$$

12 Due to this trick, we can compute the gradient using N samples

$$z^j \sim p_u(z|x^i) \tag{15.12}$$

13 from the encoder; this is easy if, say, the encoder outputs the mean and
 14 standard-deviation of the distribution of the latents. Given z^j , the gradient

$$\nabla_u \log q_u(z^j|x^i)$$

15 is just the standard back-propagation gradient of the quantity $\log q_u(z^j|x^i)$
 16 with respect to weights u of the deep network and can be computed using
 17 autograd.

The key difference between the Reparameterization Trick and

the score-function estimator is that in the latter, we do not need to make sure that the gradient $d\ell/dz^j$ can be back-propagated across the sampling operation. The score-function estimator directly computes the gradient of the entire expectation by a weighted average across the samples.

Having two different ways of computing the same gradient may seem redundant but they both are suited to very different applications. The Reparameterization Trick is not accurate in cases when the distribution $q_u(z | x^i)$ is multi-modal because we have only one mean $\mu(x^i)$ around which the samples are drawn. The score-function trick does not have this problem because so long as iid samples are drawn in (15.12) (using any method, e.g., importance sampling) we obtain true estimate of the gradient. The problem in score-function estimator lies in that the denominator $q_u(z | x^i)$ in (15.11) can take very small values if the particular sample z is unlikely. The summation (15.10) is a combination of many N , some very large in magnitude and some very small; the variance of score-function estimate of the gradient in (15.10) can therefore be quite large in most problems.

Typically, the Reparameterization Trick is commonly used in generative models while both the Reparameterization Trick and the score-function estimator are used widely in Reinforcement Learning.

1 15.4.3 Gradient of the remaining terms in ELBO

2 The gradient with respect to weights v of the decoder of the first term in
3 ELBO

$$\nabla_v \mathbb{E}_{z \sim q_u(z|x^i)} [\log p_v(x^i | z)]$$

4 is simply the standard backpropagation gradient (the sampling distribution
5 of the encoder does not depend on the weights of the decoder).

6 Let us focus on the second term

$$\text{KL} \left(q_u(z | x^i) \parallel \prod_{j=1}^m p_j(z_j) \right). \quad (15.13)$$

7 where $p_j(z_j) = N(0, 1)$ are terms of the mean-field prior. The gradient
8 of this term with respect to weights of the decoder is zero

$$\nabla_v \text{KL} \left(q_u(z | x^i) \parallel \prod_{j=1}^m p_j(z_j) \right) = 0.$$

9 Following the reasoning in the Reparameterization Trick, we are positing
10 that $q_u(z | x^i)$ is a Gaussian distribution:

$$q_u(z | x^i) = N(\mu(x^i; u), \sigma^2(x^i; u)I).$$

1 Notice that $\sigma^2(x^i; u) \in \mathbb{R}^m$ is the diagonal of the covariance and therefore
 2 the individual marginals $q_u(z_j | x^i)$ and $q_u(z_{j'} | x^i)$ for two indices j, j'
 3 are independent. We can therefore write

$$q_u(z | x^i) = \prod_{j=1}^m N(\mu_j(x^i; u), \sigma_j^2(x^i; u)). \quad (15.14)$$

4 The KL-divergence of a univariate Gaussian $N(\mu_1, \sigma_1^2)$ with respect
 5 to the standard Gaussian is

$$\text{KL}(N(\mu, \sigma^2) || N(0, 1)) = \log \frac{1}{\sigma} + \frac{\sigma^2 + \mu^2}{2} - \frac{1}{2}. \quad (15.15)$$

6 The general formula is

$$\text{KL}(N(\mu_1, \sigma_1^2) || N(\mu_2, \sigma_2^2)) = \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}.$$

7 Due to (15.14), the KL-divergence in (15.13) is a sum of the KL-
 8 divergences of the individual univariate Gaussians

$$\text{KL}(q_u(z | x^i) || p(z)) = -\frac{1}{2} \sum_{j=1}^m (\log \sigma_j^2(x^i; u) - \sigma_j^2(x^i; u) - \mu_j^2(x^i; u) + 1). \quad (15.16)$$

9 The right-hand side of this equation is only a function of u and its gradient
 10 can be calculated using standard back-propagation.

