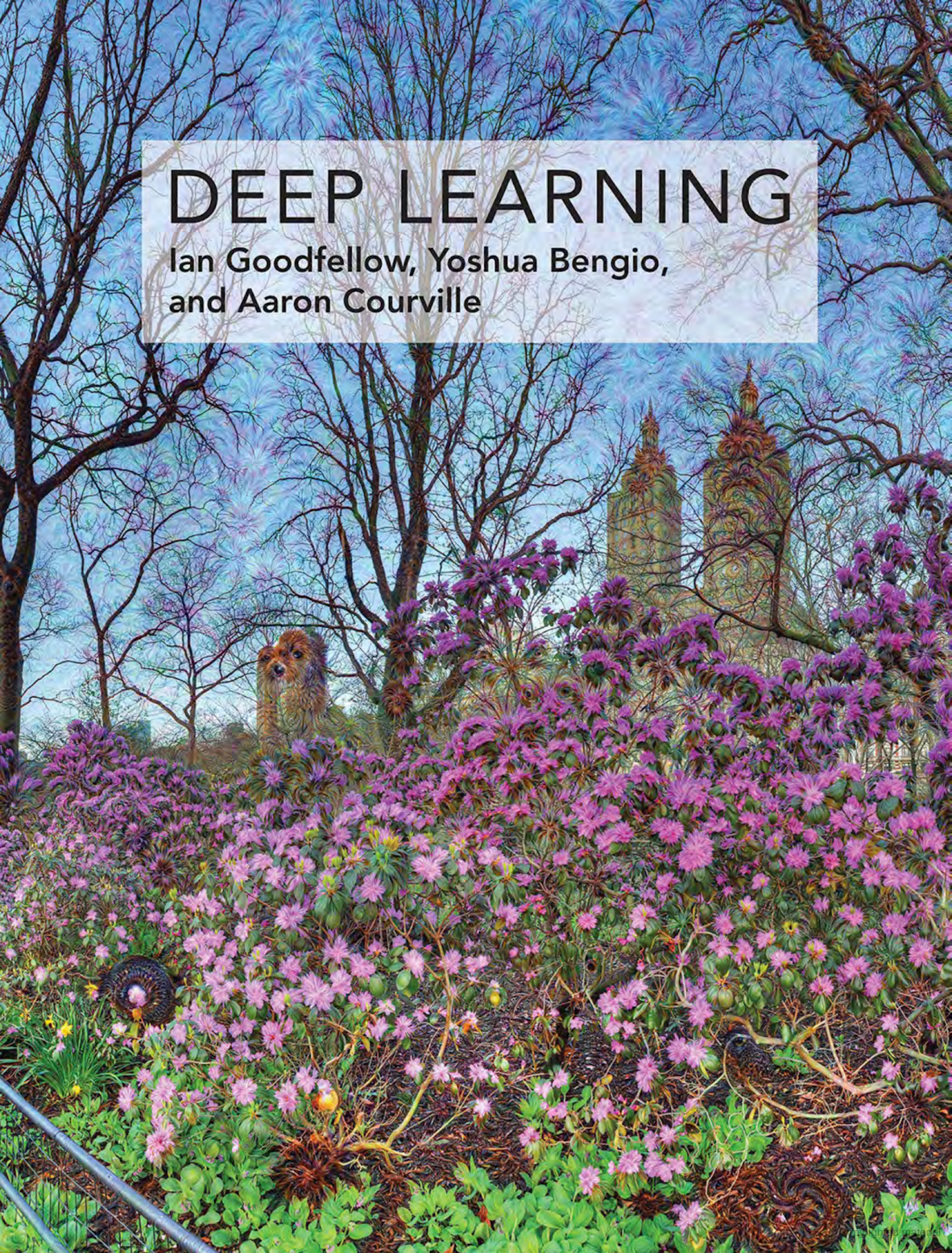
DEEP LEARNING

Ian Goodfellow, Yoshua Bengio,
and Aaron Courville



Deep Learning

Ian Goodfellow
Yoshua Bengio and
Aaron Courville

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in SFRM1095 by diacriTech, Chennai.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Goodfellow, Ian, author. | Bengio, Yoshua, author. | Courville, Aaron, author.
Title: Deep learning / Ian Goodfellow, Yoshua Bengio, and Aaron Courville.
Description: Cambridge, MA : MIT Press, [2017] | Series: Adaptive computation and machine learning series | Includes bibliographical references and index.
Identifiers: LCCN 2016022992 | ISBN 9780262035613 (hardcover : alk. paper)
Subjects: LCSH: Machine learning,
Classification: LCC Q325.5 .G66 2017 | DDC 006.3/1–dc23 LC record available at https://lccn.loc.gov/2016022992

10 9 8 7 6 5 4 3 2 1

Contents

Website

www.deeplearningbook.org

This book is accompanied by the above website. The website provides a variety of supplementary material, including exercises, lecture slides, corrections of mistakes, and other resources that should be useful to both readers and instructors.

Acknowledgments

This book would not have been possible without the contributions of many people.

We would like to thank those who commented on our proposal for the book and helped plan its contents and organization: Guillaume Alain, Kyunghyun Cho, Çağlar Gülçehre, David Krueger, Hugo Larochelle, Razvan Pascanu and Thomas Rohée.

We would like to thank the people who offered feedback on the content of the book itself. Some offered feedback on many chapters: Martín Abadi, Guillaume Alain, Ion Androutsopoulos, Fred Bertsch, Olexa Bilaniuk, Ufuk Can Biçici, Matko Bošnjak, John Boersma, Greg Brockman, Alexandre de Brébisson, Pierre Luc Carrier, Sarath Chandar, Pawel Chilinski, Mark Daoust, Oleg Dashevskii, Laurent Dinh, Stephan Dreseitl, Jim Fan, Miao Fan, Meire Fortunato, Frédéric Francis, Nando de Freitas, Çağlar Gülçehre, Jurgen Van Gael, Javier Alonso García, Jonathan Hunt, Gopi Jeyaram, Chingiz Kabytayev, Lukasz Kaiser, Varun Kanade, Asifullah Khan, Akiel Khan, John King, Diederik P. Kingma, Yann LeCun, Rudolf Mathey, Matías Mattamala, Abhinav Maurya, Kevin Murphy, Oleg Mürk, Roman Novak, Augustus Q. Odena, Simon Pavlik, Karl Pichotta, Eddie Pierce, Kari Pulli, Roussel Rahman, Tapani Raiko, Anurag Ranjan, Johannes Roith, Mihaela Rosca, Halis Sak, César Salgado, Grigory Sapunov, Yoshinori Sasaki, Mike Schuster, Julian Serban, Nir Shabat, Ken Shirriff, Andre Simpelo, David Slate, Scott Stanley, David Sussillo, Ilya Sutskever, Carles Gelada Sáez, Graham Taylor, Valentin Tolmer, Massimiliano Tomassoli, An Tran, Shubhendu Trivedi, Alexey Umnov, Vincent Vanhoucke, Marco Visentini-Scarzanella, Martin Vita, David Warde-Farley, Dustin Webb, Kelvin Xu, Wei Xue, Ke Yang, Li Yao, Zygmunt Zając and Ozan Çağlayan.

We would also like to thank those who provided us with useful feedback on individual chapters:

- Notation: Zhang Yuanhang.

- Chapter 1, Introduction: Yusuf Akgul, Sebastien Bratieres, Samira Ebrahimi, Charlie Gorichanaz, Brendan Loudermilk, Eric Morris, Cosmin Pârvulescu and Alfredo Solano.

- Chapter 2, Linear Algebra: Amjad Almahairi, Nikola Banić, Kevin Bennett, Philippe Castonguay, Oscar Chang, Eric Fosler-Lussier, Andrey Khalyavin, Sergey Oreshkov, István Petrás, Dennis Prangle, Thomas Rohée, Gitanjali Gulve Sehgal, Colby Toland, Alessandro Vitale and Bob Welland.

- Chapter 3, Probability and Information Theory: John Philip Anderson, Kai Arulkumaran, Vincent Dumoulin, Rui Fa, Stephan Gouws, Artem Oboturov, Antti Rasmus, Alexey Surkov and Volker Tresp.

- Chapter 4, Numerical Computation: Tran Lam AnIan Fischer and Hu Yuhuang.

- Chapter 5, Machine Learning Basics: Dzmitry Bahdanau, Justin Domingue, Nikhil Garg, Makoto Otsuka, Bob Pepin, Philip Popien, Bharat Prabhakar, Emmanuel Rayner, Peter Shepard, Kee-Bong Song, Zheng Sun and Andy Wu.

- Chapter 6, Deep Feedforward Networks: Uriel Berdugo, Fabrizio Bottarel, Elizabeth Burl, Ishan Durugkar, Jeff Hlywa, Jong Wook Kim, David Krueger, Aditya Kumar Praharaj and Sten Sootla.

- Chapter 7, Regularization for Deep Learning: Morten Kolbæk, Kshitij Lauria, Inkyu Lee, Sunil Mohan, Hai Phong Phan and Joshua Salisbury.

- Chapter 8, Optimization for Training Deep Models: Marcel Ackermann, Peter Armitage, Rowel Atienza, Andrew Brock, Tegan Maharaj, James Martens, Mostafa Nategh, Kashif Rasul, Klaus Strobl and Nicholas Turner.

- Chapter 9, Convolutional Networks: Martín Arjovsky, Eugene Brevdo, Konstantin Divilov, Eric Jensen, Mehdi Mirza, Alex Paino, Marjorie Sayer, Ryan Stout and Wentao Wu.

- Chapter 10, Sequence Modeling: Recurrent and Recursive Nets: Gökçen Eraslan, Steven Hickson, Razvan Pascanu, Lorenzo von Ritter, Rui Rodrigues, Dmitriy Serdyuk, Dongyu Shi and Kaiyu Yang.

- Chapter 11, Practical Methodology: Daniel Beckstein.

- Chapter 12, Applications: George Dahl, Vladimir Nekrasov and Ribana Roscher.

- Chapter 13, Linear Factor Models: Jayanth Koushik.

- Chapter 15, Representation Learning: Kunal Ghosh.
- Chapter 16, Structured Probabilistic Models for Deep Learning: Minh Lê and Anton Varfolom.
- Chapter 18, Confronting the Partition Function: Sam Bowman.
- Chapter 19, Approximate Inference: Yujia Bao.
- Chapter 20, Deep Generative Models: Nicolas Chapados, Daniel Galvez, Wenming Ma, Fady Medhat, Shakir Mohamed and Grégoire Montavon.
- Bibliography: Lukas Michelbacher and Leslie N. Smith.

We also want to thank those who allowed us to reproduce images, figures or data from their publications. We indicate their contributions in the figure captions throughout the text.

We would like to thank Lu Wang for writing pdf2htmlEX, which we used to make the web version of the book, and for offering support to improve the quality of the resulting HTML.

We would like to thank Ian's wife Daniela Flori Goodfellow for patiently supporting Ian during the writing of the book as well as for help with proofreading.

We would like to thank the Google Brain team for providing an intellectual environment where Ian could devote a tremendous amount of time to writing this book and receive feedback and guidance from colleagues. We would especially like to thank Ian's former manager, Greg Corrado, and his current manager, Samy Bengio, for their support of this project. Finally, we would like to thank Geoffrey Hinton for encouragement when writing was difficult.

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor $\boldsymbol{\mathsf{A}}$
$\boldsymbol{\mathsf{A}}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Calculus

$\dfrac{dy}{dx}$	Derivative of y with respect to x
$\dfrac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathsf{\mathbf{X}}} y$	Tensor containing derivatives of y with respect to $\mathsf{\mathbf{X}}$
$\dfrac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$\mathrm{a} \bot \mathrm{b}$	The random variables a and b are independent
$\mathrm{a} \bot \mathrm{b} \mid \mathrm{c}$	They are conditionally independent given c
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E} f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \Vert Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \rightarrow \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x};\boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\zeta(x)$	Softplus, $\log(1+\exp(x))$
$\lvert\lvert\boldsymbol{x}\rvert\rvert_p$	L^p norm of $\boldsymbol{x}$
$\lvert\lvert\boldsymbol{x}\rvert\rvert$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0,x)$
$\mathbf{1}_{\mathrm{condition}}$	is 1 if the condition is true, 0 otherwise

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathsf{\mathbf{X}})$. This denotes the application of f to the array element-wise. For example, if $\mathsf{\mathbf{C}} = \sigma(\mathsf{\mathbf{X}})$, then $\mathsf{C}_{i,j,k} = \sigma(\mathsf{X}_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\mathrm{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$

1

Introduction

Inventors have long dreamed of creating machines that think. This desire dates back to at least the time of ancient Greece. The mythical figures Pygmalion, Daedalus, and Hephaestus may all be interpreted as legendary inventors, and Galatea, Talos, and Pandora may all be regarded as artificial life (Ovid and Martin, 2004; Sparkes, 1996; Tandy, 1997).

When programmable computers were first conceived, people wondered whether such machines might become intelligent, over a hundred years before one was built (Lovelace, 1842). Today, **artificial intelligence** (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research.

In the early days of artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers—problems that can be described by a list of formal, mathematical rules. The true challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images.

This book is about a solution to these more intuitive problems. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all the knowledge that the computer needs. The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI **deep learning**.

Many of the early successes of AI took place in relatively sterile and formal environments and did not require computers to have much knowledge about the world. For example, IBM's Deep Blue chess-playing system defeated world champion Garry Kasparov in 1997 (Hsu, 2002). Chess is of course a very simple world, containing only sixty-four locations and thirty-two pieces that can move in only rigidly circumscribed ways. Devising a successful chess strategy is a tremendous accomplishment, but the challenge is not due to the difficulty of describing the set of chess pieces and allowable moves to the computer. Chess can be completely described by a very brief list of completely formal rules, easily provided ahead of time by the programmer.

Ironically, abstract and formal tasks that are among the most difficult mental undertakings for a human being are among the easiest for a computer. Computers have long been able to defeat even the best human chess player but only recently have begun matching some of the abilities of average human beings to recognize objects or speech. A person's everyday life requires an immense amount of knowledge about the world. Much of this knowledge is subjective and intuitive, and therefore difficult to articulate in a formal way. Computers need to capture this same knowledge in order to behave in an intelligent way. One of the key challenges in artificial intelligence is how to get this informal knowledge into a computer.

Several artificial intelligence projects have sought to hard-code knowledge about the world in formal languages. A computer can reason automatically about statements in these formal languages using logical inference rules. This is known as the **knowledge base** approach to artificial intelligence. None of these projects has led to a major success. One of the most famous such projects is Cyc (Lenat and Guha, 1989). Cyc is an inference engine and a database of statements in a language called CycL. These statements are entered by a staff of human supervisors. It is an unwieldy process. People struggle to devise formal rules with enough complexity to accurately describe the world. For example, Cyc failed to understand a story about a person named Fred shaving in the morning (Linde, 1992). Its inference engine detected an inconsistency in the story: it knew that people do not have electrical parts, but because Fred was holding an electric razor, it believed the entity "FredWhileShaving" contained electrical parts. It therefore asked whether Fred was still a person while he was shaving.

The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as **machine learning**. The objects or unobserved forces in the physical world that affect observable quantities. They may also exist as constructs in the human mind that provide useful simplifying explanations or inferred causes of the observed data. They can be thought of as concepts or abstractions that help us make sense of the rich variability in the data. When analyzing a speech recording, the factors of variation include the speaker's age, their sex, their accent and the words they are speaking. When analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun.

A major source of difficulty in many real-world artificial intelligence applications is that many of the factors of variation influence every single piece of data we are able to observe. The individual pixels in an image of a red car might be very close to black at night. The shape of the car's silhouette depends on the viewing angle. Most applications require us to *disentangle* the factors of variation and discard the ones that we do not care about.

Of course, it can be very difficult to extract such high-level, abstract features from raw data. Many of these factors of variation, such as a speaker's accent, can be identified only using sophisticated, nearly human-level understanding of the data. When it is nearly as difficult to obtain a representation as to solve the original problem, representation learning does not, at first glance, seem to help us.

Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning enables the computer to build complex concepts out of simpler concepts. Figure 1.2 shows how a deep learning system can represent the concept of an image of a person by combining simpler concepts, such as corners and contours, which are in turn defined in terms of edges.

The quintessential example of a deep learning model is the feedforward deep network, or **multilayer perceptron** (MLP). A multilayer perceptron is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions. We can think of each application of a different mathematical function as providing a new representation of the input.

The idea of learning the right representation for the data provides one perspective on deep learning. Another perspective on deep learning is that depth enables the computer to learn a multistep computer program. Each layer of the representation can be thought of as the state of the computer's memory after executing another set of instructions in parallel. Networks with greater depth can execute more instructions in sequence. Sequential instructions offer great power because later instructions can refer back to the results of earlier instructions.

