Deep Learning

Yoshua Bengio
Ian Goodfellow
Aaron Courville

Contents

Acknowledgments **vii**

Notation **ix**

1 Introduction **1**
1.1 Who Should Read This Book? 8
1.2 Historical Trends in Deep Learning 11

I Applied Math and Machine Learning Basics **25**

2 Linear Algebra **27**
2.1 Scalars, Vectors, Matrices and Tensors 27
2.2 Multiplying Matrices and Vectors 30
2.3 Identity and Inverse Matrices 31
2.4 Linear Dependence, Span, and Rank 32
2.5 Norms 34
2.6 Special Kinds of Matrices and Vectors 35
2.7 Eigendecomposition 37
2.8 Singular Value Decomposition 39
2.9 The Moore-Penrose Pseudoinverse 40
2.10 The Trace Operator 41
2.11 Determinant 42
2.12 Example: Principal Components Analysis 42

3 Probability and Information Theory **46**
3.1 Why Probability? 46
3.2 Random Variables 48
3.3 Probability Distributions 49
3.4 Marginal Probability 51
3.5 Conditional Probability 51

3.6 The Chain Rule of Conditional Probabilities 52
3.7 Independence and Conditional Independence 52
3.8 Expectation, Variance, and Covariance 53
3.9 Information Theory 54
3.10 Common Probability Distributions 57
3.11 Useful Properties of Common Functions 62
3.12 Bayes' Rule 64
3.13 Technical Details of Continuous Variables 64
3.14 Structured Probabilistic Models 65
3.15 Example: Naive Bayes 68

4 Numerical Computation **74**
4.1 Overflow and Underflow 74
4.2 Poor Conditioning 75
4.3 Gradient-Based Optimization 76
4.4 Constrained Optimization 85
4.5 Example: Linear Least Squares 87

5 Machine Learning Basics **89**
5.1 Learning Algorithms 89
5.2 Example: Linear Regression 97
5.3 Generalization, Capacity, Overfitting and Underfitting 99
5.4 The No Free Lunch Theorem 104
5.5 Regularization 106
5.6 Hyperparameters, Validation Sets and Cross-Validation 108
5.7 Estimators, Bias, and Variance 110
5.8 Maximum Likelihood Estimation 118
5.9 Bayesian Statistics and Prior Probability Distributions 121
5.10 Supervised Learning 128
5.11 Unsupervised Learning 131
5.12 Weakly Supervised Learning 134
5.13 The Curse of Dimensionality and Statistical Limitations of Local Generalization 135

II Modern Practical Deep Networks **147**

6 Feedforward Deep Networks **149**
6.1 From Fixed Features to Learned Features 149
6.2 Formalizing and Generalizing Neural Networks 152
6.3 Parametrizing a Learned Predictor 154

6.4 Flow Graphs and Back-Propagation . . . 167
6.5 Universal Approximation Properties and Depth . . . 180
6.6 Feature / Representation Learning . . . 184
6.7 Piecewise Linear Hidden Units . . . 186
6.8 Historical Notes . . . 188

7 Regularization **190**
7.1 Regularization from a Bayesian Perspective . . . 191
7.2 Classical Regularization: Parameter Norm Penalty . . . 193
7.3 Classical Regularization as Constrained Optimization . . . 200
7.4 Regularization and Under-Constrained Problems . . . 201
7.5 Dataset Augmentation . . . 203
7.6 Classical Regularization as Noise Robustness . . . 204
7.7 Early Stopping as a Form of Regularization . . . 208
7.8 Parameter Tying and Parameter Sharing . . . 215
7.9 Sparse Representations . . . 215
7.10 Bagging and Other Ensemble Methods . . . 215
7.11 Dropout . . . 218
7.12 Multi-Task Learning . . . 222
7.13 Adversarial Training . . . 223

8 Optimization for Training Deep Models **226**
8.1 Optimization for Model Training . . . 226
8.2 Challenges in Optimization . . . 229
8.3 Optimization Algorithms . . . 236
8.4 Approximate Natural Gradient and Second-Order Methods . . . 241
8.5 Conjugate Gradients . . . 241
8.6 BFGS . . . 241
8.7 Hints, Global Optimization and Curriculum Learning . . . 243

9 Convolutional Networks **248**
9.1 The Convolution Operation . . . 248
9.2 Motivation . . . 252
9.3 Pooling . . . 256
9.4 Convolution and Pooling as an Infinitely Strong Prior . . . 261
9.5 Variants of the Basic Convolution Function . . . 262
9.6 Structured Outputs . . . 269
9.7 Convolutional Modules . . . 269
9.8 Data Types . . . 269
9.9 Efficient Convolution Algorithms . . . 271
9.10 Random or Unsupervised Features . . . 271