11 This completes our development of ELBO. Using the gradient calcu-
 12 lated in this section, we can use SGD to maximize the objective in (15.5)
 13 and train a generative model.

14 15.5 Some comments

15 Although the mathematics of ELBO seems complicated, it is quite easy to
 16 implement generative models using variational inference in practice. You
 17 did for a simple MNIST problem in the homework/recitation but if the
 18 encoder and decoder are convolutional and deconvolutional architectures
 19 respectively, we can get very sophisticated generative models.



Figure 15.3: Samples from a state-of-the-art VAE trained on ImageNet (Razavi et al., 2019)

7 Prove that

$$\begin{aligned} \text{KL} \left(\prod_{j=1}^m q_j(z_j) || \prod_{j=1}^m p_j(z_j) \right) \\ = \sum_{j=1}^m \text{KL}(q_j(z_j) || p_j(z_j)). \end{aligned}$$

1 Variational inference and information-theoretic methods are a rich
2 (and old) area of research and there are many modifications/innovations
3 to ELBO, e.g., read [Alemi et al. \(2018\)](#) for some simple yet deep modifi-
4 cations.

Chapter 16

Generative Adversarial Networks

Reading

1. Andrew Ng's notes on generative models
<http://cs229.stanford.edu/notes/cs229-notes2.pdf>
2. The original GAN paper by Goodfellow et al. (2014)
3. "The Numerics of GANs" by Mescheder et al. (2017)

In the previous chapter, we used variational methods to build a generative model for the data. In this case, we are given samples $D = \{x^i\}_{i=1}^n$ and would like to build a model that can synthesize new data. For every data x that a decoder synthesizes at test time using latent variables z , we can calculate the likelihood

$$x \sim p_v(x|z), \text{ for any } z \sim N(0, I).$$

This likelihood is an indicator of how unlikely the data x is under z . Models for which we can calculate such likelihood are called explicit generative models, i.e., they give a sample x and also report its likelihood. In this chapter, we will look an alternative class of generative models that are implicit, i.e., they only give a sample x but do not report its likelihood.

A Generative Adversarial Network (GAN) consists of two neural networks: a Generator and a Discriminator. The Generator works in the same way as the decoder in a variational auto-encoder. Given a sample z from some distribution, most commonly a standard normal, we train a neural network to generate a sample

$$x = g_v(z).$$

1 GANs differ from explicit models in how they train the generator, the
2 discriminator is used for this purpose. We will look at this next.

3 16.1 Two-sample tests and Discriminators

4 We will first take a short trip into an area of statistics known as decision
5 theory. Consider two datasets coming from two distributions $p(x)$ and
6 $q(x)$

$$D_1 = \{x^1, \dots, x^n, : x^k \sim p(x)\}$$

$$D_2 = \{x^1, \dots, x^n, : x^k \sim q(x)\}.$$

7 We would like to check if these two distributions are the same given
8 access to only their respective datasets D_1 and D_2 . Let us define the *null*
9 *hypothesis* which claims that the two distributions are the same.

$$H_0 : p = q$$

10 The alternate hypothesis is

$$H_1 : p \neq q.$$

11 The goal of the so-called “two-sample test” is to decide whether H_0 is
12 true or not. A typical two-sample test will construct a statistic (recall from
13 Chapter 7 that a statistic is any function of the data)

$$\hat{t}$$

14 out of the two datasets, e.g., their individual means, their variances, and
15 will use this statistic to *accept or reject* the null hypothesis, i.e., decide
16 whether H_0 is true or false.