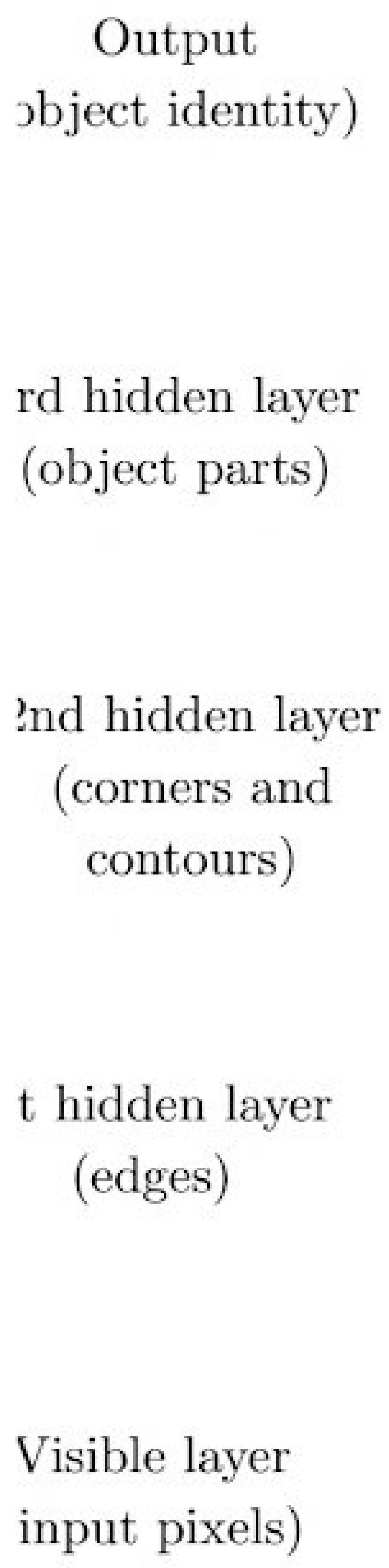


Figure 1.2: Illustration of a deep learning model. It is difficult for a computer to understand the meaning of raw sensory input data, such as this image represented as a collection of pixel values. The function mapping from a set of pixels to an object identity is very complicated. Learning or evaluating this mapping seems insurmountable if tackled directly. Deep learning resolves this difficulty by breaking the desired complicated mapping into a series of nested simple mappings, each described by a different layer of the model. The input is presented at the **visible layer**, so named because it contains the variables that we are able to observe. Then a series of **hidden layers** extracts increasingly abstract features from the image. These layers are called "hidden" because their values are not given in the data; instead the model must determine which concepts are useful for explaining the relationships in the observed data. The images here are visualizations of the kind of feature represented by each hidden unit. Given the pixels, the first layer can easily identify edges, by comparing the brightness of neighboring pixels. Given the first hidden layer's description of the edges, the second hidden layer can easily search for corners and extended contours, which are recognizable as collections of edges. Given the second hidden layer's description of the image in terms of corners and contours, the third hidden layer can detect entire parts of specific objects, by finding specific collections of contours and corners. Finally, this description of the image in terms of the object parts it contains can be used to recognize the objects present in the image. Images reproduced with permission from Zeiler and Fergus (2014).

According to this view of deep learning, not all the information in a layer's activations necessarily encodes factors of variation that explain the input. The representation also stores state information that helps to execute a program that can make sense of the input. This state information could be analogous to a counter or pointer in a traditional computer program. It has nothing to do with the content of the input specifically, but it helps the model to organize its processing.

There are two main ways of measuring the depth of a model. The first view is based on the number of sequential instructions that must be executed to evaluate the architecture. We can think of this as the length of the longest path through a flow chart that describes how to compute each of the model's outputs given its inputs. Just as two equivalent computer programs will have different lengths depending on which language the program is written in, the same function may be drawn as a flowchart with different depths depending on which functions we allow to be used as individual steps in the flowchart. Figure 1.3 illustrates how this choice of language can give two different measurements for the same architecture.

Another approach, used by deep probabilistic models, regards the depth of a model as being not the depth of the computational graph but the depth of the graph describing how concepts are related to each other. In this case, the depth

Figure 1.3: Illustration of computational graphs mapping an input to an output where each node performs an operation. Depth is the length of the longest path from input to output but depends on the definition of what constitutes a possible computational step. The computation depicted in these graphs is the output of a logistic regression model, $\sigma(\boldsymbol{w}^T\boldsymbol{x})$, where σ is the logistic sigmoid function. If we use addition, multiplication and logistic sigmoids as the elements of our computer language, then this model has depth three. If we view logistic regression as an element itself, then this model has depth one.

of the flowchart of the computations needed to compute the representation of each concept may be much deeper than the graph of the concepts themselves. This is because the system's understanding of the simpler concepts can be refined given information about the more complex concepts. For example, an AI system observing an image of a face with one eye in shadow may initially see only one eye. After detecting that a face is present, the system can then infer that a second eye is probably present as well. In this case, the graph of concepts includes only two layers—a layer for eyes and a layer for faces—but the graph of computations includes $2n$ layers if we refine our estimate of each concept given the other n times.

Because it is not always clear which of these two views—the depth of the computational graph, or the depth of the probabilistic modeling graph—is most relevant, and because different people choose different sets of smallest elements from which to construct their graphs, there is no single correct value for the depth of an architecture, just as there is no single correct value for the length of a computer program. Nor is there a consensus about how much depth a model requires to qualify as "deep." However, deep learning can be safely regarded as the study of models that involve a greater amount of composition of either learned functions or learned concepts than traditional machine learning does.

To summarize, deep learning, the subject of this book, is an approach to AI. Specifically, it is a type of machine learning, a technique that enables computer systems to improve with experience and data. We contend that machine learning is the only viable approach to building AI systems that can operate in complicated real-world environments. Deep learning is a particular kind of machine learning that achieves great power and flexibility by representing the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones. Figure 1.4 illustrates the relationship between these different AI disciplines. Figure 1.5 gives a high-level schematic of how each works.

1.1 Who Should Read This Book?

This book can be useful for a variety of readers, but we wrote it with two target audiences in mind. One of these target audiences is university students (undergraduate or graduate) learning about machine learning, including those who are beginning a career in deep learning and artificial intelligence research. The other target audience is software engineers who do not have a machine learning or statistics background but want to rapidly acquire one and begin using deep learning in their product or platform. Deep learning has already proved useful in many soft-

Figure 1.4: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

ware disciplines, including computer vision, speech and audio processing, natural language processing, robotics, bioinformatics and chemistry, video games, search engines, online advertising and finance.

This book has been organized into three parts to best accommodate a variety of readers. Part I introduces basic mathematical tools and machine learning concepts. Part II describes the most established deep learning algorithms, which are essentially solved technologies. Part III describes more speculative ideas that are widely believed to be important for future research in deep learning.

Readers should feel free to skip parts that are not relevant given their interests or background. Readers familiar with linear algebra, probability, and fundamental machine learning concepts can skip part I, for example, while those who just want

We do assume that all readers come from a computer science background. We assume familiarity with programming, a basic understanding of computational performance issues, complexity theory, introductory level calculus and some of the terminology of graph theory.

1.2 Historical Trends in Deep Learning

It is easiest to understand deep learning with some historical context. Rather than providing a detailed history of deep learning, we identify a few key trends:

- Deep learning has had a long and rich history, but has gone by many names, reflecting different philosophical viewpoints, and has waxed and waned in popularity.
- Deep learning has become more useful as the amount of available training data has increased.
- Deep learning models have grown in size over time as computer infrastructure (both hardware and software) for deep learning has improved.
- Deep learning has solved increasingly complicated applications with increasing accuracy over time.

1.2.1 The Many Names and Changing Fortunes of Neural Networks

We expect that many readers of this book have heard of deep learning as an exciting new technology, and are surprised to see a mention of "history" in a book about an emerging field. In fact, deep learning dates back to the 1940s. Deep learning only *appears* to be new, because it was relatively unpopular for several years preceding its current popularity, and because it has gone through many different names, only recently being called "deep learning." The field has been rebranded many times, reflecting the influence of different researchers and different perspectives.

A comprehensive history of deep learning is beyond the scope of this textbook. Some basic context, however, is useful for understanding deep learning. Broadly speaking, there have been three waves of development: deep learning known as **cybernetics** in the 1940s–1960s, deep learning known as **connectionism** in the 1980s–1990s, and the current resurgence under the name deep learning beginning in 2006. This is quantitatively illustrated in figure 1.7.

Figure 1.7: Two of the three historical waves of artificial neural nets research, as measured by the frequency of the phrases "cybernetics" and "connectionism" or "neural networks," according to Google Books (the third wave is too recent to appear). The first wave started with cybernetics in the 1940s–1960s, with the development of theories of biological learning (McCulloch and Pitts, 1943; Hebb, 1949) and implementations of the first models, such as the perceptron (Rosenblatt, 1958), enabling the training of a single neuron. The second wave started with the connectionist approach of the 1980–1995 period, with back-propagation (Rumelhart et al., 1986a) to train a neural network with one or two hidden layers. The current and third wave, deep learning, started around 2006 (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007a) and is just now appearing in book form as of 2016. The other two waves similarly appeared in book form much later than the corresponding scientific activity occurred.

Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, that is, models of how learning happens or could happen in the brain. As a result, one of the names that deep learning has gone by is **artificial neural networks** (ANNs). The corresponding perspective on deep learning models is that they are engineered systems inspired by the biological brain (whether the human brain or the brain of another animal). While the kinds of neural networks used for machine learning have sometimes been used to understand brain function (Hinton and Shallice, 1991), they are generally not designed to be realistic models of biological function. The neural perspective on deep learning is motivated by two main ideas. One idea is that the brain provides a proof by example that intelligent behavior is possible, and a conceptually straightforward path to building intelligence is to reverse engineer the computational principles behind the brain and duplicate its functionality. Another perspective is that it would be deeply interesting to understand the brain and the principles that underlie human intelligence, so machine learning models that shed light on these basic scientific questions are useful apart from their ability to solve engineering applications.

The modern term “deep learning” goes beyond the neuroscientific perspective on the current breed of machine learning models. It appeals to a more general principle of learning *multiple levels of composition*, which can be applied in machine learning frameworks that are not necessarily neurally inspired.

The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective. These models were designed to take a set of n input values $x_1, \ldots, x_n$ and associate them with an output y. These models would learn a set of weights $w_1, \ldots, w_n$ and compute their output $f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + \cdots + x_n w_n$. This first wave of neural networks research was known as cybernetics, as illustrated in figure 1.7.

The McCulloch-Pitts neuron (McCulloch and Pitts, 1943) was an early model of brain function. This linear model could recognize two different categories of inputs by testing whether $f(\boldsymbol{x}, \boldsymbol{w})$ is positive or negative. Of course, for the model to correspond to the desired definition of the categories, the weights needed to be set correctly. These weights could be set by the human operator. In the 1950s, the perceptron (Rosenblatt, 1958, 1962) became the first model that could learn the weights that defined the categories given examples of inputs from each category. The **adaptive linear element** (ADALINE), which dates from about the same time, simply returned the value of $f(\boldsymbol{x})$ itself to predict a real number (Widrow and Hoff, 1960) and could also learn to predict these numbers from data.

These simple learning algorithms greatly affected the modern landscape of machine learning. The training algorithm used to adapt the weights of the ADALINE was a special case of an algorithm called **stochastic gradient descent**. Slightly modified versions of the stochastic gradient descent algorithm remain the dominant training algorithms for deep learning models today.

Models based on the $f(\boldsymbol{x}, \boldsymbol{w})$ used by the perceptron and ADALINE are called **linear models**. These models remain some of the most widely used machine learning models, though in many cases they are *trained* in different ways than the original models were trained.

Linear models have many limitations. Most famously, they cannot learn the XOR function, where $f([0, 1], \boldsymbol{w}) = 1$ and $f([1, 0], \boldsymbol{w}) = 1$ but $f([1, 1], \boldsymbol{w}) = 0$ and $f([0, 0], \boldsymbol{w}) = 0$. Critics who observed these flaws in linear models caused a backlash against biologically inspired learning in general (Minsky and Papert, 1969). This was the first major dip in the popularity of neural networks.

Today, neuroscience is regarded as an important source of inspiration for deep learning researchers, but it is no longer the predominant guide for the field.

The main reason for the diminished role of neuroscience in deep learning research today is that we simply do not have enough information about the brain to use it as a guide. To obtain a deep understanding of the actual algorithms used by the brain, we would need to be able to monitor the activity of (at the very least) thousands of interconnected neurons simultaneously. Because we are not able to do this, we are far from understanding even some of the most simple and well-studied parts of the brain (Olshausen and Field, 2005).

Neuroscience has given us a reason to hope that a single deep learning algorithm can solve many different tasks. Neuroscientists have found that ferrets can learn to "see" with the auditory processing region of their brain if their brains are rewired to send visual signals to that area (Von Melchner et al., 2000). This suggests that much of the mammalian brain might use a single algorithm to solve most of the different tasks that the brain solves. Before this hypothesis, machine learning research was more fragmented, with different communities of researchers studying natural language processing, vision, motion planning and speech recognition. Today, these application communities are still separate, but it is common for deep learning research groups to study many or even all these application areas simultaneously.

We are able to draw some rough guidelines from neuroscience. The basic idea of having many computational units that become intelligent only via their interactions with each other is inspired by the brain. The neocognitron (Fukushima, 1980) introduced a powerful model architecture for processing images that was inspired by the structure of the mammalian visual system and later became the basis for the modern convolutional network (LeCun et al., 1998b), as we will see in section 9.10. Most neural networks today are based on a model neuron called the **rectified linear unit**. The original cognitron (Fukushima, 1975) introduced a more complicated version that was highly inspired by our knowledge of brain function. The simplified modern version was developed incorporating ideas from many viewpoints, with Nair and Hinton (2010) and Glorot et al. (2011a) citing neuroscience as an influence, and Jarrett et al. (2009) citing more engineering-oriented influences. While neuroscience is an important source of inspiration, it need not be taken as a rigid guide. We know that actual neurons compute very different functions than modern rectified linear units, but greater neural realism has not yet led to an improvement in machine learning performance. Also, while neuroscience has successfully inspired several neural network *architectures*, we do not yet know enough about biological learning for neuroscience to offer much guidance for the *learning algorithms* we use to train these architectures.

Media accounts often emphasize the similarity of deep learning to the brain. While it is true that deep learning researchers are more likely to cite the brain as an influence than researchers working in other machine learning fields, such as kernel machines or Bayesian statistics, one should not view deep learning as an attempt to simulate the brain. Modern deep learning draws inspiration from many fields, especially applied math fundamentals like linear algebra, probability, information theory, and numerical optimization. While some deep learning researchers cite neuroscience as an important source of inspiration, others are not concerned with neuroscience at all.

It is worth noting that the effort to understand how the brain works on an algorithmic level is alive and well. This endeavor is primarily known as "computational neuroscience" and is a separate field of study from deep learning. It is common for researchers to move back and forth between both fields. The field of deep learning is primarily concerned with how to build computer systems that are able to successfully solve tasks requiring intelligence, while the field of computational neuroscience is primarily concerned with building more accurate models of how the brain actually works.

In the 1980s, the second wave of neural network research emerged in great part via a movement called **connectionism**, or **parallel distributed processing** (Rumelhart et al., 1986c; McClelland et al., 1995). Connectionism arose in the context of cognitive science. Cognitive science is an interdisciplinary approach to understanding the mind, combining multiple different levels of analysis. During the early 1980s, most cognitive scientists studied models of symbolic reasoning. Despite their popularity, symbolic models were difficult to explain in terms of how the brain could actually implement them using neurons. The connectionists began to study models of cognition that could actually be grounded in neural implementations (Touretzky and Minton, 1985), reviving many ideas dating back to the work of psychologist Donald Hebb in the 1940s (Hebb, 1949).

The central idea in connectionism is that a large number of simple computational units can achieve intelligent behavior when networked together. This insight applies equally to neurons in biological nervous systems as it does to hidden units in computational models.

Several key concepts arose during the connectionism movement of the 1980s that remain central to today's deep learning.

One of these concepts is that of **distributed representation** (Hinton et al., 1986). This is the idea that each input to a system should be represented by many features, and each feature should be involved in the representation of many possible inputs. For example, suppose we have a vision system that can recognize cars, trucks, and birds, and these objects can each be red, green, or blue. One way of representing these inputs would be to have a separate neuron or hidden unit important new development is that today we can provide these algorithms with the resources they need to succeed. Figure 1.8 shows how the size of benchmark datasets has expanded remarkably over time. This trend is driven by the increasing digitization of society. As more and more of our activities take place on computers, more and more of what we do is recorded. As our computers are increasingly networked together, it becomes easier to centralize these records and curate them into a dataset appropriate for machine learning applications. The age of "Big Data" has made machine learning much easier because the key burden of statistical estimation—generalizing well to new data after observing only a small amount

Figure 1.8: Increasing dataset size over time. In the early 1900s, statisticians studied datasets using hundreds or thousands of manually compiled measurements (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). In the 1950s through the 1980s, the pioneers of biologically inspired machine learning often worked with small synthetic datasets, such as low-resolution bitmaps of letters, that were designed to incur low computational cost and demonstrate that neural networks were able to learn specific kinds of functions (Widrow and Hoff, 1960; Rumelhart et al., 1986b). In the 1980s and 1990s, machine learning became more statistical and began to leverage larger datasets containing tens of thousands of examples, such as the MNIST dataset (shown in figure 1.9) of scans of handwritten numbers (LeCun et al., 1998b). In the first decade of the 2000s, more sophisticated datasets of this same size, such as the CIFAR-10 dataset (Krizhevsky and Hinton, 2009), continued to be produced. Toward the end of that decade and throughout the first half of the 2010s, significantly larger datasets, containing hundreds of thousands to tens of millions of examples, completely changed what was possible with deep learning. These datasets included the public Street View House Numbers dataset (Netzer et al., 2011), various versions of the ImageNet dataset (Deng et al., 2009, 2010a; Russakovsky et al., 2014a), and the Sports-1M dataset (Karpathy et al., 2014). At the top of the graph, we see that datasets of translated sentences, such as IBM's dataset constructed from the Canadian Hansard (Brown et al., 1990) and the WMT 2014 English to French dataset (Schwenk, 2014), are typically far ahead of other dataset sizes.