9.11 The Neuroscientific Basis for Convolutional Networks 273
9.12 Convolutional Networks and the History of Deep Learning 280

10 Sequence Modeling: Recurrent and Recursive Nets 281
10.1 Unfolding Flow Graphs and Sharing Parameters 282
10.2 Recurrent Neural Networks 284
10.3 Bidirectional RNNs 295
10.4 Deep Recurrent Networks 296
10.5 Recursive Neural Networks 299
10.6 Auto-Regressive Networks 299
10.7 Facing the Challenge of Long-Term Dependencies 305
10.8 Handling Temporal Dependencies with N-Grams, HMMs, CRFs and Other Graphical Models 317
10.9 Combining Neural Networks and Search 328

11 Practical methodology 333
11.1 Basic Machine Learning Methodology 333
11.2 Manual Hyperparameter Tuning 334
11.3 Hyper-parameter Optimization Algorithms 334
11.4 Debugging Strategies 336

12 Applications 339
12.1 Large Scale Deep Learning 339
12.2 Computer Vision 345
12.3 Speech Recognition 352
12.4 Natural Language Processing and Neural Language Models . . . 353
12.5 Structured Outputs 369
12.6 Other Applications 369

III Deep Learning Research 370

13 Structured Probabilistic Models for Deep Learning 372
13.1 The Challenge of Unstructured Modeling 373
13.2 Using Graphs to Describe Model Structure 377
13.3 Advantages of Structured Modeling 391
13.4 Learning About Dependencies 392
13.5 Inference and Approximate Inference Over Latent Variables . . . 394
13.6 The Deep Learning Approach to Structured Probabilistic Models 395

14 Monte Carlo Methods 400
14.1 Markov Chain Monte Carlo Methods 400

14.2 The Difficulty of Mixing Between Well-Separated Modes 402

15 Linear Factor Models and Auto-Encoders 404
15.1 Regularized Auto-Encoders 405
15.2 Denoising Auto-encoders 408
15.3 Representational Power, Layer Size and Depth 410
15.4 Reconstruction Distribution 411
15.5 Linear Factor Models 412
15.6 Probabilistic PCA and Factor Analysis 413
15.7 Reconstruction Error as Log-Likelihood 417
15.8 Sparse Representations 418
15.9 Denoising Auto-Encoders 423
15.10 Contractive Auto-Encoders 428

16 Representation Learning 431
16.1 Greedy Layerwise Unsupervised Pre-Training 432
16.2 Transfer Learning and Domain Adaptation 439
16.3 Semi-Supervised Learning 446
16.4 Semi-Supervised Learning and Disentangling Underlying Causal Factors 447
16.5 Assumption of Underlying Factors and Distributed Representation 449
16.6 Exponential Gain in Representational Efficiency from Distributed Representations 453
16.7 Exponential Gain in Representational Efficiency from Depth . . . 455
16.8 Priors Regarding The Underlying Factors 457

17 The Manifold Perspective on Representation Learning 461
17.1 Manifold Interpretation of PCA and Linear Auto-Encoders . . . 469
17.2 Manifold Interpretation of Sparse Coding 472
17.3 The Entropy Bias from Maximum Likelihood 472
17.4 Manifold Learning via Regularized Auto-Encoders 473
17.5 Tangent Distance, Tangent-Prop, and Manifold Tangent Classifier 474

18 Confronting the Partition Function 478
18.1 The Log-Likelihood Gradient of Energy-Based Models 479
18.2 Stochastic Maximum Likelihood and Contrastive Divergence . . . 481
18.3 Pseudolikelihood 488
18.4 Score Matching and Ratio Matching 490
18.5 Denoising Score Matching 492
18.6 Noise-Contrastive Estimation 492
18.7 Estimating the Partition Function 494

19 Approximate inference **502**
19.1 Inference as Optimization 504
19.2 Expectation Maximization 505
19.3 MAP Inference: Sparse Coding as a Probabilistic Model 506
19.4 Variational Inference and Learning 507
19.5 Stochastic Inference 511
19.6 Learned Approximate Inference 511

20 Deep Generative Models **513**
20.1 Boltzmann Machines 513
20.2 Restricted Boltzmann Machines 516
20.3 Training Restricted Boltzmann Machines 519
20.4 Deep Belief Networks 523
20.5 Deep Boltzmann Machines 526
20.6 Boltzmann Machines for Real-Valued Data 537
20.7 Convolutional Boltzmann Machines 540
20.8 Other Boltzmann Machines 541
20.9 Directed Generative Nets 541
20.10 A Generative View of Autoencoders 543
20.11 Generative Stochastic Networks 549
20.12 Methodological Notes 551