17 Let’s say that we pick a threshold t_α , and the test statistic \hat{t} is the
18 difference of the means

$$\hat{t} = \left| \frac{1}{n} \sum_{x \in D_1} x - \frac{1}{n} \sum_{x \in D_2} x \right|.$$

19 **Level of a test** A statistician will then say that the null hypothesis is
20 valid with *level* α if

$$P_{D_1 \sim p, D_2 \sim p}(\hat{t} > t_\alpha) \leq \alpha. \quad (16.1)$$

21 In other words, if the null hypothesis were true (both D_1 and D_2 are
22 drawn from the same distribution p) and if the probability of our empirical
23 statistic \hat{t} being larger than some *chosen* threshold t_α is smaller than some
24 *chosen* probability α , then we know that the two distributions are the same
25 despite only having finite data to check. The threshold α is called the
26 *p*-value in the statistics literature and you will have seen statements like
27 “gene marker XX is correlated with disease YY with *p*-value of 10^{-3} ” or
28 “smokers and non-smokers have different distributions of cancers with

▲ The concept of a hypothesis here is different from what we saw in generalization/VC-theory. Hypothesis in decision theory simply means our hunch about a particular situation, e.g., $p = q$.

1 p -value of 10^{-3} .

2 **Power of a test** The power of a two-sample test is the probability of
 3 rejecting the null hypothesis when it is actually false. We want tests with
 4 a large power, i.e., we like

$$P_{D_1 \sim p, D_2 \sim q} (\hat{t} > t_\alpha) \quad (16.2)$$

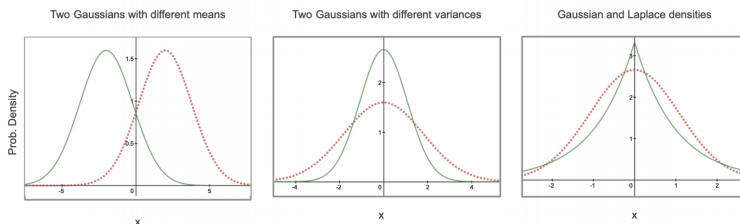
5 being large if the two datasets D_1 and D_2 are drawn from two different
 6 distributions p and q respectively.

The key point to remember about two-sample tests is that they let us check if two distributions are the same without knowing anything about the distributions. We only need access to the samples and can run this test. This is fundamentally different than say

$$\text{KL}(q \parallel p) = \int q(x) \log \frac{q(x)}{p(x)} dx$$

where we need to know the probabilities $q(x), p(x)$ to compute the distance between distributions.

7 **Example 16.1.** A two-sample test requires three things, a statistic \hat{t} , a
 8 level α and a threshold for the statistic t_α . The latter two are numbers that
 9 a statistician can pick, e.g., picking $\alpha = 0.05$ is an accepted standard in
 10 most biological studies.



11

12 16.2 Building the Discriminator in a GAN

Finding two-sample test statistics for arbitrary distributions is

difficult, especially for high-dimensional problems where the samples are natural images. The key idea behind a Generator Adversarial Network (GAN) is to learn the statistic \hat{t} .

A good statistic is the one that lets us distinguish between data that comes from Nature's distribution and data that is synthesized by our generative model. This statistic, which is called the discriminator in GAN, is a critic of the generative model's results. It has a *high power* in (16.2) if the generated samples are different from those of Nature. Why? Because in this case for most thresholds t_α that we can pick, the power of the two-sample test in (16.2) will be large.

The discriminator should also be sound, i.e., if the two distributions are indeed the same (e.g., if our generator is as good as good as Nature's renderer), the discriminator should have a *low level* α in (16.1).

1 We are going to train a binary classifier

$$d_u : \mathcal{X} \mapsto [0, 1]$$

2 that will act as the discriminator in a GAN. You should think of the
3 decision boundary of this binary classifier as the difference of the test
4 statistic and our threshold $\hat{t} - t_\alpha$.