Figure 1.9: Example inputs from the MNIST dataset. The "NIST" stands for National Institute of Standards and Technology, the agency that originally collected this data. The "M" stands for "modified," since the data has been preprocessed for easier use with machine learning algorithms. The MNIST dataset consists of scans of handwritten digits and associated labels describing which digit 0–9 is contained in each image. This simple classification problem is one of the simplest and most widely used tests in deep learning research. It remains popular despite being quite easy for modern techniques to solve. Geoffrey Hinton has described it as "the *drosophila* of machine learning," meaning that it enables machine learning researchers to study their algorithms in controlled laboratory conditions, much as biologists often study fruit flies.

of data—has been considerably lightened. As of 2016, a rough rule of thumb is that a supervised deep learning algorithm will generally achieve acceptable performance with around 5,000 labeled examples per category and will match or exceed human performance when trained with a dataset containing at least 10 million labeled examples. Working successfully with datasets smaller than this is an important research area, focusing in particular on how we can take advantage of large quantities of unlabeled examples, with unsupervised or semi-supervised learning.

1.2.3 Increasing Model Sizes

Another key reason that neural networks are wildly successful today after enjoying comparatively little success since the 1980s is that we have the computational resources to run much larger models today. One of the main insights of connectionism is that animals become intelligent when many of their neurons work together. An individual neuron or small collection of neurons is not particularly useful.

Biological neurons are not especially densely connected. As seen in figure 1.10, our machine learning models have had a number of connections per neuron within an order of magnitude of even mammalian brains for decades.

In terms of the total number of neurons, neural networks have been astonishingly small until quite recently, as shown in figure 1.11. Increasing neural network size over time. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. This growth is driven by faster computers with larger memory and by the availability of larger datasets. Larger networks are able to achieve higher accuracy on more complex tasks. This trend looks set to continue for decades. Unless new technologies enable faster scaling, artificial neural networks will not have the same number of neurons as the human brain until at least the 2050s. Biological neurons may represent more complicated functions than current artificial neurons, so biological neural networks may be even larger than this plot portrays.

In retrospect, it is not particularly surprising that neural networks with fewer neurons than a leech were unable to solve sophisticated artificial intelligence problems. Even today's networks, which we consider quite large from a computational systems point of view, are smaller than the nervous system of even relatively primitive vertebrate animals like frogs.

The increase in model size over time, due to the availability of faster CPUs, the advent of general purpose GPUs (described in section 12.1.2), faster network connectivity and better software infrastructure for distributed computing, is one of the most important trends in the history of deep learning. This trend is generally expected to continue well into the future.

Figure 1.10: Number of connections per neuron over time. Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, and it is quite common for other neural networks to have as many connections per neuron as smaller mammals like mice. Even the human brain does not have an exorbitant amount of connections per neuron. Biological neural network sizes from Wikipedia (2015).

1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett et al., 2009)
6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU convolutional network (Krizhevsky et al., 2012)
9. COTS HPC unsupervised convolutional network (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

1.2.4 Increasing Accuracy, Complexity and Real-World Impact

Since the 1980s, deep learning has consistently improved in its ability to provide accurate recognition and prediction. Moreover, deep learning has consistently been applied with success to broader and broader sets of applications.

The earliest deep models were used to recognize individual objects in tightly cropped, extremely small images (Rumelhart et al., 1986a). Since then there has been a gradual increase in the size of images neural networks could process. Modern object recognition networks process rich high-resolution photographs and do not have a requirement that the photo be cropped near the object to be recognized

Figure 1.11: Increasing neural network size over time. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton et al., 2006)
11. GPU-accelerated convolutional network (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina et al., 2009)
14. Unsupervised convolutional network (Jarrett et al., 2009)
15. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU convolutional network (Krizhevsky et al., 2012)
19. COTS HPC unsupervised convolutional network (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)

(Krizhevsky et al., 2012). Similarly, the earliest networks could recognize only two kinds of objects (or in some cases, the absence or presence of a single kind of object), while these modern networks typically recognize at least 1,000 different categories of objects. The largest contest in object recognition is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) held each year. A dramatic moment in the meteoric rise of deep learning came when a convolutional network images used to construct a 3-D map of the human brain (Knowles-Barley et al., 2014). We expect deep learning to appear in more and more scientific fields in the future.

In summary, deep learning is an approach to machine learning that has drawn heavily on our knowledge of the human brain, statistics and applied math as it developed over the past several decades. In recent years, deep learning has seen tremendous growth in its popularity and usefulness, largely as the result of more powerful computers, larger datasets and techniques to train deeper networks. The years ahead are full of challenges and opportunities to improve deep learning even further and to bring it to new frontiers.

I

Applied Math and Machine Learning Basics

This part of the book introduces the basic mathematical concepts needed to understand deep learning. We begin with general ideas from applied math that enable us to define functions of many variables, find the highest and lowest points on these functions, and quantify degrees of belief.

Next, we describe the fundamental goals of machine learning. We describe how to accomplish these goals by specifying a model that represents certain beliefs, designing a cost function that measures how well those beliefs correspond with reality, and using a training algorithm to minimize that cost function.

This elementary framework is the basis for a broad variety of machine learning algorithms, including approaches to machine learning that are not deep. In the subsequent parts of the book, we develop deep learning algorithms within this framework.

2

Linear Algebra

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. Yet because linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms. We therefore precede our introduction to deep learning with a focused presentation of the key linear algebra prerequisites.

If you are already familiar with linear algebra, feel free to skip this chapter. If you have previous experience with these concepts but need a detailed reference sheet to review key formulas, we recommend *The Matrix Cookbook* (Petersen and Pedersen, 2006). If you have had no exposure at all to linear algebra, this chapter will teach you enough to read this book, but we highly recommend that you also consult another resource focused exclusively on teaching linear algebra, such as Shilov (1977). This chapter completely omits many important linear algebra topics that are not essential for understanding deep learning.

2.1 Scalars, Vectors, Matrices and Tensors

The study of linear algebra involves several types of mathematical objects:

- **Scalars**: A scalar is just a single number, in contrast to most of the other objects studied in linear algebra, which are usually arrays of multiple numbers. We write scalars in italics. We usually give scalars lowercase variable names. When we introduce them, we specify what kind of number they are. For example, we might say “Let $s \in \mathbb{R}$ be the slope of the line,” while defining a real-valued scalar, or “Let $n \in \mathbb{N}$ be the number of units,” while defining a natural number scalar.

- **Vectors**: A vector is an array of numbers. The numbers are arranged in order. We can identify each individual number by its index in that ordering. Typically we give vectors lowercase names in bold typeface, such as $\boldsymbol{x}$. The elements of the vector are identified by writing its name in italic typeface, with a subscript. The first element of $\boldsymbol{x}$ is x_1, the second element is x_2, and so on. We also need to say what kind of numbers are stored in the vector. If each element is in $\mathbb{R}$, and the vector has n elements, then the vector lies in the set formed by taking the Cartesian product of $\mathbb{R}$ n times, denoted as $\mathbb{R}^n$. When we need to explicitly identify the elements of a vector, we write them as a column enclosed in square brackets:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \tag{2.1}$$

  We can think of vectors as identifying points in space, with each element giving the coordinate along a different axis.

  Sometimes we need to index a set of elements of a vector. In this case, we define a set containing the indices and write the set as a subscript. For example, to access x_1, x_3 and x_6, we define the set $S = \{1, 3, 6\}$ and write $\boldsymbol{x}_S$. We use the $-$ sign to index the complement of a set. For example $\boldsymbol{x}_{-1}$ is the vector containing all elements of $\boldsymbol{x}$ except for x_1, and $\boldsymbol{x}_{-S}$ is the vector containing all elements of $\boldsymbol{x}$ except for x_1, x_3 and x_6.

- **Matrices**: A matrix is a 2-D array of numbers, so each element is identified by two indices instead of just one. We usually give matrices uppercase variable names with bold typeface, such as $\boldsymbol{A}$. If a real-valued matrix $\boldsymbol{A}$ has a height of m and a width of n, then we say that $\boldsymbol{A} \in \mathbb{R}^{m \times n}$. We usually identify the elements of a matrix using its name in italic but not bold font, and the indices are listed with separating commas. For example, $A_{1,1}$ is the upper left entry of $\boldsymbol{A}$ and $A_{m,n}$ is the bottom right entry. We can identify all the numbers with vertical coordinate i by writing a “:” for the horizontal coordinate. For example, $\boldsymbol{A}_{i,:}$ denotes the horizontal cross section of $\boldsymbol{A}$ with vertical coordinate i. This is known as the i-th **row** of $\boldsymbol{A}$. Likewise, $\boldsymbol{A}_{:,i}$ is

Matrix multiplication is *not* commutative (the condition $\boldsymbol{AB} = \boldsymbol{BA}$ does not always hold), unlike scalar multiplication. However, the dot product between two vectors is commutative:

$$\boldsymbol{x}^\top \boldsymbol{y} = \boldsymbol{y}^\top \boldsymbol{x}. \tag{2.8}$$

The transpose of a matrix product has a simple form:

$$(\boldsymbol{AB})^\top = \boldsymbol{B}^\top \boldsymbol{A}^\top. \tag{2.9}$$

This enables us to demonstrate equation 2.8 by exploiting the fact that the value of such a product is a scalar and therefore equal to its own transpose:

$$\boldsymbol{x}^\top \boldsymbol{y} = \left(\boldsymbol{x}^\top \boldsymbol{y}\right)^\top = \boldsymbol{y}^\top \boldsymbol{x}. \tag{2.10}$$

Since the focus of this textbook is not linear algebra, we do not attempt to develop a comprehensive list of useful properties of the matrix product here, but the reader should be aware that many more exist.

We now know enough linear algebra notation to write down a system of linear equations:

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.11}$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is a known matrix, $\boldsymbol{b} \in \mathbb{R}^m$ is a known vector, and $\boldsymbol{x} \in \mathbb{R}^n$ is a vector of unknown variables we would like to solve for. Each element x_i of $\boldsymbol{x}$ is one of these unknown variables. Each row of $\boldsymbol{A}$ and each element of $\boldsymbol{b}$ provide another constraint. We can rewrite equation 2.11 as

$$\boldsymbol{A}_{1,:}\boldsymbol{x} = b_1 \tag{2.12}$$

$$\boldsymbol{A}_{2,:}\boldsymbol{x} = b_2 \tag{2.13}$$

$$\dots \tag{2.14}$$

$$\boldsymbol{A}_{m,:}\boldsymbol{x} = b_m \tag{2.15}$$

or even more explicitly as

$$\boldsymbol{A}_{1,1}x_1 + \boldsymbol{A}_{1,2}x_2 + \cdots + \boldsymbol{A}_{1,n}x_n = b_1 \tag{2.16}$$

$$\boldsymbol{A}_{2,1}x_1 + \boldsymbol{A}_{2,2}x_2 + \cdots + \boldsymbol{A}_{2,n}x_n = b_2 \tag{2.17}$$

$$\dots \tag{2.18}$$

$$\boldsymbol{A}_{m,1}x_1 + \boldsymbol{A}_{m,2}x_2 + \cdots + \boldsymbol{A}_{m,n}x_n = b_m. \tag{2.19}$$

Matrix-vector product notation provides a more compact representation for equations of this form.

2.3 Identity and Inverse Matrices

Linear algebra offers a powerful tool called **matrix inversion** that enables us to analytically solve equation 2.11 for many values of $\boldsymbol{A}$.

To describe matrix inversion, we first need to define the concept of an **identity matrix**. An identity matrix is a matrix that does not change any vector when we multiply that vector by that matrix. We denote the identity matrix that preserves n-dimensional vectors as $\boldsymbol{I}_n$. Formally, $\boldsymbol{I}_n \in \mathbb{R}^{n \times n}$, and

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n \boldsymbol{x} = \boldsymbol{x}. \tag{2.20}$$

The structure of the identity matrix is simple: all the entries along the main diagonal are 1, while all the other entries are zero. See figure 2.2 for an example.

The **matrix inverse** of $\boldsymbol{A}$ is denoted as $\boldsymbol{A}^{-1}$, and it is defined as the matrix such that

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n. \tag{2.21}$$

We can now solve equation 2.11 using the following steps:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.22}$$

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{2.23}$$

$$\boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{2.24}$$

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}. \tag{2.25}$$

Of course, this process depends on it being possible to find $\boldsymbol{A}^{-1}$. We discuss the conditions for the existence of $\boldsymbol{A}^{-1}$ in the following section.

When $\boldsymbol{A}^{-1}$ exists, several different algorithms can find it in closed form. In theory, the same inverse matrix can then be used to solve the equation many times for different values of $\boldsymbol{b}$. $\boldsymbol{A}^{-1}$ is primarily useful as a theoretical tool, however, and should not actually be used in practice for most software applications. Because $\boldsymbol{A}^{-1}$ can be represented with only limited precision on a digital computer, algorithms that make use of the value of $\boldsymbol{b}$ can usually obtain more accurate estimates of $\boldsymbol{x}$.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2.2: Example identity matrix: This is $\boldsymbol{I}_3$.

2.4 Linear Dependence and Span

For $\boldsymbol{A}^{-1}$ to exist, equation 2.11 must have exactly one solution for every value of $\boldsymbol{b}$. It is also possible for the system of equations to have no solutions or infinitely many solutions for some values of $\boldsymbol{b}$. It is not possible, however, to have more than one but less than infinitely many solutions for a particular $\boldsymbol{b}$; if both $\boldsymbol{x}$ and $\boldsymbol{y}$ are solutions, then

$$\boldsymbol{z} = \alpha \boldsymbol{x} + (1 - \alpha) \boldsymbol{y} \tag{2.26}$$

is also a solution for any real α.

To analyze how many solutions the equation has, think of the columns of $\boldsymbol{A}$ as specifying different directions we can travel in from the **origin** (the point specified by the vector of all zeros), then determine how many ways there are of reaching $\boldsymbol{b}$. In this view, each element of $\boldsymbol{x}$ specifies how far we should travel in each of these directions, with x_i specifying how far to move in the direction of column i:

$$\boldsymbol{A}\boldsymbol{x} = \sum_i x_i \boldsymbol{A}_{:,i}. \tag{2.27}$$

In general, this kind of operation is called a **linear combination**. Formally, a linear combination of some set of vectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ is given by multiplying each vector $\boldsymbol{v}^{(i)}$ by a corresponding scalar coefficient and adding the results:

$$\sum_i c_i \boldsymbol{v}^{(i)}. \tag{2.28}$$

The **span** of a set of vectors is the set of all points obtainable by linear combination of the original vectors.

Determining whether $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ has a solution thus amounts to testing whether $\boldsymbol{b}$ is in the span of the columns of $\boldsymbol{A}$. This particular span is known as the **column space**, or the **range**, of $\boldsymbol{A}$.

In order for the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ to have a solution for all values of $\boldsymbol{b} \in \mathbb{R}^m$, we therefore require that the column space of $\boldsymbol{A}$ be all of $\mathbb{R}^m$. If any point in $\mathbb{R}^m$ is excluded from the column space, that point is a potential value of $\boldsymbol{b}$ that has no solution. The requirement that the column space of $\boldsymbol{A}$ be all of $\mathbb{R}^m$ implies immediately that $\boldsymbol{A}$ must have at least m columns, that is, $n \geq m$. Otherwise, the dimensionality of the column space would be less than m. For example, consider a 3×2 matrix. The target $\boldsymbol{b}$ is 3-D, but $\boldsymbol{x}$ is only 2-D, so modifying the value of $\boldsymbol{x}$ at best enables us to trace out a 2-D plane within $\mathbb{R}^3$. The equation has a solution if and only if $\boldsymbol{b}$ lies on that plane.

Having $n \geq m$ is only a necessary condition for every point to have a solution. It is not a sufficient condition, because it is possible for some of the columns to be redundant. Consider a 2×2 matrix where both of the columns are identical. This has the same column space as a 2×1 matrix containing only one copy of the replicated column. In other words, the column space is still just a line and fails to encompass all of $\mathbb{R}^2$, even though there are two columns.

Formally, this kind of redundancy is known as **linear dependence**. A set of vectors is **linearly independent** if no vector in the set is a linear combination of the other vectors. If we add a vector to a set that is a linear combination of the other vectors in the set, the new vector does not add any points to the set's span. This means that for the column space of the matrix to encompass all of $\mathbb{R}^m$, the matrix must contain at least one set of m linearly independent columns. This condition is both necessary and sufficient for equation 2.11 to have a solution for every value of $\boldsymbol{b}$. Note that the requirement is for a set to have exactly m linear independent columns, not at least m. No set of m-dimensional vectors can have more than m mutually linearly independent columns, but a matrix with more than m columns may have more than one such set.

For the matrix to have an inverse, we additionally need to ensure that equation 2.11 has *at most* one solution for each value of $\boldsymbol{b}$. To do so, we need to make certain that the matrix has at most m columns. Otherwise there is more than one way of parametrizing each solution.

Together, this means that the matrix must be **square**, that is, we require that $m = n$ and that all the columns be linearly independent. A square matrix with linearly dependent columns is known as **singular**.

If $\boldsymbol{A}$ is not square or is square but singular, solving the equation is still possible, but we cannot use the method of matrix inversion to find the solution.

So far we have discussed matrix inverses as being multiplied on the left. It is also possible to define an inverse that is multiplied on the right:

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}. \tag{2.29}$$

For square matrices, the left inverse and right inverse are equal.

2.5 Norms

Sometimes we need to measure the size of a vector. In machine learning, we usually measure the size of vectors using a function called a **norm**. Formally, the L^p norm is given by

$$||\boldsymbol{x}||_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}} \tag{2.30}$$

for $p \in \mathbb{R}, p \geq 1$.