5 We next create a dataset to train this classifier. Given n images from
6 Nature's distribution $p(x)$ and the distribution of our generator's images
7 $q(x)$, we will label the former with $y = 1$ and the latter with $y = 0$ to
8 create a joint dataset:

$$\begin{aligned} D_1 &= \{(x^i, 1)_{i=1, \dots, n} : x^i \sim p(x)\} \\ D_2 &= \{(x^i, 0)_{i=1, \dots, n} : x^i \sim q(x)\} \\ D &= D_1 \cup D_2. \end{aligned}$$

9 Fitting d_u on this problem can be done simply using the logistic loss
10 wherein d_u is modeling

$$P(y = 1|x) = d_u(x).$$

11 The logistic loss is therefore

$$u^* = \operatorname{argmin}_u -\frac{1}{n} \sum_{x \sim D_1} \log d_u(x) - \frac{1}{n} \sum_{x \sim D_2} \log(1 - d_u(x)). \quad (16.3)$$

12 Observe that this is the same logistic loss that we are used to; the only
13 difference being that the entire dataset has $2n$ samples with all the ones in
14 D_1 having labels $y = 1$ and all the ones in D_2 having labels $y = 0$.

▲ Notice how rigorous theory is used as an inspiration for developing GANs. This is a common theme that you will see in the deep learning literature; the models may seem *ad hoc* and sprung out of sheer intuition, but the reason they work well is often because there are sound theoretical principles behind them. Building this skill requires studying the classical curriculum (ML, statistics, optimization) but being creative in applying this curriculum with deep networks.

1 **What is the ideal discriminator?** The population risk corresponding
 2 to the discriminator's objective in (16.3) is

$$d^* = \operatorname{argmax}_d \mathbb{E}_{x \sim p} [\log d(x)] + \mathbb{E}_{x \sim q} [\log(1 - d(x))]. \quad (16.4)$$

3 We can take the variational derivative of this objective (just like you did
 4 in HW 3 to compute the optimal classifier in the bias-variance tradeoff) to
 5 get

$$d^*(x) = \frac{p(x)}{p(x) + q(x)}. \quad (16.5)$$

6 Observe that the ideal discriminator is $1/2$ if the two distributions p and
 7 q are the same. The intuitive reason for this is that if the data D were
 8 really coming from the same distribution, we would never be able to fit a
 9 logistic classifier to get better than 50% error because D_1 and D_2 have
 10 different labels in spite of having similar input data.

11 Think of you would use our discriminator to build a two-sample test
 12 for a given dataset. If given two datasets D_1 and D_2 labeled as above

$$\hat{t} := \frac{1}{n} \sum_{x \in D_1} \mathbf{1}_{\{d_u(x) > 0\}} + \frac{1}{n} \sum_{x \in D_2} \mathbf{1}_{\{d_u(x) < 0\}}$$

13 and the threshold $t_\alpha = 1/2$. This construction is an example of what is
 14 called a “classifier-based two-sample test”; you can read more about it
 15 at [Lopez-Paz and Oquab \(2016\)](#).

It can be shown that if the two distributions are not the same, the power of the two-sample test is an increasing function of the statistic \hat{t} . Therefore if we wanted to maximize the power, maximizing the test statistic \hat{t} of the discriminator is a good idea. This makes the discriminator more and more sensitive to the differences between samples from p and q .

16 16.3 Building the Generator of a GAN

The second key idea in a GAN is that the generator

▲ For a functional

$$L[d] = \int \log d(x) p(x) \, dx$$

the variational derivative is

$$\frac{\delta L}{\delta d}(x) = \frac{p(x)}{d(x)}.$$

Similarly, the variational derivative for

$$L[d] = \int \log(1 - d(x)) q(x) \, dx$$

is

$$\frac{\delta L}{\delta d}(x) = \frac{q(x)}{1 - d(x)}.$$

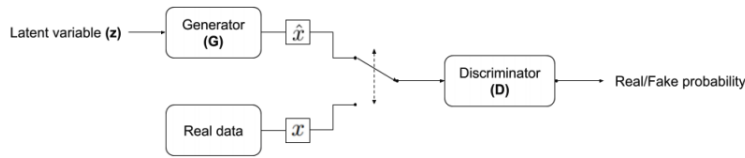


Figure 16.1: Schematic of the architecture in a GAN

$$g_v : \mathcal{Z} \rightarrow \mathcal{X}$$

that maps the latent space $\mathcal{Z} \subset \mathbb{R}^m$ to data space \mathcal{X} is trained to *minimize* the power of the two-sample test.

The generator g_v wants to synthesize data that look like they came from Nature's distribution $p(x)$. As the generator's distribution q comes closer to p , the accuracy of the discriminator d_u will degrade (it cannot distinguish between them as easily) and thereby discriminator will be forced to make its test statistic more sensitive to subtle differences between the two distributions.

16.4 Putting the discriminator and generator together

The GAN objective combines two objectives: the discriminator updates its weights u to maximize the power and the generator updates its weights v to minimize the power. We will write the population version of the optimization problem as follows.

$$\min_v \max_u E_{x \sim p(x)} [\log d_u(x)] + E_{x \sim q(x)} [\log (1 - d_u(x))] \quad (16.6)$$

Let us fill in a few more details. The dataset of real images consists of samples from Nature's distribution $p(x)$, so we will write it as a finite sum over our dataset $D = \{x^i \sim p\}_{i=1}^n$. The generator uses samples z from some generic distribution, e.g., a standard Gaussian distribution.

$$\min_v \max_u \frac{1}{n} \sum_{x \in D} [\log d_u(x)] + E_{z \sim N(0, I)} [\log (1 - d_u(g_v(z)))] \quad (16.7)$$

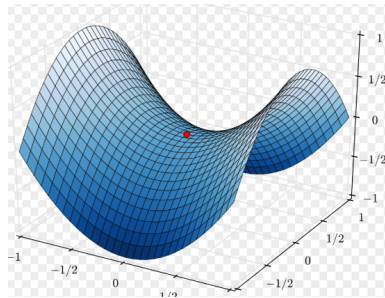
Training a GAN The objective in (16.7) is an example of a min-max optimization problem. Such problems are quite difficult to solve and this is why training GANs is quite difficult. In practice, we typically resort to a few crude tricks. We sample a mini-batch of real images $\{x^1, \dots, x^b\}$ and another mini-batch of noise vectors $\{z^1, \dots, z^b\}$. Using these two mini-batches

1. we update the generator g_v using the gradient of the objective with respect to v .

2. update the discriminator d_u using the gradient of the loss with respect to u .

There is no need for the Reparametrization Trick here because there is no expectation being taken over parametrized distributions. This is a big benefit of the GAN formulation as compared to variational inference; the former does not have to be careful while picking a variational family and complex deep networks can be used as the generator or the discriminator easily. Let us next make a few comments about the objective in (16.7).

Solving min-max problems is difficult This is a min-max problem: the generator is minimizing the objective and the discriminator is maximizing the objective. Such problems are hard to solve in optimization especially with gradient descent techniques. Consider an example of a saddle point



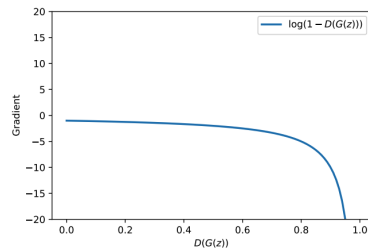
where the loss function increases in one direction and decreases in the other direction. Finding the solution of the min-max objective involves finding the saddle point. It is easy to appreciate that depending on how many steps of gradient descent we take for either of the min/max players we risk falling down or climbing up the hill. There are many many other factors that make solving such problems hard, e.g., learning rate, momentum, stochastic gradients if we are using mini-batches. Hyper-parameters are very tricky to pick while training GANs and this is often called “instability of training”.

A harsh discriminator inhibits the training of the generator The generator has a much more difficult task than the discriminator. During early stages of training, the generator needs to learn how to synthesize images whereas the discriminator can easily distinguish between bad images generated by the generator and good ones from our original dataset using very similar test statistics, e.g., an average of the RGB values all the pixels.

The gradient of the second term in the objective is

$$\nabla_v \log(1 - d_u(g_v(z))) = -\frac{\nabla_v d_u(g_v(z))}{1 - d_u(g_v(z))}.$$

As a function of $d_u(g_v(z))$ the second term in the objective thus looks like



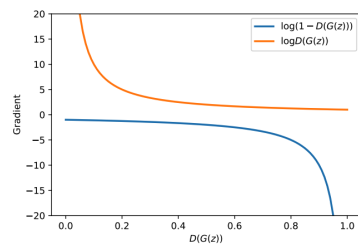
1

2 In other words, the gradient with respect to the generator's weights v
 3 is essentially zero if the generator is not working well (this is the case
 4 when $d_u(g_v(z))$ predicts a large negative value). This does not allow the
 5 generator to learn well; it is essentially like your advisor shooting down
 6 all your ideas.