Norms, including the L^p norm, are functions mapping vectors to non-negative values. On an intuitive level, the norm of a vector $\boldsymbol{x}$ measures the distance from the origin to the point $\boldsymbol{x}$. More rigorously, a norm is any function f that satisfies the following properties:

- $f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$
- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$ (the **triangle inequality**)
- $\forall \alpha \in \mathbb{R}, f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

The L^2 norm, with $p = 2$, is known as the **Euclidean norm**, which is simply the Euclidean distance from the origin to the point identified by $\boldsymbol{x}$. The L^2 norm is used so frequently in machine learning that it is often denoted simply as $||\boldsymbol{x}||$, with the subscript 2 omitted. It is also common to measure the size of a vector using the squared L^2 norm, which can be calculated simply as $\boldsymbol{x}^\top \boldsymbol{x}$.

The squared L^2 norm is more convenient to work with mathematically and computationally than the L^2 norm itself. For example, each derivative of the squared L^2 norm with respect to each element of $\boldsymbol{x}$ depends only on the corresponding element of $\boldsymbol{x}$, while all the derivatives of the L^2 norm depend on the entire vector. In many contexts, the squared L^2 norm may be undesirable because it increases very slowly near the origin. In several machine learning applications, it is important to discriminate between elements that are exactly zero and elements that are small but nonzero. In these cases, we turn to a function that grows at the same rate in all locations, but that retains mathematical simplicity: the L^1 norm. The L^1 norm may be simplified to

$$||\boldsymbol{x}||_1 = \sum_i |x_i|. \tag{2.31}$$

The L^1 norm is commonly used in machine learning when the difference between zero and nonzero elements is very important. Every time an element of $\boldsymbol{x}$ moves away from 0 by ϵ, the L^1 norm increases by ϵ.

We sometimes measure the size of the vector by counting its number of nonzero elements. Some authors refer to this function as the "L^0 norm," but this is incorrect terminology. The number of nonzero entries in a vector is not a norm, because

For example, integers can be decomposed into prime factors. The way we represent the number 12 will change depending on whether we write it in base ten or in binary, but it will always be true that $12 = 2 \times 2 \times 3$. From this representation we can conclude useful properties, for example, that 12 is not divisible by 5, and that any integer multiple of 12 will be divisible by 3.

Much as we can discover something about the true nature of an integer by decomposing it into prime factors, we can also decompose matrices in ways that show us information about their functional properties that is not obvious from the representation of the matrix as an array of elements.

One of the most widely used kinds of matrix decomposition is called **eigendecomposition**, in which we decompose a matrix into a set of eigenvectors and eigenvalues.

An **eigenvector** of a square matrix $\boldsymbol{A}$ is a nonzero vector $\boldsymbol{v}$ such that multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}. \tag{2.39}$$

The scalar λ is known as the **eigenvalue** corresponding to this eigenvector. (One can also find a **left eigenvector** such that $\boldsymbol{v}^\top \boldsymbol{A} = \lambda \boldsymbol{v}^\top$, but we are usually concerned with right eigenvectors.)

If $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$, then so is any rescaled vector $s\boldsymbol{v}$ for $s \in \mathbb{R}, s \neq 0$. Moreover, $s\boldsymbol{v}$ still has the same eigenvalue. For this reason, we usually look only for unit eigenvectors.

Suppose that a matrix $\boldsymbol{A}$ has n linearly independent eigenvectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ with corresponding eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$. We may concatenate all the eigenvectors to form a matrix $\boldsymbol{V}$ with one eigenvector per column: $\boldsymbol{V} = [\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}]$. Likewise, we can concatenate the eigenvalues to form a vector $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_n]^\top$. The **eigendecomposition** of $\boldsymbol{A}$ is then given by

$$\boldsymbol{A} = \boldsymbol{V} \operatorname{diag}(\boldsymbol{\lambda}) \boldsymbol{V}^{-1}. \tag{2.40}$$

We have seen that *constructing* matrices with specific eigenvalues and eigenvectors enables us to stretch space in desired directions. Yet we often want to **decompose** matrices into their eigenvalues and eigenvectors. Doing so can help us analyze certain properties of the matrix, much as decomposing an integer into its prime factors can help us understand the behavior of that integer.

Not every matrix can be decomposed into eigenvalues and eigenvectors. In some cases, the decomposition exists but involves complex rather than real numbers.

Fortunately, in this book, we usually need to decompose only a specific class of matrices that have a simple decomposition. Specifically, every real symmetric matrix can be decomposed into an expression using only real-valued eigenvectors and eigenvalues:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top, \tag{2.41}$$

where $\boldsymbol{Q}$ is an orthogonal matrix composed of eigenvectors of $\boldsymbol{A}$, and $\boldsymbol{\Lambda}$ is a diagonal matrix. The eigenvalue $\Lambda_{i,i}$ is associated with the eigenvector in column i of $\boldsymbol{Q}$, denoted as $\boldsymbol{Q}_{:,i}$. Because $\boldsymbol{Q}$ is an orthogonal matrix, we can think of $\boldsymbol{A}$ as scaling space by λ_i in direction $\boldsymbol{v}^{(i)}$. See figure 2.3 for an example.

While any real symmetric matrix $\boldsymbol{A}$ is guaranteed to have an eigendecomposition, the eigendecomposition may not be unique. If any two or more eigenvectors share the same eigenvalue, then any set of orthogonal vectors lying in their span are also eigenvectors with that eigenvalue, and we could equivalently choose a $\boldsymbol{Q}$ using those eigenvectors instead. By convention, we usually sort the entries of $\boldsymbol{\Lambda}$ in descending order. Under this convention, the eigendecomposition is unique only if all the eigenvalues are unique.

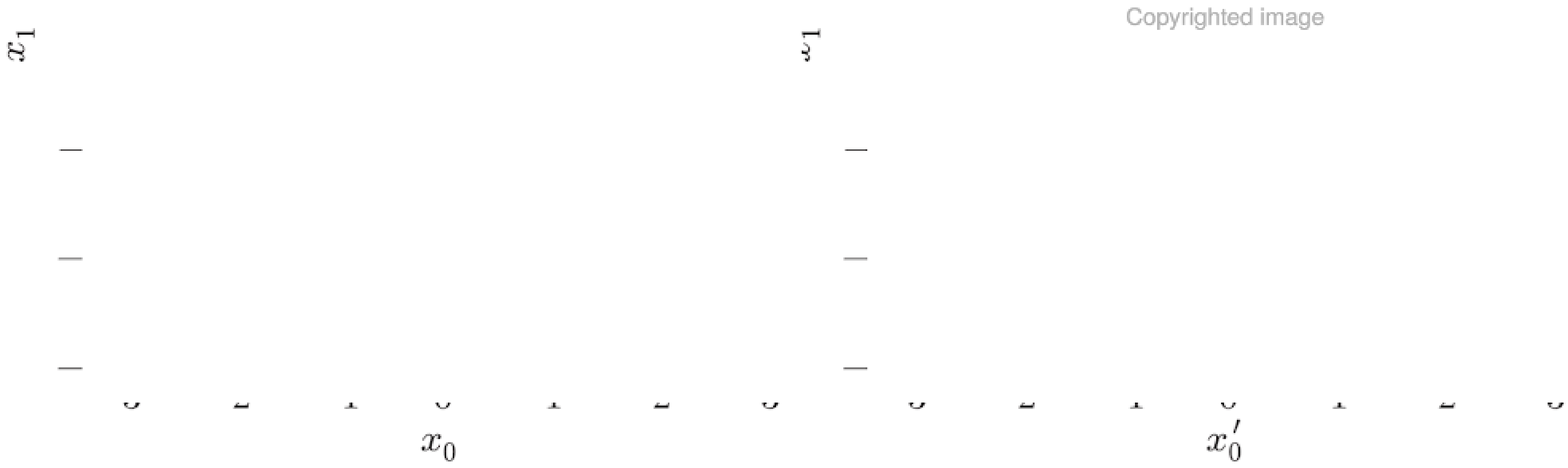


Figure 2.3: Effect of eigenvectors and eigenvalues. An example of the effect of eigenvectors and eigenvalues. Here, we have a matrix $\boldsymbol{A}$ with two orthonormal eigenvectors, $\boldsymbol{v}^{(1)}$ with eigenvalue λ_1 and $\boldsymbol{v}^{(2)}$ with eigenvalue λ_2. *(Left)*We plot the set of all unit vectors $\boldsymbol{u} \in \mathbb{R}^2$ as a unit circle. *(Right)*We plot the set of all points $\boldsymbol{A}\boldsymbol{u}$. By observing the way that $\boldsymbol{A}$ distorts the unit circle, we can see that it scales space in direction $\boldsymbol{v}^{(i)}$ by λ_i.

The eigendecomposition of a matrix tells us many useful facts about the matrix. The matrix is singular if and only if any of the eigenvalues are zero. The eigendecomposition of a real symmetric matrix can also be used to optimize quadratic expressions of the form $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}$ subject to $||\boldsymbol{x}||_2 = 1$. Whenever $\boldsymbol{x}$ is equal to an eigenvector of $\boldsymbol{A}$, f takes on the value of the corresponding eigenvalue. The maximum value of f within the constraint region is the maximum eigenvalue and its minimum value within the constraint region is the minimum eigenvalue.

A matrix whose eigenvalues are all positive is called **positive definite**. A matrix whose eigenvalues are all positive or zero valued is called **positive semidefinite**. Likewise, if all eigenvalues are negative, the matrix is **negative definite**, and if all eigenvalues are negative or zero valued, it is **negative semidefinite**. Positive semidefinite matrices are interesting because they guarantee that $\forall \boldsymbol{x}$, $\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \geq 0$. Positive definite matrices additionally guarantee that $\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$.

2.8 Singular Value Decomposition

In section 2.7, we saw how to decompose a matrix into eigenvectors and eigenvalues. The **singular value decomposition** (SVD) provides another way to factorize a matrix, into **singular vectors** and **singular values**. The SVD enables us to discover some of the same kind of information as the eigendecomposition reveals; however, the SVD is more generally applicable. Every real matrix has a singular value decomposition, but the same is not true of the eigenvalue decomposition. For example, if a matrix is not square, the eigendecomposition is not defined, and we must use a singular value decomposition instead.

Recall that the eigendecomposition involves analyzing a matrix $\boldsymbol{A}$ to discover a matrix $\boldsymbol{V}$ of eigenvectors and a vector of eigenvalues $\boldsymbol{\lambda}$ such that we can rewrite $\boldsymbol{A}$ as

$$\boldsymbol{A} = \boldsymbol{V}\operatorname{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}. \tag{2.42}$$

The singular value decomposition is similar, except this time we will write $\boldsymbol{A}$ as a product of three matrices:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^\top. \tag{2.43}$$

Suppose that $\boldsymbol{A}$ is an $m \times n$ matrix. Then $\boldsymbol{U}$ is defined to be an $m \times m$ matrix, $\boldsymbol{D}$ to be an $m \times n$ matrix, and $\boldsymbol{V}$ to be an $n \times n$ matrix.

Each of these matrices is defined to have a special structure. The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are both defined to be orthogonal matrices. The matrix $\boldsymbol{D}$ is defined to be a diagonal matrix. Note that $\boldsymbol{D}$ is not necessarily square.

The elements along the diagonal of $\boldsymbol{D}$ are known as the **singular values** of the matrix $\boldsymbol{A}$. The columns of $\boldsymbol{U}$ are known as the **left-singular vectors**. The columns of $\boldsymbol{V}$ are known as as the **right-singular vectors**.

We can actually interpret the singular value decomposition of $\boldsymbol{A}$ in terms of the eigendecomposition of functions of $\boldsymbol{A}$. The left-singular vectors of $\boldsymbol{A}$ are the eigenvectors of $\boldsymbol{A}\boldsymbol{A}^\top$. The right-singular vectors of $\boldsymbol{A}$ are the eigenvectors of $\boldsymbol{A}^\top\boldsymbol{A}$. The nonzero singular values of $\boldsymbol{A}$ are the square roots of the eigenvalues of $\boldsymbol{A}^\top\boldsymbol{A}$. The same is true for $\boldsymbol{A}\boldsymbol{A}^\top$.

Perhaps the most useful feature of the SVD is that we can use it to partially generalize matrix inversion to nonsquare matrices, as we will see in the next section.

2.9 The Moore-Penrose Pseudoinverse

Matrix inversion is not defined for matrices that are not square. Suppose we want to make a left-inverse $\boldsymbol{B}$ of a matrix $\boldsymbol{A}$ so that we can solve a linear equation

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y} \tag{2.44}$$

by left-multiplying each side to obtain

$$\boldsymbol{x} = \boldsymbol{B}\boldsymbol{y}. \tag{2.45}$$

Depending on the structure of the problem, it may not be possible to design a unique mapping from $\boldsymbol{A}$ to $\boldsymbol{B}$.

If $\boldsymbol{A}$ is taller than it is wide, then it is possible for this equation to have no solution. If $\boldsymbol{A}$ is wider than it is tall, then there could be multiple possible solutions.

The **Moore-Penrose pseudoinverse** enables us to make some headway in these cases. The pseudoinverse of $\boldsymbol{A}$ is defined as a matrix

$$\boldsymbol{A}^+ = \lim_{\alpha \searrow 0} (\boldsymbol{A}^\top \boldsymbol{A} + \alpha \boldsymbol{I})^{-1} \boldsymbol{A}^\top. \tag{2.46}$$

Practical algorithms for computing the pseudoinverse are based not on this definition, but rather on the formula

$$\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{D}^+\boldsymbol{U}^\top, \tag{2.47}$$

where $\boldsymbol{U}$, $\boldsymbol{D}$ and $\boldsymbol{V}$ are the singular value decomposition of $\boldsymbol{A}$, and the pseudoinverse $\boldsymbol{D}^{+}$ of a diagonal matrix $\boldsymbol{D}$ is obtained by taking the reciprocal of its nonzero elements then taking the transpose of the resulting matrix.

When $\boldsymbol{A}$ has more columns than rows, then solving a linear equation using the pseudoinverse provides one of the many possible solutions. Specifically, it provides the solution $\boldsymbol{x} = \boldsymbol{A}^{+}\boldsymbol{y}$ with minimal Euclidean norm $||\boldsymbol{x}||_2$ among all possible solutions.

When $\boldsymbol{A}$ has more rows than columns, it is possible for there to be no solution. In this case, using the pseudoinverse gives us the $\boldsymbol{x}$ for which $\boldsymbol{A}\boldsymbol{x}$ is as close as possible to $\boldsymbol{y}$ in terms of Euclidean norm $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}||_2$.

2.10 The Trace Operator

The trace operator gives the sum of all the diagonal entries of a matrix:

$$\operatorname{Tr}(\boldsymbol{A}) = \sum_i \boldsymbol{A}_{i,i}. \tag{2.48}$$

The trace operator is useful for a variety of reasons. Some operations that are difficult to specify without resorting to summation notation can be specified using matrix products and the trace operator. For example, the trace operator provides an alternative way of writing the Frobenius norm of a matrix:

$$||A||_F = \sqrt{\operatorname{Tr}(\boldsymbol{A}\boldsymbol{A}^{\top})}. \tag{2.49}$$

Writing an expression in terms of the trace operator opens up opportunities to manipulate the expression using many useful identities. For example, the trace operator is invariant to the transpose operator:

$$\operatorname{Tr}(\boldsymbol{A}) = \operatorname{Tr}(\boldsymbol{A}^{\top}). \tag{2.50}$$

The trace of a square matrix composed of many factors is also invariant to moving the last factor into the first position, if the shapes of the corresponding matrices allow the resulting product to be defined:

$$\operatorname{Tr}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{C}) = \operatorname{Tr}(\boldsymbol{C}\boldsymbol{A}\boldsymbol{B}) = \operatorname{Tr}(\boldsymbol{B}\boldsymbol{C}\boldsymbol{A}) \tag{2.51}$$

or more generally,

$$\operatorname{Tr}(\prod_{i=1}^{n} \boldsymbol{F}^{(i)}) = \operatorname{Tr}(\boldsymbol{F}^{(n)} \prod_{i=1}^{n-1} \boldsymbol{F}^{(i)}). \tag{2.52}$$

To make further progress, we must substitute in the definition of $g(\boldsymbol{c})$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{D}^\top \boldsymbol{D}\boldsymbol{c} \tag{2.60}$$

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{I}_l \boldsymbol{c} \tag{2.61}$$

(by the orthogonality and unit norm constraints on $\boldsymbol{D}$)

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}. \tag{2.62}$$

We can solve this optimization problem using vector calculus (see section 4.3 if you do not know how to do this):

$$\nabla_{\boldsymbol{c}}(-2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}) = \boldsymbol{0} \tag{2.63}$$

$$-2\boldsymbol{D}^\top \boldsymbol{x} + 2\boldsymbol{c} = \boldsymbol{0} \tag{2.64}$$

$$\boldsymbol{c} = \boldsymbol{D}^\top \boldsymbol{x}. \tag{2.65}$$

This makes the algorithm efficient: we can optimally encode $\boldsymbol{x}$ using just a matrix-vector operation. To encode a vector, we apply the encoder function

$$f(\boldsymbol{x}) = \boldsymbol{D}^\top \boldsymbol{x}. \tag{2.66}$$

Using a further matrix multiplication, we can also define the PCA reconstruction operation:

$$r(\boldsymbol{x}) = g\left(f\left(\boldsymbol{x}\right)\right) = \boldsymbol{D}\boldsymbol{D}^\top \boldsymbol{x}. \tag{2.67}$$

Next, we need to choose the encoding matrix $\boldsymbol{D}$. To do so, we revisit the idea of minimizing the L^2 distance between inputs and reconstructions. Since we will use the same matrix $\boldsymbol{D}$ to decode all the points, we can no longer consider the points in isolation. Instead, we must minimize the Frobenius norm of the matrix of errors computed over all dimensions and all points:

$$\boldsymbol{D}^* = \arg\underset{\boldsymbol{D}}{\mathrm{mi}} \qquad\qquad \text{ubject to } \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l. \tag{2.68}$$

To derive the algorithm for finding $\boldsymbol{D}^*$, we start by considering the case where $l = 1$. In this case, $\boldsymbol{D}$ is just a single vector, $\boldsymbol{d}$. Substituting equation 2.67 into equation 2.68 and simplifying $\boldsymbol{D}$ into $\boldsymbol{d}$, the problem reduces to

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^{(i)}||_2^2 \text{ subject to } ||\boldsymbol{d}||_2 = 1. \tag{2.69}$$

The above formulation is the most direct way of performing the substitution but is not the most stylistically pleasing way to write the equation. It places the scalar value $\boldsymbol{d}^\top \boldsymbol{x}^{(i)}$ on the right of the vector $\boldsymbol{d}$. Scalar coefficients are conventionally written on the left of vector they operate on. We therefore usually write such a formula as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{d}^\top \boldsymbol{x}^{(i)} \boldsymbol{d}||_2^2 \text{ subject to } ||\boldsymbol{d}||_2 = 1, \tag{2.70}$$

or, exploiting the fact that a scalar is its own transpose, as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(i)\top} \boldsymbol{d}\boldsymbol{d}||_2^2 \text{ subject to } ||\boldsymbol{d}||_2 = 1. \tag{2.71}$$

The reader should aim to become familiar with such cosmetic rearrangements.