7 Most GAN implementations therefore modify the second term in the
 8 objective to be

$$- \mathbf{E}_{z \sim N(0, I)} [\log d_u(g_v(z))]$$

9 which does not suffer from the small gradient problem.



10

11 **Synthesizing new images from a GAN** The generator samples la-
 12 tent factors $z \sim N(0, I)$ at test time to synthesize new images. The
 13 discriminator is not used at test time.

14 16.5 How to perform validation for a GAN?

15 For variational generative models, we can use the log-likelihood of
 16 synthesized images to obtain some understanding of whether the model
 17 is working well. If the log-likelihood of new images is similar to the
 18 log-likelihood of images in the training data then the new images are good
 19 at least as far as the model is concerned even if they may have perceptual
 20 artifacts.

21 Doing so is not so easy for implicit models because they do not output
 22 the likelihood of the generated samples. Run the generator a few times to
 23 eyeball the quality of images it generates.



1

2 But this is a very heuristic and qualitative metric.

3 **Frechet Inception Distance (FID)** A number of other metrics exist
 4 for evaluating generative models. One popular one is the so-called
 5 Frechet Inception Distance (FID) where we take any pre-trained model for
 6 classification, in this case people typically use the Inception architecture,
 7 and evaluate

$$\text{FID}(p, q) = \|\mu_p - \mu_q\|_2^2 + \text{trace} \left(\Sigma_p + \Sigma_q - 2(\Sigma_p \Sigma_q)^{1/2} \right).$$

8 where μ_p, Σ_p are the mean and covariace of the features of an Inception
 9 network when real images are fed to it and similarly μ_q, Σ_q are the
 10 mean/covariance of the features when GAN-generated images are fed to
 11 the same network.

12 The above formula is the Wasserstein distance between the two densities
 13 p, q , There are many similar techniques such as the Maximum Mean
 14 Discrepancy (MMD) that can be used to understand the discrepancy
 15 between the two distributions once the features are computed using some
 16 pre-trained model on their respective images.

17 Roughly speaking, the evaluation methodology in generative models,
 18 especially for images, is quite flawed. This is not a new phenomenon in
 19 machine learning/statistics because it is a intrinsically difficult problem to
 20 measure when two distributions are the same given only finite data from
 21 them. The problem is exacerbated in deep generative models because
 22 deep networks are very good at over-fitting, e.g., GANs can often end up
 23 memorizing the training data, they can generate very realistic images that
 24 are essentially the same as those in the training data. Nevertheless, a lot
 25 of techniques exist to make GANs synthesize high-quality images. See
 26 some examples at [Brock et al. \(2018\)](#); [Karras et al. \(2017\)](#).

The key behind the empirical success of GANs is to convert

a problem about estimating distributions, sampling from them etc. into a classification problem. Deep networks are extremely good at classification as compared to other problems like regression, reconstruction etc. and GANs leverage this remarkably. This is a trick that you will do well to remember when you use deep networks in the future: typically you will always get better results if you manage to rewrite your problem as a classification problem. I suspect the real reason for this is that we do not have good regularization techniques for deep networks for non-classification problems.

16.6 The zoo of GANs

Due to the numerous issues with GANs, there have been a large number of variants and attempts to improve their empirical performance. They fall mainly into the following categories.

1. Optimization tricks to train the generator-discriminator pair in a more stable fashion.
2. New loss functions that change the binary cross-entropy loss of the discriminator to something else. A majority of papers, including the example we saw above, fall into this category.
3. Characterizing the kind of critical points, equilibria of the training process; this is a similar line of analysis as the study of the energy landscape of deep networks for standard supervised learning.
4. Connections with variational inference suggest that GANs and their training techniques are essentially variational inference in disguise (Nowozin et al., 2016).
5. Coming up with new ways of evaluating generative models.