At this point, it can be helpful to rewrite the problem in terms of a single design matrix of examples, rather than as a sum over separate example vectors. This will enable us to use more compact notation. Let $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ be the matrix defined by stacking all the vectors describing the points, such that $\boldsymbol{X}_{i,:} = \boldsymbol{x}^{(i)^\top}$. We can now rewrite the problem as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} ||\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top||_F^2 \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1. \tag{2.72}$$

Disregarding the constraint for the moment, we can simplify the Frobenius norm portion as follows:

$$\arg\min_{\boldsymbol{d}} ||\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top||_F^2 \tag{2.73}$$

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}\left(\left(\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\right)^\top \left(\boldsymbol{X} - \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top\right)\right) \tag{2.74}$$

(by equation 2.49)

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X} - \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.75}$$

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}) - \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) - \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.76}$$

$$= \arg\min_{\boldsymbol{d}} - \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) - \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.77}$$

(because terms not involving $\boldsymbol{d}$ do not affect the $\arg\min$)

$$= \arg\min_{\boldsymbol{d}} -2 \operatorname{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \tag{2.78}$$

(because we can cycle the order of the matrices inside a trace, equation 2.52)

$$= \arg\min_{\boldsymbol{d}} -2\,\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{d}\boldsymbol{d}^\top) \tag{2.79}$$

(using the same property again).

At this point, we reintroduce the constraint:

$$\arg\min_{\boldsymbol{d}} -2\,\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1 \tag{2.80}$$

$$= \arg\min_{\boldsymbol{d}} -2\,\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) + \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1 \tag{2.81}$$

(due to the constraint)

$$= \arg\min_{\boldsymbol{d}} -\mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1 \tag{2.82}$$

$$= \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1 \tag{2.83}$$

$$= \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1. \tag{2.84}$$

This optimization problem may be solved using eigendecomposition. Specifically, the optimal $\boldsymbol{d}$ is given by the eigenvector of $\boldsymbol{X}^\top \boldsymbol{X}$ corresponding to the largest eigenvalue.

This derivation is specific to the case of $l = 1$ and recovers only the first principal component. More generally, when we wish to recover a basis of principal components, the matrix $\boldsymbol{D}$ is given by the l eigenvectors corresponding to the largest eigenvalues. This may be shown using proof by induction. We recommend writing this proof as an exercise.

Linear algebra is one of the fundamental mathematical disciplines necessary to understanding deep learning. Another key area of mathematics that is ubiquitous in machine learning is probability theory, presented next.

3

Probability and Information Theory

In this chapter, we describe probability theory and information theory.

Probability theory is a mathematical framework for representing uncertain statements. It provides a means of quantifying uncertainty as well as axioms for deriving new uncertain statements. In artificial intelligence applications, we use probability theory in two major ways. First, the laws of probability tell us how AI systems should reason, so we design our algorithms to compute or approximate various expressions derived using probability theory. Second, we can use probability and statistics to theoretically analyze the behavior of proposed AI systems.

Probability theory is a fundamental tool of many disciplines of science and engineering. We provide this chapter to ensure that readers whose background is primarily in software engineering, with limited exposure to probability theory, can understand the material in this book.

While probability theory allows us to make uncertain statements and to reason in the presence of uncertainty, information theory enables us to quantify the amount of uncertainty in a probability distribution.

If you are already familiar with probability theory and information theory, you may wish to skip this chapter except for section 3.14, which describes the graphs we use to describe structured probabilistic models for machine learning. If you have absolutely no prior experience with these subjects, this chapter should be sufficient to successfully carry out deep learning research projects, but we do suggest that you consult an additional resource, such as Jaynes (2003).

has certain symptoms. For more details about why a small set of common sense assumptions implies that the same axioms must control both kinds of probability, see Ramsey (1926).

Probability can be seen as the extension of logic to deal with uncertainty. Logic provides a set of formal rules for determining what propositions are implied to be true or false given the assumption that some other set of propositions is true or false. Probability theory provides a set of formal rules for determining the likelihood of a proposition being true given the likelihood of other propositions.

3.2 Random Variables

A **random variable** is a variable that can take on different values randomly. We typically denote the random variable itself with a lowercase letter in plain typeface, and the values it can take on with lowercase script letters. For example, x_1 and x_2 are both possible values that the random variable x can take on. For vector-valued variables, we would write the random variable as $\mathbf{x}$ and one of its values as $\boldsymbol{x}$. On its own, a random variable is just a description of the states that are possible; it must be coupled with a probability distribution that specifies how likely each of these states are.

Random variables may be discrete or continuous. A discrete random variable is one that has a finite or countably infinite number of states. Note that these states are not necessarily the integers; they can also just be named states that are not considered to have any numerical value. A continuous random variable is associated with a real value.

3.3 Probability Distributions

A **probability distribution** is a description of how likely a random variable or set of random variables is to take on each of its possible states. The way we describe probability distributions depends on whether the variables are discrete or continuous.

3.3.1 Discrete Variables and Probability Mass Functions

A probability distribution over discrete variables may be described using a **probability mass function** (PMF). We typically denote probability mass functions with a capital P. Often we associate each random variable with a different probability mass function and the reader must infer which PMF to use based on the identity of the random variable, rather than on the name of the function; $P(\mathrm{x})$ is usually not the same as $P(\mathrm{y})$.

The probability mass function maps from a state of a random variable to the probability of that random variable taking on that state. The probability that $\mathrm{x} = x$ is denoted as $P(x)$, with a probability of 1 indicating that $\mathrm{x} = x$ is certain and a probability of 0 indicating that $\mathrm{x} = x$ is impossible. Sometimes to disambiguate which PMF to use, we write the name of the random variable explicitly: $P(\mathrm{x} = x)$. Sometimes we define a variable first, then use $\sim$ notation to specify which distribution it follows later: $\mathrm{x} \sim P(\mathrm{x})$.

Probability mass functions can act on many variables at the same time. Such a probability distribution over many variables is known as a **joint probability distribution**. $P(\mathrm{x} = x, \mathrm{y} = y)$ denotes the probability that $\mathrm{x} = x$ and $\mathrm{y} = y$ simultaneously. We may also write $P(x, y)$ for brevity.

To be a PMF on a random variable x, a function P must satisfy the following properties:

- The domain of P must be the set of all possible states of x.
- $\forall x \in \mathrm{x}, 0 \leq P(x) \leq 1$. An impossible event has probability 0, and no state can be less probable than that. Likewise, an event that is guaranteed to happen has probability 1, and no state can have a greater chance of occurring.
- $\sum_{x \in \mathrm{x}} P(x) = 1$. We refer to this property as being **normalized**. Without this property, we could obtain probabilities greater than one by computing the probability of one of many events occurring.

For example, consider a single discrete random variable x with k different states. We can place a **uniform distribution** on x—that is, make each of its states equally likely—by setting its PMF to

$$P(\mathrm{x} = x_i) = \frac{1}{k} \tag{3.1}$$

for all i. We can see that this fits the requirements for a probability mass function. The value $\frac{1}{k}$ is positive because k is a positive integer. We also see that

$$\sum_i P(\mathrm{x} = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1, \tag{3.2}$$

so the distribution is properly normalized.

3.3.2 Continuous Variables and Probability Density Functions

When working with continuous random variables, we describe probability distributions using a **probability density function** (PDF) rather than a probability mass function. To be a probability density function, a function p must satisfy the following properties:

- The domain of p must be the set of all possible states of x.
- $\forall x \in \mathrm{x}, p(x) \geq 0$. Note that we do not require $p(x) \leq 1$.
- $\int p(x)dx = 1$.

A probability density function $p(x)$ does not give the probability of a specific state directly; instead the probability of landing inside an infinitesimal region with volume δx is given by $p(x)\delta x$.

We can integrate the density function to find the actual probability mass of a set of points. Specifically, the probability that x lies in some set $\mathbb{S}$ is given by the integral of $p(x)$ over that set. In the univariate example, the probability that x lies in the interval $[a, b]$ is given by $\int_{[a,b]} p(x)dx$.

For an example of a PDF corresponding to a specific probability density over a continuous random variable, consider a uniform distribution on an interval of the real numbers. We can do this with a function $u(x; a, b)$, where a and b are the endpoints of the interval, with $b > a$. The ";" notation means "parametrized by"; we consider x to be the argument of the function, while a and b are parameters that define the function. To ensure that there is no probability mass outside the interval, we say $u(x; a, b) = 0$ for all $x \notin [a, b]$. Within $[a, b]$, $u(x; a, b) = \frac{1}{b-a}$. We can see that this is non-negative everywhere. Additionally, it integrates to 1. We often denote that x follows the uniform distribution on $[a, b]$ by writing $\mathrm{x} \sim U(a, b)$.

3.4 Marginal Probability

Sometimes we know the probability distribution over a set of variables and we want to know the probability distribution over just a subset of them. The probability distribution over the subset is known as the **marginal probability distribution.**

For example, suppose we have discrete random variables x and y, and we know $P(\mathrm{x}, \mathrm{y})$. We can find $P(\mathrm{x})$ with the **sum rule**:

$$\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_y P(\mathrm{x} = x, \mathrm{y} = y). \tag{3.3}$$

The name "marginal probability" comes from the process of computing marginal probabilities on paper. When the values of $P(\mathrm{x}, \mathrm{y})$ are written in a grid with different values of x in rows and different values of y in columns, it is natural to sum across a row of the grid, then write $P(x)$ in the margin of the paper just to the right of the row.

For continuous variables, we need to use integration instead of summation:

$$p(x) = \int p(x, y)dy. \tag{3.4}$$

3.5 Conditional Probability

In many cases, we are interested in the probability of some event, given that some other event has happened. This is called a **conditional probability**. We denote the conditional probability that $\mathrm{y} = y$ given $\mathrm{x} = x$ as $P(\mathrm{y} = y \mid \mathrm{x} = x)$. This conditional probability can be computed with the formula

$$P(\mathrm{y} = y \mid \mathrm{x} = x) = \frac{P(\mathrm{y} = y, \mathrm{x} = x)}{P(\mathrm{x} = x)}. \tag{3.5}$$

The conditional probability is only defined when $P(\mathrm{x} = x) > 0$. We cannot compute the conditional probability conditioned on an event that never happens.

It is important not to confuse conditional probability with computing what would happen if some action were undertaken. The conditional probability that a person is from Germany given that they speak German is quite high, but if a randomly selected person is taught to speak German, their country of origin does not change. Computing the consequences of an action is called making an **intervention query**. Intervention queries are the domain of **causal modeling**, which we do not explore in this book.

3.6 The Chain Rule of Conditional Probabilities

Any joint probability distribution over many random variables may be decomposed into conditional distributions over only one variable:

$$P(\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)})\Pi_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(i-1)}). \tag{3.6}$$

This observation is known as the **chain rule**, or **product rule**, of probability. It follows immediately from the definition of conditional probability in equation 3.5.

For example, applying the definition twice, we get

$$\begin{aligned} P(\mathrm{a},\mathrm{b},\mathrm{c}) &= P(\mathrm{a} \mid \mathrm{b},\mathrm{c})P(\mathrm{b},\mathrm{c}) \\ P(\mathrm{b},\mathrm{c}) &= P(\mathrm{b} \mid \mathrm{c})P(\mathrm{c}) \\ P(\mathrm{a},\mathrm{b},\mathrm{c}) &= P(\mathrm{a} \mid \mathrm{b},\mathrm{c})P(\mathrm{b} \mid \mathrm{c})P(\mathrm{c}). \end{aligned}$$

3.7 Independence and Conditional Independence

Two random variables x and y are **independent** if their probability distribution can be expressed as a product of two factors, one involving only x and one involving only y:

$$\forall x \in \mathrm{x}, y \in \mathrm{y},\ p(\mathrm{x}=x,\mathrm{y}=y) = p(\mathrm{x}=x)p(\mathrm{y}=y). \tag{3.7}$$

Two random variables x and y are **conditionally independent** given a random variable z if the conditional probability distribution over x and y factorizes in this way for every value of z:

$$\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z},\ p(\mathrm{x}=x,\mathrm{y}=y \mid \mathrm{z}=z) = p(\mathrm{x}=x \mid \mathrm{z}=z)p(\mathrm{y}=y \mid \mathrm{z}=z). \tag{3.8}$$

We can denote independence and conditional independence with compact notation: $\mathrm{x}\perp\mathrm{y}$ means that x and y are independent, while $\mathrm{x}\perp\mathrm{y} \mid \mathrm{z}$ means that x and y are conditionally independent given z.

3.8 Expectation, Variance and Covariance

The **expectation**, or **expected value**, of some function $f(x)$ with respect to a probability distribution $P(\mathrm{x})$ is the average, or mean value, that f takes on when x is drawn from P. For discrete variables this can be computed with a summation:

$$\mathbb{E}_{\mathrm{x}\sim P}[f(x)] = \sum_x P(x)f(x), \tag{3.9}$$

while for continuous variables, it is computed with an integral:

$$\mathbb{E}_{\mathrm{x}\sim p}[f(x)] = \int p(x)f(x)dx. \tag{3.10}$$

is parametrized by a vector $\boldsymbol{p} \in [0,1]^{k-1}$, where p_i gives the probability of the i-th state. The final, k-th state's probability is given by $1 - \mathbf{1}^\top \boldsymbol{p}$. Note that we must constrain $\mathbf{1}^\top \boldsymbol{p} \leq 1$. Multinoulli distributions are often used to refer to distributions over categories of objects, so we do not usually assume that state 1 has numerical value 1, and so on. For this reason, we do not usually need to compute the expectation or variance of multinoulli-distributed random variables.

The Bernoulli and multinoulli distributions are sufficient to describe any distribution over their domain. They are able to describe any distribution over their domain not so much because they are particularly powerful but rather because their domain is simple; they model discrete variables for which it is feasible to enumerate all the states. When dealing with continuous variables, there are uncountably many states, so any distribution described by a small number of parameters must impose strict limits on the distribution.

3.9.3 Gaussian Distribution

The most commonly used distribution over real numbers is the **normal distribution**, also known as the **Gaussian distribution**:

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.21}$$

See figure 3.1 for a plot of the normal distribution density function.

The two parameters $\mu \in \mathbb{R}$ and $\sigma \in (0, \infty)$ control the normal distribution. The parameter μ gives the coordinate of the central peak. This is also the mean of the distribution: $\mathbb{E}[\mathrm{x}] = \mu$. The standard deviation of the distribution is given by σ, and the variance by σ^2.

When we evaluate the PDF, we need to square and invert σ. When we need to frequently evaluate the PDF with different parameter values, a more efficient way of parametrizing the distribution is to use a parameter $\beta \in (0, \infty)$ to control the **precision**, or inverse variance, of the distribution:

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x-\mu)^2\right). \tag{3.22}$$

Normal distributions are a sensible choice for many applications. In the absence of prior knowledge about what form a distribution over the real numbers should take, the normal distribution is a good default choice for two major reasons.

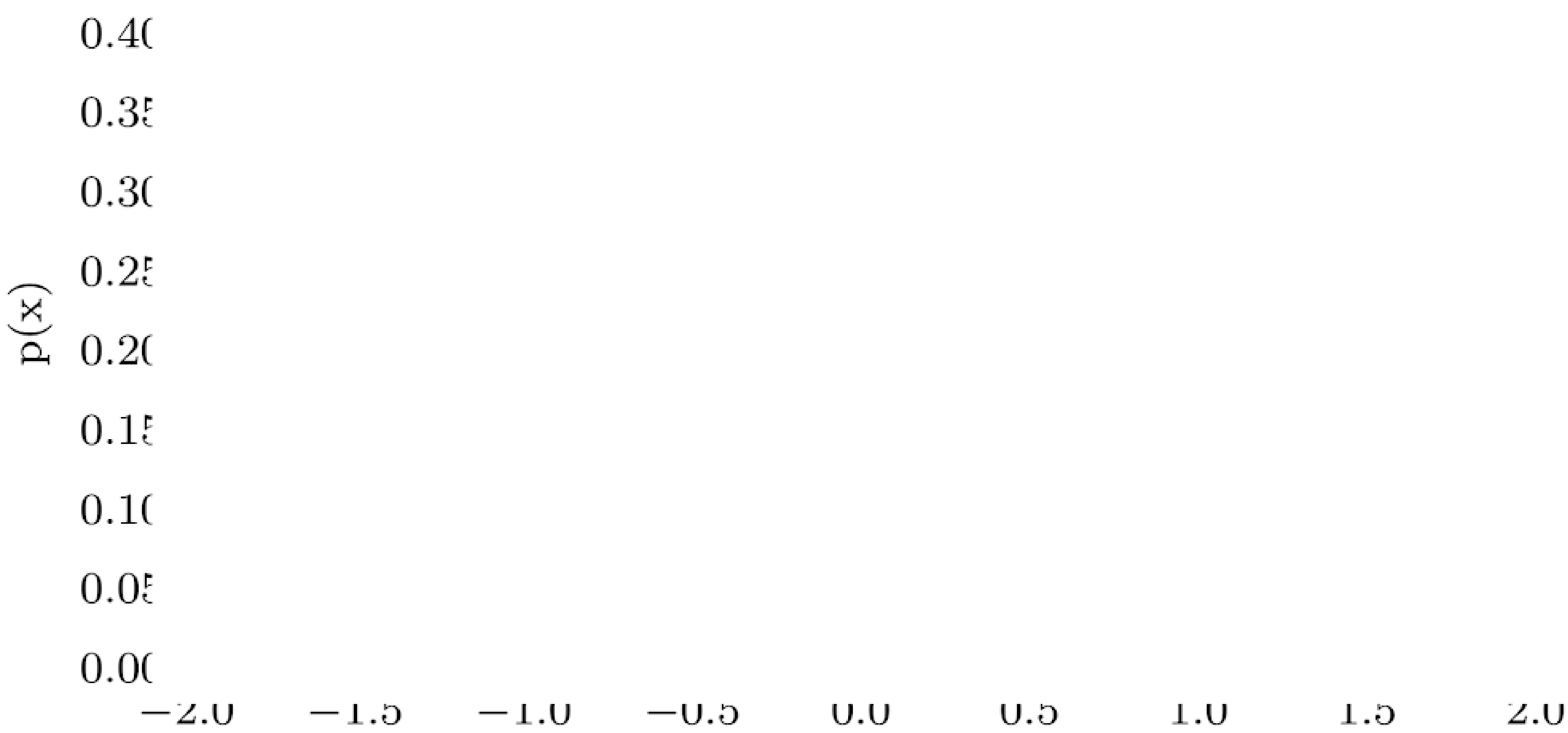


Figure 3.1: The normal distribution. The normal distribution $\mathcal{N}(x; \mu, \sigma^2)$ exhibits a classic “bell curve” shape, with the x coordinate of its central peak given by μ, and the width of its peak controlled by σ. In this example, we depict the **standard normal distribution**, with $\mu = 0$ and $\sigma = 1$.

First, many distributions we wish to model are truly close to being normal distributions. The **central limit theorem** shows that the sum of many independent random variables is approximately normally distributed. This means that in practice, many complicated systems can be modeled successfully as normally distributed noise, even if the system can be decomposed into parts with more structured behavior.

Second, out of all possible probability distributions with the same variance, the normal distribution encodes the maximum amount of uncertainty over the real numbers. We can thus think of the normal distribution as being the one that inserts the least amount of prior knowledge into a model. Fully developing and justifying this idea requires more mathematical tools and is postponed to section 19.4.2.

The normal distribution generalizes to $\mathbb{R}^n$, in which case it is known as the **multivariate normal distribution**. It may be parametrized with a positive definite symmetric matrix $\boldsymbol{\Sigma}$:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \quad \text{Copyrighted image} \quad \text{xp}\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right). \tag{3.23}$$

The parameter $\boldsymbol{\mu}$ still gives the mean of the distribution, though now it is vector valued. The parameter $\boldsymbol{\Sigma}$ gives the covariance matrix of the distribution.

As in the univariate case, when we wish to evaluate the PDF several times for many different values of the parameters, the covariance is not a computationally efficient way to parametrize the distribution, since we need to invert $\boldsymbol{\Sigma}$ to evaluate the PDF. We can instead use a **precision matrix $\boldsymbol{\beta}$**:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \qquad\qquad \mathrm{xp}\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\beta}(\boldsymbol{x}-\boldsymbol{\mu})\right). \tag{3.24}$$

We often fix the covariance matrix to be a diagonal matrix. An even simpler version is the **isotropic** Gaussian distribution, whose covariance matrix is a scalar times the identity matrix.

3.9.4 Exponential and Laplace Distributions

In the context of deep learning, we often want to have a probability distribution with a sharp point at $x = 0$. To accomplish this, we can use the **exponential distribution**:

$$p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x). \tag{3.25}$$

The exponential distribution uses the indicator function $\mathbf{1}_{x \geq 0}$ to assign probability zero to all negative values of x.

A closely related probability distribution that allows us to place a sharp peak of probability mass at an arbitrary point μ is the **Laplace distribution**

$$\text{Laplace}(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x-\mu|}{\gamma}\right). \tag{3.26}$$

3.9.5 The Dirac Distribution and Empirical Distribution

In some cases, we wish to specify that all the mass in a probability distribution clusters around a single point. This can be accomplished by defining a PDF using the **Dirac delta function**, $\delta(x)$:

$$p(x) = \delta(x - \mu). \tag{3.27}$$

The Dirac delta function is defined such that it is zero valued everywhere except 0, yet integrates to 1. The Dirac delta function is not an ordinary function that associates each value x with a real-valued output; instead it is a different kind of mathematical object called a **generalized function** that is defined in terms of its properties when integrated. We can think of the Dirac delta function as being the limit point of a series of functions that put less and less mass on all points other than zero.

By defining $p(x)$ to be δ shifted by $-\mu$ we obtain an infinitely narrow and infinitely high peak of probability mass where $x = \mu$.

A common use of the Dirac delta distribution is as a component of an **empirical distribution**,

$$\hat{p}(\boldsymbol{x}) = \frac{1}{m} \sum_{i=1}^{m} \delta(\boldsymbol{x} - \boldsymbol{x}^{(i)}) \tag{3.28}$$

which puts probability mass $\frac{1}{m}$ on each of the m points $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$, forming a given data set or collection of samples. The Dirac delta distribution is only necessary to define the empirical distribution over continuous variables. For discrete variables, the situation is simpler: an empirical distribution can be conceptualized as a multinoulli distribution, with a probability associated with each possible input value that is simply equal to the **empirical frequency** of that value in the training set.

We can view the empirical distribution formed from a dataset of training examples as specifying the distribution that we sample from when we train a model on this dataset. Another important perspective on the empirical distribution is that it is the probability density that maximizes the likelihood of the training data (see section 5.5).

3.9.6 Mixtures of Distributions

It is also common to define probability distributions by combining other simpler probability distributions. One common way of combining distributions is to construct a **mixture distribution**. A mixture distribution is made up of several component distributions. On each trial, the choice of which component distribution should generate the sample is determined by sampling a component identity from a multinoulli distribution:

$$P(\mathrm{x}) = \sum_{i} P(\mathrm{c} = i) P(\mathrm{x} \mid \mathrm{c} = i), \tag{3.29}$$

where $P(\mathrm{c})$ is the multinoulli distribution over component identities.

We have already seen one example of a mixture distribution: the empirical distribution over real-valued variables is a mixture distribution with one Dirac component for each training example.

The mixture model is one simple strategy for combining probability distributions to create a richer distribution. In chapter 16, we explore the art of building complex probability distributions from simple ones in more detail.

The mixture model allows us to briefly glimpse a concept that will be of paramount importance later—the **latent variable**. A latent variable is a random variable that we cannot observe directly. The component identity variable c of the mixture model provides an example. Latent variables may be related to x through the joint distribution, in this case, $P(\mathrm{x}, \mathrm{c}) = P(\mathrm{x} \mid \mathrm{c})P(\mathrm{c})$. The distribution $P(\mathrm{c})$ over the latent variable and the distribution $P(\mathrm{x} \mid \mathrm{c})$ relating the latent variables to the visible variables determines the shape of the distribution $P(\mathrm{x})$, even though it is possible to describe $P(\mathrm{x})$ without reference to the latent variable. Latent variables are discussed further in section 16.5.

A very powerful and common type of mixture model is the **Gaussian mixture model**, in which the components $p(\mathbf{x} \mid \mathrm{c} = i)$ are Gaussians. Each component has a separately parametrized mean $\boldsymbol{\mu}^{(i)}$ and covariance $\boldsymbol{\Sigma}^{(i)}$. Some mixtures can have more constraints. For example, the covariances could be shared across components via the constraint $\boldsymbol{\Sigma}^{(i)} = \boldsymbol{\Sigma}, \forall i$. As with a single Gaussian distribution, the mixture of Gaussians might constrain the covariance matrix for each component to be diagonal or isotropic.

In addition to the means and covariances, the parameters of a Gaussian mixture specify the **prior probability** $\alpha_i = P(\mathrm{c} = i)$ given to each component i. The word “prior” indicates that it expresses the model's beliefs about c *before* it has observed $\mathbf{x}$. By comparison, $P(\mathrm{c} \mid \boldsymbol{x})$ is a **posterior probability**, because it is computed *after* observation of $\mathbf{x}$. A Gaussian mixture model is a **universal approximator** of densities, in the sense that any smooth density can be approximated with any specific nonzero amount of error by a Gaussian mixture model with enough components.

Figure 3.2 shows samples from a Gaussian mixture model.

3.10 Useful Properties of Common Functions

Certain functions arise often while working with probability distributions, especially the probability distributions used in deep learning models.

One of these functions is the **logistic sigmoid**:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{3.30}$$

The function $\sigma^{-1}(x)$ is called the **logit** in statistics, but this term is rarely used in machine learning.

Equation 3.41 provides extra justification for the name "softplus." The softplus function is intended as a smoothed version of the **positive part function**, $x^+ = \max\{0, x\}$. The positive part function is the counterpart of the **negative part function**, $x^- = \max\{0, -x\}$. To obtain a smooth function that is analogous to the negative part, one can use $\zeta(-x)$. Just as x can be recovered from its positive part and its negative part via the identity $x^+ - x^- = x$, it is also possible to recover x using the same relationship between $\zeta(x)$ and $\zeta(-x)$, as shown in equation 3.41.

3.11 Bayes' Rule

We often find ourselves in a situation where we know $P(\mathrm{y} \mid \mathrm{x})$ and need to know $P(\mathrm{x} \mid \mathrm{y})$. Fortunately, if we also know $P(\mathrm{x})$, we can compute the desired quantity using **Bayes' rule**:

$$P(\mathrm{x} \mid \mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y} \mid \mathrm{x})}{P(\mathrm{y})}. \tag{3.42}$$

Note that while $P(\mathrm{y})$ appears in the formula, it is usually feasible to compute $P(\mathrm{y}) = \sum_x P(\mathrm{y} \mid x)P(x)$, so we do not need to begin with knowledge of $P(\mathrm{y})$.

Bayes' rule is straightforward to derive from the definition of conditional probability, but it is useful to know the name of this formula since many texts refer to it by name. It is named after the Reverend Thomas Bayes, who first discovered a special case of the formula. The general version presented here was independently discovered by Pierre-Simon Laplace.

3.12 Technical Details of Continuous Variables

A proper formal understanding of continuous random variables and probability density functions requires developing probability theory in terms of a branch of mathematics known as **measure theory**. Measure theory is beyond the scope of this textbook, but we can briefly sketch some of the issues that measure theory is employed to resolve.

In section 3.3.2, we saw that the probability of a continuous vector-valued $\mathbf{x}$ lying in some set $\mathbb{S}$ is given by the integral of $p(\boldsymbol{x})$ over the set $\mathbb{S}$. Some choices of set $\mathbb{S}$ can produce paradoxes. For example, it is possible to construct two sets $\mathbb{S}_1$ and $\mathbb{S}_2$ such that $p(\boldsymbol{x} \in \mathbb{S}_1) + p(\boldsymbol{x} \in \mathbb{S}_2) > 1$ but $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$. These sets are generally constructed making very heavy use of the infinite precision of real numbers, for example by making fractal-shaped sets or sets that are defined by transforming the set of rational numbers.[2] One of the key contributions of measure theory is to provide a characterization of the set of sets we can compute the probability of without encountering paradoxes. In this book, we integrate only over sets with relatively simple descriptions, so this aspect of measure theory never becomes a relevant concern.

For our purposes, measure theory is more useful for describing theorems that apply to most points in $\mathbb{R}^n$ but do not apply to some corner cases. Measure theory provides a rigorous way of describing that a set of points is negligibly small. Such a set is said to have **measure zero**. We do not formally define this concept in this textbook. For our purposes, it is sufficient to understand the intuition that a set of measure zero occupies no volume in the space we are measuring. For example, within $\mathbb{R}^2$, a line has measure zero, while a filled polygon has positive measure. Likewise, an individual point has measure zero. Any union of countably many sets that each have measure zero also has measure zero (so the set of all the rational numbers has measure zero, for instance).

Another useful term from measure theory is **almost everywhere**. A property that holds almost everywhere holds throughout all space except for on a set of measure zero. Because the exceptions occupy a negligible amount of space, they can be safely ignored for many applications. Some important results in probability theory hold for all discrete values but hold "almost everywhere" only for continuous values.

Another technical detail of continuous variables relates to handling continuous random variables that are deterministic functions of one another. Suppose we have two random variables, $\mathbf{x}$ and $\mathbf{y}$, such that $\boldsymbol{y} = g(\boldsymbol{x})$, where g is an invertible, continuous, differentiable transformation. One might expect that $p_y(\boldsymbol{y}) = p_x(g^{-1}(\boldsymbol{y}))$. This is actually not the case.

As a simple example, suppose we have scalar random variables x and y. Suppose $\mathrm{y} = \frac{\mathrm{x}}{2}$ and $\mathrm{x} \sim U(0,1)$. If we use the rule $p_y(y) = p_x(2y)$ then p_y will be 0 everywhere except the interval $[0, \frac{1}{2}]$, and it will be 1 on this interval. This means

$$\int p_y(y)dy = \frac{1}{2}, \tag{3.43}$$

which violates the definition of a probability distribution. This is a common mistake. The problem with this approach is that it fails to account for the distortion of space introduced by the function g. Recall that the probability of $\boldsymbol{x}$ lying in an infinitesimally small region with volume $\delta\boldsymbol{x}$ is given by $p(\boldsymbol{x})\delta\boldsymbol{x}$. Since g can expand

---

[2] The Banach-Tarski theorem provides a fun example of such sets.

or contract space, the infinitesimal volume surrounding $\boldsymbol{x}$ in $\boldsymbol{x}$ space may have different volume in $\boldsymbol{y}$ space.

To see how to correct the problem, we return to the scalar case. We need to preserve the property

$$|p_y(g(x))dy| = |p_x(x)dx|. \tag{3.44}$$

Solving from this, we obtain

$$p_y(y) = p_x(g^{-1}(y)) \left| \frac{\partial x}{\partial y} \right| \tag{3.45}$$

or equivalently

$$p_x(x) = p_y(g(x)) \left| \frac{\partial g(x)}{\partial x} \right|. \tag{3.46}$$

In higher dimensions, the derivative generalizes to the determinant of the **Jacobian matrix**—the matrix with $J_{i,j} = \frac{\partial x_i}{\partial y_j}$. Thus, for real-valued vectors $\boldsymbol{x}$ and $\boldsymbol{y}$,

$$p_x(\boldsymbol{x}) = p_y(g(\boldsymbol{x})) \left| \det \left(\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}} \right) \right|. \tag{3.47}$$

3.13 Information Theory

Information theory is a branch of applied mathematics that revolves around quantifying how much information is present in a signal. It was originally invented to study sending messages from discrete alphabets over a noisy channel, such as communication via radio transmission. In this context, information theory tells how to design optimal codes and calculate the expected length of messages sampled from specific probability distributions using various encoding schemes. In the context of machine learning, we can also apply information theory to continuous variables where some of these message length interpretations do not apply. This field is fundamental to many areas of electrical engineering and computer science. In this textbook, we mostly use a few key ideas from information theory to characterize probability distributions or to quantify similarity between probability distributions. For more detail on information theory, see Cover and Thomas (2006) or MacKay (2003).

The basic intuition behind information theory is that learning that an unlikely event has occurred is more informative than learning that a likely event has occurred. A message saying "the sun rose this morning" is so uninformative as to be unnecessary to send, but a message saying "there was a solar eclipse this morning" is very informative.

We would like to quantify information in a way that formalizes this intuition.

- Likely events should have low information content, and in the extreme case, events that are guaranteed to happen should have no information content whatsoever.
- Less likely events should have higher information content.
- Independent events should have additive information. For example, finding out that a tossed coin has come up as heads twice should convey twice as much information as finding out that a tossed coin has come up as heads once.

To satisfy all three of these properties, we define the **self-information** of an event $\mathrm{x} = x$ to be

$$I(x) = -\log P(x). \tag{3.48}$$

In this book, we always use log to mean the natural logarithm, with base e. Our definition of $I(x)$ is therefore written in units of **nats**. One nat is the amount of information gained by observing an event of probability $\frac{1}{e}$. Other texts use base-2 logarithms and units called **bits** or **shannons**; information measured in bits is just a rescaling of information measured in nats.