In addition to the above lines, there are many other novel and interesting applications such as Cycle-GANs and conditional-GANs.

Bibliography

- 2 Aizerman, M. A. (1964). Theoretical foundations of the potential function method in pattern recognition
3 learning. *Automation and remote control*, 25:821–837.
- 4 Alemi, A., Poole, B., Fischer, I., Dillon, J., Saurous, R. A., and Murphy, K. (2018). Fixing a broken elbo. In
5 *International Conference on Machine Learning*, pages 159–168. PMLR.
- 6 Amari, S. (1967). A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*,
7 (3):299–307.
- 8 Balasubramanian, V. (2015). Heterogeneity and Efficiency in the Brain. *Proceedings of the IEEE*, 103(8):1346–
9 1358.
- 10 Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples
11 without local minima. *Neural networks*, 2(1):53–58.
- 12 Bartlett, P. L., Harvey, N., Liaw, C., and Mehrabian, A. (2019). Nearly-tight vc-dimension and pseudodimension
13 bounds for piecewise linear neural networks. *J. Mach. Learn. Res.*, 20:63–1.
- 14 Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). Variational inference: A review for statisticians.
15 *Journal of the American Statistical Association*, 112(518):859–877.
- 16 Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436.
17 Springer.
- 18 Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *Siam*
19 *Review*, 60(2):223–311.
- 20 Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.
- 21 Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.
- 22 Brock, A., Donahue, J., and Simonyan, K. (2018). Large scale gan training for high fidelity natural image
23 synthesis. *arXiv preprint arXiv:1809.11096*.
- 24 Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J., Sagun, L., and
25 Zecchina, R. (2016). Entropy-sgd: Biasing gradient descent into wide valleys. *arXiv:1611.01838*.
- 26 Chaudhari, P. and Soatto, S. (2017). Stochastic gradient descent performs variational inference, converges to
27 limit cycles for deep networks. *arXiv preprint arXiv:1710.11029*.
- 28 Cortes, C. and Vapnik, V. (1995). Support vector machine. *Machine learning*, 20(3):273–297.
- 29 Efron, B. (1992). Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages
30 569–593. Springer.

- 1 Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition.
2 *Neural networks*, 1(2):119–130.
- 3 Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio,
4 Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages
5 2672–2680.
- 6 Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural
7 computation*, 18(7):1527–1554.
- 8 Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.
- 9 Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex.
10 *The Journal of physiology*, 195(1):215–243.
- 11 Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal
12 covariate shift. *arXiv preprint arXiv:1502.03167*.
- 13 Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., and Wilson, A. G. (2018). Averaging weights leads to
14 wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.
- 15 Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2017). Progressive growing of gans for improved quality,
16 stability, and variation. *arXiv preprint arXiv:1710.10196*.
- 17 Kawaguchi, K. (2016). Deep learning without poor local minima. In *Advances in neural information processing
18 systems*, pages 586–594.
- 19 Kidambi, R., Netrapalli, P., Jain, P., and Kakade, S. (2018). On the insufficiency of existing momentum
20 schemes for stochastic optimization. In *2018 Information Theory and Applications Workshop (ITA)*, pages
21 1–9. IEEE.
- 22 Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- 23 Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.
- 24 Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural
25 networks. In *Advances in neural information processing systems*, pages 1097–1105.
- 26 Kushner, H. and Yin, G. G. (2003). *Stochastic approximation and recursive algorithms and applications*,
27 volume 35. Springer Science & Business Media.
- 28 Le Roux, N., Schmidt, M. W., Bach, F. R., et al. (2012). A stochastic gradient method with an exponential
29 convergence rate for finite training sets. In *NIPS*, pages 2672–2680.
- 30 LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.
- 31 LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989).
32 Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.
- 33 LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document
34 recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- 35 Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. (2018). Visualizing the loss landscape of neural nets. In
36 *Advances in Neural Information Processing Systems*, pages 6389–6399.