When x is continuous, we use the same definition of information by analogy, but some of the properties from the discrete case are lost. For example, an event with unit density still has zero information, despite not being an event that is guaranteed to occur.

Self-information deals only with a single outcome. We can quantify the amount of uncertainty in an entire probability distribution using the **Shannon entropy**,

$$H(\mathrm{x}) = \mathbb{E}_{\mathrm{x}\sim P}[I(x)] = -\mathbb{E}_{\mathrm{x}\sim P}[\log P(x)], \tag{3.49}$$

also denoted $H(P)$. In other words, the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution. It gives a lower bound on the number of bits (if the logarithm is base 2, otherwise the units are different) needed on average to encode symbols drawn from a distribution P. Distributions that are nearly deterministic (where the outcome is nearly certain) have low entropy; distributions that are closer to uniform have high entropy. See figure 3.5 for a demonstration. When x is continuous, the Shannon entropy is known as the **differential entropy**.

If we have two separate probability distributions $P(\mathrm{x})$ and $Q(\mathrm{x})$ over the same random variable x, we can measure how different these two distributions are using

Figure 3.5: Shannon entropy of a binary random variable. This plot shows how distributions that are closer to deterministic have low Shannon entropy while distributions that are close to uniform have high Shannon entropy. On the horizontal axis, we plot p, the probability of a binary random variable being equal to 1. The entropy is given by $(p-1)\log(1-p)-p\log p$. When p is near 0, the distribution is nearly deterministic, because the random variable is nearly always 0. When p is near 1, the distribution is nearly deterministic, because the random variable is nearly always 1. When $p=0.5$, the entropy is maximal, because the distribution is uniform over the two outcomes.

the **Kullback-Leibler (KL) divergence**:

$$D_{\mathrm{KL}}(P\|Q)=\mathbb{E}_{\mathrm{x}\sim P}\left[\log\frac{P(x)}{Q(x)}\right]=\mathbb{E}_{\mathrm{x}\sim P}\left[\log P(x)-\log Q(x)\right]. \tag{3.50}$$

In the case of discrete variables, it is the extra amount of information (measured in bits if we use the base-2 logarithm, but in machine learning we usually use nats and the natural logarithm) needed to send a message containing symbols drawn from probability distribution P, when we use a code that was designed to minimize the length of messages drawn from probability distribution Q.

The KL divergence has many useful properties, most notably being non-negative. The KL divergence is 0 if and only if P and Q are the same distribution in the case of discrete variables, or equal "almost everywhere" in the case of continuous variables. Because the KL divergence is non-negative and measures the difference between two distributions, it is often conceptualized as measuring some sort of distance between these distributions. It is not a true distance measure because it is not symmetric: $D_{\mathrm{KL}}(P\|Q)\neq D_{\mathrm{KL}}(Q\|P)$ for some P and Q. This asymmetry means that there are important consequences to the choice of whether to use $D_{\mathrm{KL}}(P\|Q)$ or $D_{\mathrm{KL}}(Q\|P)$. See figure 3.6 for more detail.

Figure 3.7: A directed graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{a}, \mathrm{b})p(\mathrm{d} \mid \mathrm{b})p(\mathrm{e} \mid \mathrm{c}). \tag{3.54}$$

This graphical model enables us to quickly see some properties of the distribution. For example, a and c interact directly, but a and e interact only indirectly via c.

See figure 3.7 for an example of a directed graph and the factorization of probability distributions it represents.

Undirected models use graphs with undirected edges, and they represent factorizations into a set of functions; unlike in the directed case, these functions are usually not probability distributions of any kind. Any set of nodes that are all connected to each other in $\mathcal{G}$ is called a clique. Each clique $\mathcal{C}^{(i)}$ in an undirected model is associated with a factor $\phi^{(i)}(\mathcal{C}^{(i)})$. These factors are just functions, not probability distributions. The output of each factor must be non-negative, but there is no constraint that the factor must sum or integrate to 1 like a probability distribution.

The probability of a configuration of random variables is **proportional** to the product of all these factors—assignments that result in larger factor values are more likely. Of course, there is no guarantee that this product will sum to 1. We therefore divide by a normalizing constant Z, defined to be the sum or integral over all states of the product of the ϕ functions, in order to obtain a normalized probability distribution:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)} \left(\mathcal{C}^{(i)} \right). \tag{3.55}$$

See figure 3.8 for an example of an undirected graph and the factorization of probability distributions it represents.

Figure 3.8: An undirected graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = \frac{1}{Z} \phi^{(1)}(\mathrm{a}, \mathrm{b}, \mathrm{c}) \phi^{(2)}(\mathrm{b}, \mathrm{d}) \phi^{(3)}(\mathrm{c}, \mathrm{e}). \tag{3.56}$$

This graphical model enables us to quickly see some properties of the distribution. For example, a and c interact directly, but a and e interact only indirectly via c.

Keep in mind that these graphical representations of factorizations are a language for describing probability distributions. They are not mutually exclusive families of probability distributions. Being directed or undirected is not a property of a probability distribution; it is a property of a particular **description** of a probability distribution, but any probability distribution may be described in both ways.

Throughout parts I and II of this book, we use structured probabilistic models merely as a language to describe which direct probabilistic relationships different machine learning algorithms choose to represent. No further understanding of structured probabilistic models is needed until the discussion of research topics, in part III, where we explore structured probabilistic models in much greater detail.

This chapter has reviewed the basic concepts of probability theory that are most relevant to deep learning. One more set of fundamental mathematical tools remains: numerical methods.

4

Numerical Computation

Machine learning algorithms usually require a high amount of numerical computation. This typically refers to algorithms that solve mathematical problems by methods that update estimates of the solution via an iterative process, rather than analytically deriving a formula to provide a symbolic expression for the correct solution. Common operations include optimization (finding the value of an argument that minimizes or maximizes a function) and solving systems of linear equations. Even just evaluating a mathematical function on a digital computer can be difficult when the function involves real numbers, which cannot be represented precisely using a finite amount of memory.

4.1 Overflow and Underflow

The fundamental difficulty in performing continuous math on a digital computer is that we need to represent infinitely many real numbers with a finite number of bit patterns. This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer. In many cases, this is just rounding error. Rounding error is problematic, especially when it compounds across many operations, and can cause algorithms that work in theory to fail in practice if they are not designed to minimize the accumulation of rounding error.

One form of rounding error that is particularly devastating is **underflow**. Underflow occurs when numbers near zero are rounded to zero. Many functions behave qualitatively differently when their argument is zero rather than a small positive number. For example, we usually want to avoid division by zero (some software environments will raise exceptions when this occurs, others will return a result with a placeholder not-a-number value) or taking the logarithm of zero (this is usually treated as $-\infty$, which then becomes not-a-number if it is used for many further arithmetic operations).

Another highly damaging form of numerical error is **overflow**. Overflow occurs when numbers with large magnitude are approximated as ∞ or $-\infty$. Further arithmetic will usually change these infinite values into not-a-number values.

One example of a function that must be stabilized against underflow and overflow is the **softmax function**. The softmax function is often used to predict the probabilities associated with a multinoulli distribution. The softmax function is defined to be

$$\operatorname{softmax}(\boldsymbol{x})_i = \tag{4.1}$$

Consider what happens when all the x_i are equal to some constant c. Analytically, we can see that all the outputs should be equal to $\frac{1}{n}$. Numerically, this may not occur when c has large magnitude. If c is very negative, then $\exp(c)$ will underflow. This means the denominator of the softmax will become 0, so the final result is undefined. When c is very large and positive, $\exp(c)$ will overflow, again resulting in the expression as a whole being undefined. Both of these difficulties can be resolved by instead evaluating $\operatorname{softmax}(\boldsymbol{z})$ where $\boldsymbol{z} = \boldsymbol{x} - \max_i x_i$. Simple algebra shows that the value of the softmax function is not changed analytically by adding or subtracting a scalar from the input vector. Subtracting $\max_i x_i$ results in the largest argument to exp being 0, which rules out the possibility of overflow. Likewise, at least one term in the denominator has a value of 1, which rules out the possibility of underflow in the denominator leading to a division by zero.

There is still one small problem. Underflow in the numerator can still cause the expression as a whole to evaluate to zero. This means that if we implement $\log \operatorname{softmax}(\boldsymbol{x})$ by first running the softmax subroutine then passing the result to the log function, we could erroneously obtain $-\infty$. Instead, we must implement a separate function that calculates $\log \operatorname{softmax}$ in a numerically stable way. The $\log \operatorname{softmax}$ function can be stabilized using the same trick as we used to stabilize the softmax function.

For the most part, we do not explicitly detail all the numerical considerations involved in implementing the various algorithms described in this book. Developers of low-level libraries should keep numerical issues in mind when implementing deep learning algorithms. Most readers of this book can simply rely on low-level libraries that provide stable implementations. In some cases, it is possible to implement a new algorithm and have the new implementation automatically stabilized. Theano (Bergstra et al., 2010; Bastien et al., 2012) is an example of a software package that automatically detects and stabilizes many common numerically unstable expressions that arise in the context of deep learning.

4.2 Poor Conditioning

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

Consider the function $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. When $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has an eigenvalue decomposition, its **condition number** is

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right| . \tag{4.2}$$

This is the ratio of the magnitude of the largest and smallest eigenvalue. When this number is large, matrix inversion is particularly sensitive to error in the input.

This sensitivity is an intrinsic property of the matrix itself, not the result of rounding error during matrix inversion. Poorly conditioned matrices amplify pre-existing errors when we multiply by the true matrix inverse. In practice, the error will be compounded further by numerical errors in the inversion process itself.

4.3 Gradient-Based Optimization

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$. We usually phrase most optimization problems in terms of minimizing $f(\boldsymbol{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\boldsymbol{x})$.

The function we want to minimize or maximize is called the **objective function**, or **criterion**. When we are minimizing it, we may also call it the **cost function**, **loss function**, or **error function**. In this book, we use these terms interchangeably, though some machine learning publications assign special meaning to some of these terms.

We often denote the value that minimizes or maximizes a function with a superscript $*$. For example, we might say $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$.

We assume the reader is already familiar with calculus but provide a brief review of how calculus concepts relate to optimization here.

For functions with multiple inputs, we must make use of the concept of **partial derivatives**. The partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ measures how f changes as only the variable x_i increases at point $\boldsymbol{x}$. The **gradient** generalizes the notion of derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all the partial derivatives, denoted $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Element i of the gradient is the partial derivative of f with respect to x_i. In multiple dimensions, critical points are points where every element of the gradient is equal to zero.

The **directional derivative** in direction $\boldsymbol{u}$ (a unit vector) is the slope of the function f in direction u. In other words, the directional derivative is the derivative of the function $f(\boldsymbol{x} + \alpha \boldsymbol{u})$ with respect to α, evaluated at $\alpha = 0$. Using the chain rule, we can see that $\frac{\partial}{\partial \alpha} f(\boldsymbol{x} + \alpha \boldsymbol{u})$ evaluates to $\boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ when $\alpha = 0$.

To minimize f, we would like to find the direction in which f decreases the fastest. We can do this using the directional derivative:

$$\min_{\boldsymbol{u}, \boldsymbol{u}^\top \boldsymbol{u} = 1} \boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{4.3}$$

$$= \min_{\boldsymbol{u}, \boldsymbol{u}^\top \boldsymbol{u} = 1} ||\boldsymbol{u}||_2 ||\nabla_{\boldsymbol{x}} f(\boldsymbol{x})||_2 \cos \theta \tag{4.4}$$

where θ is the angle between $\boldsymbol{u}$ and the gradient. Substituting in $||\boldsymbol{u}||_2 = 1$ and ignoring factors that do not depend on $\boldsymbol{u}$, this simplifies to $\min_{\boldsymbol{u}} \cos \theta$. This is minimized when $\boldsymbol{u}$ points in the opposite direction as the gradient. In other words, the gradient points directly uphill, and the negative gradient points directly downhill. We can decrease f by moving in the direction of the negative gradient. This is known as the **method of steepest descent**, or **gradient descent**.

Steepest descent proposes a new point

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{4.5}$$

where ϵ is the **learning rate**, a positive scalar determining the size of the step. We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f(\boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$ for several values of ϵ and choose the one that results in the smallest objective function value. This last strategy is called a **line search**.

Steepest descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we may be able to avoid running this iterative algorithm and just jump directly to the critical point by solving the equation $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$ for $\boldsymbol{x}$.

Although gradient descent is limited to optimization in continuous spaces, the general concept of repeatedly making a small move (that is approximately the best small move) toward better configurations can be generalized to discrete spaces. Ascending an objective function of discrete parameters is called **hill climbing** (Russel and Norvig, 2003).

4.3.1 Beyond the Gradient: Jacobian and Hessian Matrices

Sometimes we need to find all the partial derivatives of a function whose input and output are both vectors. The matrix containing all such partial derivatives is known as a **Jacobian matrix**. Specifically, if we have a function $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n$, then the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ of $\boldsymbol{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$.

We are also sometimes interested in a derivative of a derivative. This is known as a **second derivative**. For example, for a function $f : \mathbb{R}^n \to \mathbb{R}$, the derivative with respect to x_i of the derivative of f with respect to x_j is denoted as $\frac{\partial^2}{\partial x_i \partial x_j} f$. In a single dimension, we can denote $\frac{d^2}{dx^2} f$ by $f''(x)$. The second derivative tells us how the first derivative will change as we vary the input. This is important because it tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone. We can think of the second derivative as measuring **curvature**. Suppose we have a quadratic function (many functions that arise in practice are not quadratic but can be approximated well as quadratic, at least locally). If such a function has a second derivative of zero, then there is no curvature. It is a perfectly flat line, and its value can be predicted using only the gradient. If the gradient is 1, then we can make a step of size ϵ along the negative gradient, and the cost function will decrease by ϵ. If the second derivative is negative, the function curves downward, so the cost function will actually decrease by more than ϵ. Finally, if the second derivative is positive, the function curves upward, so the cost function can decrease by less than ϵ. See figure 4.4 to see how different forms of curvature affect the relationship between the value of the cost function predicted by the gradient and the true value.

When our function has multiple input dimensions, there are many second derivatives. These derivatives can be collected together into a matrix called the **Hessian matrix**. The Hessian matrix $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}). \tag{4.6}$$

Equivalently, the Hessian is the Jacobian of the gradient.

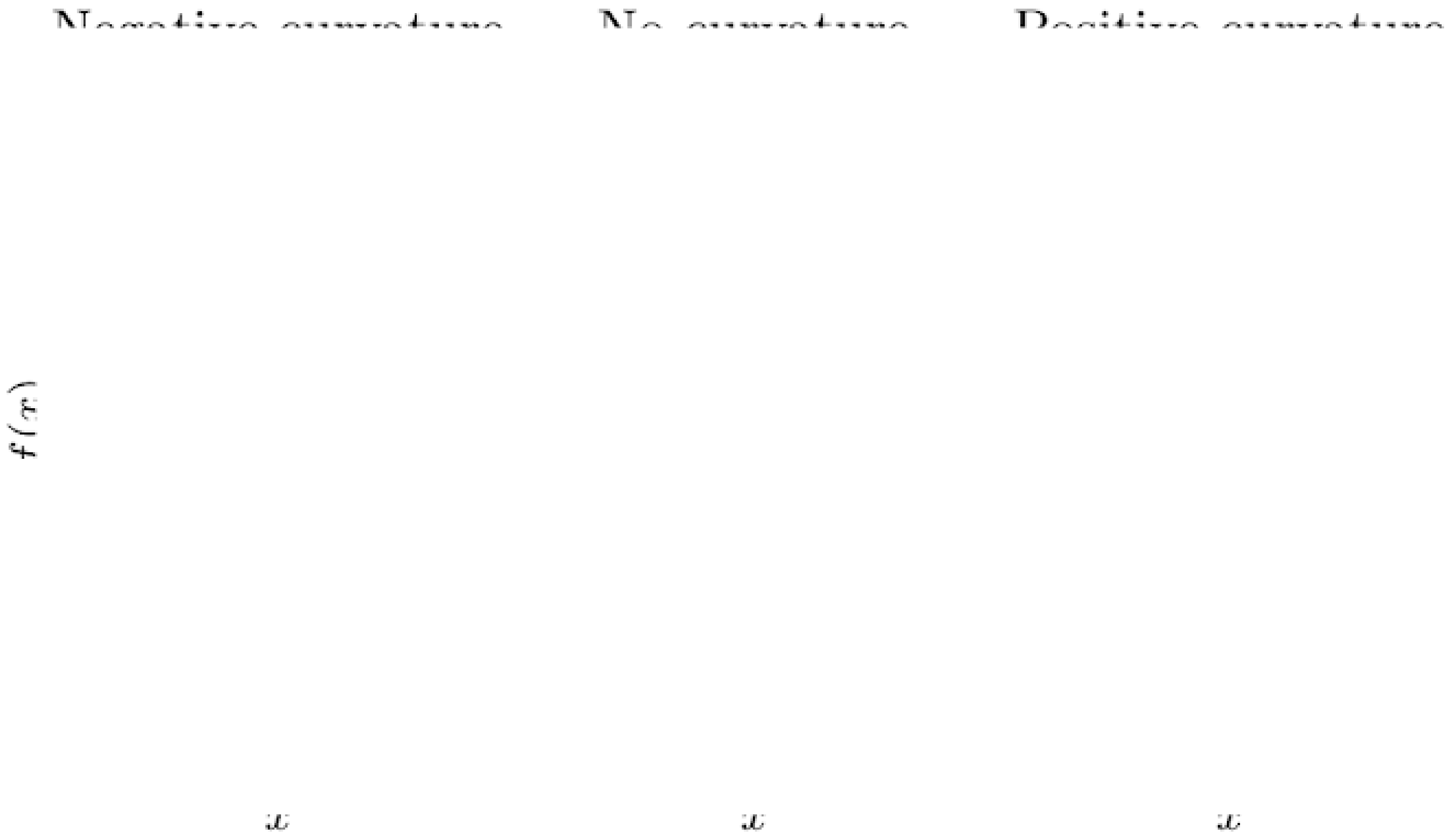


Figure 4.4: The second derivative determines the curvature of a function. Here we show quadratic functions with various curvature. The dashed line indicates the value of the cost function we would expect based on the gradient information alone as we make a gradient step downhill. With negative curvature, the cost function actually decreases faster than the gradient predicts. With no curvature, the gradient predicts the decrease correctly. With positive curvature, the function decreases more slowly than expected and eventually begins to increase, so steps that are too large can actually increase the function inadvertently.

Anywhere that the second partial derivatives are continuous, the differential operators are commutative; that is, their order can be swapped:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\boldsymbol{x}). \tag{4.7}$$

This implies that $H_{i,j} = H_{j,i}$, so the Hessian matrix is symmetric at such points. Most of the functions we encounter in the context of deep learning have a symmetric Hessian almost everywhere. Because the Hessian matrix is real and symmetric, we can decompose it into a set of real eigenvalues and an orthogonal basis of eigenvectors. The second derivative in a specific direction represented by a unit vector $\boldsymbol{d}$ is given by $\boldsymbol{d}^\top \boldsymbol{H} \boldsymbol{d}$. When $\boldsymbol{d}$ is an eigenvector of $\boldsymbol{H}$, the second derivative in that direction is given by the corresponding eigenvalue. For other directions of $\boldsymbol{d}$, the directional second derivative is a weighted average of all the eigenvalues, with weights between 0 and 1, and eigenvectors that have a smaller angle with $\boldsymbol{d}$ receiving more weight. The maximum eigenvalue determines the maximum second derivative, and the minimum eigenvalue determines the minimum second derivative.

The (directional) second derivative tells us how well we can expect a gradient descent step to perform. We can make a second-order Taylor series approximation to the function $f(\boldsymbol{x})$ around the current point $\boldsymbol{x}^{(0)}$:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)}), \tag{4.8}$$

where $\boldsymbol{g}$ is the gradient and $\boldsymbol{H}$ is the Hessian at $\boldsymbol{x}^{(0)}$. If we use a learning rate of ϵ, then the new point $\boldsymbol{x}$ will be given by $\boldsymbol{x}^{(0)} - \epsilon\boldsymbol{g}$. Substituting this into our approximation, we obtain

$$f(\boldsymbol{x}^{(0)} - \epsilon\boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon\boldsymbol{g}^\top\boldsymbol{g} + \frac{1}{2}\epsilon^2\boldsymbol{g}^\top\boldsymbol{H}\boldsymbol{g}. \tag{4.9}$$

There are three terms here: the original value of the function, the expected improvement due to the slope of the function, and the correction we must apply to account for the curvature of the function. When this last term is too large, the gradient descent step can actually move uphill. When $\boldsymbol{g}^\top\boldsymbol{H}\boldsymbol{g}$ is zero or negative, the Taylor series approximation predicts that increasing ϵ forever will decrease f forever. In practice, the Taylor series is unlikely to remain accurate for large ϵ, so one must resort to more heuristic choices of ϵ in this case. When $\boldsymbol{g}^\top\boldsymbol{H}\boldsymbol{g}$ is positive, solving for the optimal step size that decreases the Taylor series approximation of the function the most yields

$$\epsilon^* = \frac{\boldsymbol{g}^\top\boldsymbol{g}}{\boldsymbol{g}^\top\boldsymbol{H}\boldsymbol{g}}. \tag{4.10}$$

In the worst case, when $\boldsymbol{g}$ aligns with the eigenvector of $\boldsymbol{H}$ corresponding to the maximal eigenvalue $\lambda_{\max}$, then this optimal step size is given by $\frac{1}{\lambda_{\max}}$. To the extent that the function we minimize can be approximated well by a quadratic function, the eigenvalues of the Hessian thus determine the scale of the learning rate.

The second derivative can be used to determine whether a critical point is a local maximum, a local minimum, or a saddle point. Recall that on a critical point, $f'(x) = 0$. When the second derivative $f''(x) > 0$, the first derivative $f'(x)$ increases as we move to the right and decreases as we move to the left. This means $f'(x - \epsilon) < 0$ and $f'(x + \epsilon) > 0$ for small enough ϵ. In other words, as we move right, the slope begins to point uphill to the right, and as we move left, the slope begins to point uphill to the left. Thus, when $f'(x) = 0$ and $f''(x) > 0$, we can conclude that x is a local minimum. Similarly, when $f'(x) = 0$ and $f''(x) < 0$, we can conclude that x is a local maximum. This is known as the **second derivative test**. Unfortunately, when $f''(x) = 0$, the test is inconclusive. In this case x may be a saddle point or a part of a flat region.

In multiple dimensions, we need to examine all the second derivatives of the function. Using the eigendecomposition of the Hessian matrix, we can generalize the second derivative test to multiple dimensions. At a critical point, where $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 0$, we can examine the eigenvalues of the Hessian to determine whether the critical point is a local maximum, local minimum, or saddle point. When the Hessian is positive definite (all its eigenvalues are positive), the point is a local minimum. This can be seen by observing that the directional second derivative in any direction must be positive, and making reference to the univariate second derivative test. Likewise, when the Hessian is negative definite (all its eigenvalues are negative), the point is a local maximum. In multiple dimensions, it is actually possible to find positive evidence of saddle points in some cases. When at least one eigenvalue is positive and at least one eigenvalue is negative, we know that $\boldsymbol{x}$ is a local maximum on one cross section of f but a local minimum on another cross section. See figure 4.5 for an example. Finally, the multidimensional second derivative test can be inconclusive, just as the univariate version can. The test is inconclusive whenever all the nonzero eigenvalues have the same sign but at least one eigenvalue is zero. This is because the univariate second derivative test is inconclusive in the cross section corresponding to the zero eigenvalue.

In multiple dimensions, there is a different second derivative for each direction at a single point. The condition number of the Hessian at this point measures how much the second derivatives differ from each other. When the Hessian has a poor condition number, gradient descent performs poorly. This is because in one

Figure 4.5: A saddle point containing both positive and negative curvature. The function in this example is $f(\boldsymbol{x}) = x_1^2 - x_2^2$. Along the axis corresponding to x_1, the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding to x_2, the function curves downward. This direction is an eigenvector of the Hessian with negative eigenvalue. The name "saddle point" derives from the saddle-like shape of this function. This is the quintessential example of a function with a saddle point. In more than one dimension, it is not necessary to have an eigenvalue of 0 to get a saddle point: it is only necessary to have both positive and negative eigenvalues. We can think of a saddle point with both signs of eigenvalues as being a local maximum within one cross section and a local minimum within another cross section.

guarantees by making stronger restrictions. These algorithms are applicable only to convex functions—functions for which the Hessian is positive semidefinite everywhere. Such functions are well-behaved because they lack saddle points, and all their local minima are necessarily global minima. However, most problems in deep learning are difficult to express in terms of convex optimization. Convex optimization is used only as a subroutine of some deep learning algorithms. Ideas from the analysis of convex optimization algorithms can be useful for proving the convergence of deep learning algorithms, but in general, the importance of convex optimization is greatly diminished in the context of deep learning. For more information about convex optimization, see Boyd and Vandenberghe (2004) or Rockafellar (1997).

4.4 Constrained Optimization

Sometimes we wish not only to maximize or minimize a function $f(\boldsymbol{x})$ over all possible values of $\boldsymbol{x}$. Instead we may wish to find the maximal or minimal value of $f(\boldsymbol{x})$ for values of $\boldsymbol{x}$ in some set $\mathbb{S}$. This is known as **constrained optimization**. Points $\boldsymbol{x}$ that lie within the set $\mathbb{S}$ are called **feasible** points in constrained optimization terminology.

We often wish to find a solution that is small in some sense. A common approach in such situations is to impose a norm constraint, such as $||\boldsymbol{x}|| \leq 1$.

One simple approach to constrained optimization is simply to modify gradient descent taking the constraint into account. If we use a small constant step size ϵ, we can make gradient descent steps, then project the result back into $\mathbb{S}$. If we use a line search, we can search only over step sizes ϵ that yield new $\boldsymbol{x}$ points that are feasible, or we can project each point on the line back into the constraint region. When possible, this method can be made more efficient by projecting the gradient into the tangent space of the feasible region before taking the step or beginning the line search (Rosen, 1960).

A more sophisticated approach is to design a different, unconstrained optimization problem whose solution can be converted into a solution to the original, constrained optimization problem. For example, if we want to minimize $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^2$ with $\boldsymbol{x}$ constrained to have exactly unit L^2 norm, we can instead minimize $g(\theta) = f([\cos\theta, \sin\theta]^\top)$ with respect to θ, then return $[\cos\theta, \sin\theta]$ as the solution to the original problem. This approach requires creativity; the transformation between optimization problems must be designed specifically for each case we encounter.

The **Karush–Kuhn–Tucker** (KKT) approach[1] provides a very general solution to constrained optimization. With the KKT approach, we introduce a new function called the **generalized Lagrangian** or **generalized Lagrange function**.

To define the Lagrangian, we first need to describe $\mathbb{S}$ in terms of equations and inequalities. We want a description of $\mathbb{S}$ in terms of m functions $g^{(i)}$ and n functions $h^{(j)}$ so that $\mathbb{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\}$. The equations involving $g^{(i)}$ are called the **equality constraints**, and the inequalities involving $h^{(j)}$ are called **inequality constraints**.

We introduce new variables λ_i and α_j for each constraint, these are called the KKT multipliers. The generalized Lagrangian is then defined as

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \tag{4.14}$$

We can now solve a constrained minimization problem using unconstrained optimization of the generalized Lagrangian. As long as at least one feasible point exists and $f(\boldsymbol{x})$ is not permitted to have value ∞, then

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \tag{4.15}$$

has the same optimal objective function value and set of optimal points $\boldsymbol{x}$ as

$$\min_{\boldsymbol{x} \in \mathbb{S}} f(\boldsymbol{x}). \tag{4.16}$$

This follows because any time the constraints are satisfied,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}), \tag{4.17}$$

while any time a constraint is violated,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty. \tag{4.18}$$

These properties guarantee that no infeasible point can be optimal, and that the optimum within the feasible points is unchanged.

---

[1] The KKT approach generalizes the method of **Lagrange multipliers**, which allows equality constraints but not inequality constraints.

To perform constrained maximization, we can construct the generalized Lagrange function of $-f(\boldsymbol{x})$, which leads to this optimization problem:

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} -f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \tag{4.19}$$

We may also convert this to a problem with maximization in the outer loop:

$$\max_{\boldsymbol{x}} \min_{\boldsymbol{\lambda}} \min_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) - \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \tag{4.20}$$

The sign of the term for the equality constraints does not matter; we may define it with addition or subtraction as we wish, because the optimization is free to choose any sign for each λ_i.

The inequality constraints are particularly interesting. We say that a constraint $h^{(i)}(\boldsymbol{x})$ is **active** if $h^{(i)}(\boldsymbol{x}^*) = 0$. If a constraint is not active, then the solution to the problem found using that constraint would remain at least a local solution if that constraint were removed. It is possible that an inactive constraint excludes other solutions. For example, a convex problem with an entire region of globally optimal points (a wide, flat region of equal cost) could have a subset of this region eliminated by constraints, or a nonconvex problem could have better local stationary points excluded by a constraint that is inactive at convergence. Yet the point found at convergence remains a stationary point whether or not the inactive constraints are included. Because an inactive $h^{(i)}$ has negative value, then the solution to $\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ will have $\alpha_i = 0$. We can thus observe that at the solution, $\boldsymbol{\alpha} \odot \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$. In other words, for all i, we know that at least one of the constraints $\alpha_i \geq 0$ or $h^{(i)}(\boldsymbol{x}) \leq 0$ must be active at the solution. To gain some intuition for this idea, we can say that either the solution is on the boundary imposed by the inequality and we must use its KKT multiplier to influence the solution to $\boldsymbol{x}$, or the inequality has no influence on the solution and we represent this by zeroing out its KKT multiplier.

A simple set of properties describe the optimal points of constrained optimization problems. These properties are called the Karush-Kuhn-Tucker (KKT) conditions (Karush, 1939; Kuhn and Tucker, 1951). They are necessary conditions, but not always sufficient conditions, for a point to be optimal. The conditions are:

- The gradient of the generalized Lagrangian is zero.
- All constraints on both $\boldsymbol{x}$ and the KKT multipliers are satisfied.
- The inequality constraints exhibit "complementary slackness": $\boldsymbol{\alpha} \odot \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$.

For more information about the KKT approach, see Nocedal and Wright (2006).

4.5 Example: Linear Least Squares

Suppose we want to find the value of $\boldsymbol{x}$ that minimizes

$$f(\boldsymbol{x}) = \frac{1}{2}||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2. \tag{4.21}$$

Specialized linear algebra algorithms can solve this problem efficiently; however, we can also explore how to solve it using gradient-based optimization as a simple example of how these techniques work.

First, we need to obtain the gradient:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b}. \tag{4.22}$$

We can then follow this gradient downhill, taking small steps. See algorithm 4.1 for details.

Algorithm 4.1 An algorithm to minimize $f(\boldsymbol{x}) = \frac{1}{2}||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2$ with respect to $\boldsymbol{x}$ using gradient descent, starting from an arbitrary value of $\boldsymbol{x}$.

```
Set the step size ($\epsilon$) and tolerance ($\delta$) to small, positive numbers.
while $||\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b}||_2 > \delta$ do
  $\boldsymbol{x} \leftarrow \boldsymbol{x} - \epsilon\left(\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b}\right)$
end while
```

One can also solve this problem using Newton's method. In this case, because the true function is quadratic, the quadratic approximation employed by Newton's method is exact, and the algorithm converges to the global minimum in a single step.

Now suppose we wish to minimize the same function, but subject to the constraint $\boldsymbol{x}^\top\boldsymbol{x} \le 1$. To do so, we introduce the Lagrangian

$$L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda\left(\boldsymbol{x}^\top\boldsymbol{x} - 1\right). \tag{4.23}$$

We can now solve the problem

$$\min_{\boldsymbol{x}} \max_{\lambda, \lambda \ge 0} L(\boldsymbol{x}, \lambda). \tag{4.24}$$

The smallest-norm solution to the unconstrained least-squares problem may be found using the Moore-Penrose pseudoinverse: $\boldsymbol{x} = \boldsymbol{A}^+\boldsymbol{b}$. If this point is feasible, then it is the solution to the constrained problem. Otherwise, we must find a solution where the constraint is active. By differentiating the Lagrangian with respect to $\boldsymbol{x}$, we obtain the equation

$$\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top \boldsymbol{b} + 2\lambda \boldsymbol{x} = 0. \tag{4.25}$$

This tells us that the solution will take the form

$$\boldsymbol{x} = (\boldsymbol{A}^\top \boldsymbol{A} + 2\lambda \boldsymbol{I})^{-1} \boldsymbol{A}^\top \boldsymbol{b}. \tag{4.26}$$

The magnitude of λ must be chosen such that the result obeys the constraint. We can find this value by performing gradient ascent on λ. To do so, observe

$$\frac{\partial}{\partial \lambda} L(\boldsymbol{x}, \lambda) = \boldsymbol{x}^\top \boldsymbol{x} - 1. \tag{4.27}$$

When the norm of $\boldsymbol{x}$ exceeds 1, this derivative is positive, so to follow the derivative uphill and increase the Lagrangian with respect to λ, we increase λ. Because the coefficient on the $\boldsymbol{x}^\top \boldsymbol{x}$ penalty has increased, solving the linear equation for $\boldsymbol{x}$ will now yield a solution with a smaller norm. The process of solving the linear equation and adjusting λ continues until $\boldsymbol{x}$ has the correct norm and the derivative on λ is 0.

This concludes the mathematical preliminaries that we use to develop machine learning algorithms. We are now ready to build and analyze some full-fledged learning systems.

descent. We describe how to combine various algorithm components, such as an optimization algorithm, a cost function, a model, and a dataset, to build a machine learning algorithm. Finally, in section 5.11, we describe some of the factors that have limited the ability of traditional machine learning to generalize. These challenges have motivated the development of deep learning algorithms that overcome these obstacles.

5.1 Learning Algorithms

A machine learning algorithm is an algorithm that is able to learn from data. But what do we mean by learning? Mitchell (1997) provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." One can imagine a wide variety of experiences E, tasks T, and performance measures P, and we do not attempt in this book to formally define what may be used for each of these entities. Instead, in the following sections, we provide intuitive descriptions and examples of the different kinds of tasks, performance measures, and experiences that can be used to construct machine learning algorithms.

5.1.1 The Task, T

Machine learning enables us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientific and philosophical point of view, machine learning is interesting because developing our understanding of it entails developing our understanding of the principles that underlie intelligence.

In this relatively formal definition of the word "task," the process of learning itself is not the task. Learning is our means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually.

Machine learning tasks are usually described in terms of how the machine learning system should process an **example**. An example is a collection of **features** that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $\boldsymbol{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature. For example, the features of an image are usually the values of the pixels in the image.