- 1 Liu, C. and Belkin, M. (2018). Mass: an accelerated stochastic method for over-parametrized learning. *arXiv*
2 *preprint arXiv:1810.13395*.
- 3 Lopez-Paz, D. and Oquab, M. (2016). Revisiting classifier two-sample tests. *arXiv preprint arXiv:1610.06545*.
- 4 McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The*
5 *bulletin of mathematical biophysics*, 5(4):115–133.
- 6 Mescheder, L., Nowozin, S., and Geiger, A. (2017). The numerics of gans. In *Advances in Neural Information*
7 *Processing Systems*, pages 1825–1835.
- 8 Minsky, M. and Papert, S. A. (2017). *Perceptrons: An introduction to computational geometry*. MIT press.
- 9 Nowozin, S., Cseke, B., and Tomioka, R. (2016). f-gan: Training generative neural samplers using variational
10 divergence minimization. In *Advances in neural information processing systems*, pages 271–279.
- 11 Pedro, D. (2000). A unified bias-variance decomposition and its applications. In *17th International Conference*
12 *on Machine Learning*, pages 231–238.
- 13 Pickering, A. (2010). *The cybernetic brain: Sketches of another future*. University of Chicago Press.
- 14 Polyak, B. T. (1990). A new method of stochastic approximation type. *Avtomatika i telemekhanika*, (7):98–107.
- 15 Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal*
16 *on Control and Optimization*, 30(4):838–855.
- 17 Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin,
18 P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. In
19 *International conference on machine learning*, pages 8748–8763. PMLR.
- 20 Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *Advances in neural*
21 *information processing systems*, pages 1177–1184.
- 22 Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics
23 processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880.
- 24 Razavi, A., van den Oord, A., and Vinyals, O. (2019). Generating diverse high-fidelity images with vq-vae-2.
25 In *Advances in Neural Information Processing Systems*, pages 14866–14876.
- 26 Recht, B. and Ré, C. (2012). Beneath the valley of the noncommutative arithmetic-geometric mean inequality:
27 conjectures, case-studies, and consequences. *arXiv preprint arXiv:1202.4184*.
- 28 Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*,
29 pages 400–407.
- 30 Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the
31 brain. *Psychological review*, 65(6):386.
- 32 Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error
33 propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.
- 34 Ruppert, D. (1988). Efficient estimations from a slowly convergent robbins-monro process. Technical report,
35 Cornell University Operations Research and Industrial Engineering.
- 36 Salakhutdinov, R. and Larochelle, H. (2010). Efficient learning of deep boltzmann machines. In *Proceedings*
37 *of the thirteenth international conference on artificial intelligence and statistics*, pages 693–700.

-
- 1 Scholkopf, B. and Smola, A. J. (2018). *Learning with kernels: support vector machines, regularization,*
2 *optimization, and beyond*. Adaptive Computation and Machine Learning series.
- 3 Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all
4 convolutional net. *arXiv:1412.6806*.
- 5 Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way
6 to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.
- 7 Sukthanker, R. S., Huang, Z., Kumar, S., Timofte, R., and Van Gool, L. (2022). Generative flows with invertible
8 attentions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages
9 11234–11243.
- 10 Transtrum, M. K., Machta, B. B., and Sethna, J. P. (2011). Geometry of nonlinear least squares with applications
11 to sloppy models and optimization. *Physical Review E*, 83(3):036701.
- 12 Turing, A. M. (2009). Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65.
13 Springer.
- 14 Vapnik, V. (2013). *The nature of statistical learning theory*. Springer science & business media.
- 15 Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I.
16 (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.
- 17 Wiener, N. (1965). *Cybernetics or Control and Communication in the Animal and the Machine*, volume 25.
18 MIT press.
- 19 Yang, R., Mao, J., and Chaudhari, P. (2022). Does the data induce capacity control in deep learning? In
20 *International Conference on Machine Learning*, pages 25166–25197.
- 21 Zaheer, M., Kottur, S., Ravanbakhsh, S., Póczos, B., Salakhutdinov, R. R., and Smola, A. J. (2017). Deep sets.
22 *Advances in neural information processing systems*, 30.