Bibliography **555**

Index **593**

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: Hugo Larochelle, Guillaume Alain, Kyunghyun Cho, Çağlar Gülçehre, Razvan Pascanu, David Krueger 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: Julian Serban, Laurent Dinh, Guillaume Alain, Kelvin Xu, Ilya Sutskever, Vincent Vanhoucke, David Warde-Farley, Jurgen Van Gael, Dustin Webb, Johannes Roith, Ion Androutsopoulos, Pawel Chilinski, Halis Sak, Frédéric Francis, Jonathan Hunt, and Grigory Sapunov.

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

- Chapter 1, Introduction: Johannes Roith, Eric Morris, Samira Ebrahimi, Ozan Çağlayan, Martín Abadi, and Sebastien Bratieres.
- Chapter 2, Linear Algebra: Pierre Luc Carrier, Li Yao, Thomas Rohée, Colby Toland, Amjad Almahairi, Sergey Oreshkov, István Petrás, and Dennis Prangle.
- Chapter 3, Probability and Information Theory: Rasmus Antti, Stephan Gouws, Vincent Dumoulin, Artem Oboturov, Li Yao, John Philip Anderson, and Kai Arulkumaran.
- Chapter 4, Numerical Computation: Meire Fortunato, and Tran Lam An.
- Chapter 5, Machine Learning Basics: Dzmitry Bahdanau, and Meire Fortunato.
- Chapter 8, Optimization for Training Deep Models: Marcel Ackermann.
- Chapter 9, Convolutional Networks: Mehdi Mirza, Çağlar Gülçehre, and Martín Arjovsky.

- Chapter 10, Sequence Modeling: Recurrent and Recursive Nets: Dmitriy Serdyuk, and Dongyu Shi.
- Chapter 18, Confronting the Partition Function: Sam Bowman, and Ozan Çağlayan.
- Bibliography, Leslie N. Smith.

We also want to thank those who allowed us to reproduce images, figures or data from their publications: David Warde-Farley, Matthew D. Zeiler, Rob Fergus, Chris Olah, Jason Yosinski, Nicolas Chapados, and James Bergstra. We indicate their contributions in the figure captions throughout the text.

Finally, we would like to thank Google for allowing Ian Goodfellow to work on the book as his 20% project while working at Google. In particular, we would like to thank Ian's former manager, Greg Corrado, and his subsequent manager, Samy Bengio, for their support of this effort.

Notation

This section provides a concise reference describing the notation used throughout this book. If you are unfamiliar with any of these mathematical concepts, this notation reference may seem intimidating. However, do not despair, we describe most of these ideas in chapters 1-3.

Numbers and Arrays

a	A scalar (integer or real) value with the name "a"
$\boldsymbol{a}$	A vector with the name "a"
$\boldsymbol{A}$	A matrix with the name "A"
A	A tensor with the name "A"
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}_i$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i.
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with entries given by $\boldsymbol{a}$
a	A scalar random variable with the name "a"
$\mathbf{a}$	A vector-valued random variable with the name "a"
$\mathbf{A}$	A matrix-valued random variable with the name "A"

Sets and Graphs

$\mathbb{A}$	A set with the name "A"
$\mathbb{R}$	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 elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph with the name "G"
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	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 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 $\boldsymbol{A}$
$\mathbf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor $\mathbf{A}$
$\mathbf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$
$\boldsymbol{x}^{(t)}$	The t-th example (input) from a dataset
$y^{(t)}$ or $\boldsymbol{y}^{(t)}$	The target associated with $\boldsymbol{x}^{(t)}$ for supervised learning
$\boldsymbol{X}$	The matrix of input examples, with one row per example $\boldsymbol{x}^{(t)}$.

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 $\boldsymbol{B}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to x
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to x
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of a function $f : \mathbb{R}^n \to \mathbb{R}^m$
$\boldsymbol{H}(f)(\boldsymbol{x})$	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

$\mathrm{a} \perp \mathrm{b}$	The random variables a and b are independent.
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	They are are conditionally independent given c.
$\mathbb{E}_{x \sim P}[f(x)]$ 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 $P(\mathrm{x}, \mathrm{y})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \| Q)$	Kullback-Leibler divergence of P and Q

Functions

$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parameterized by $\boldsymbol{\theta}$
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $1/(1 + \exp(-x))$
$\zeta(x)$	Softplus, $\log(1 + \exp(x))$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$
$\mathbf{1}_{condition}$	is 1 if the condition is true, 0 otherwise.

Sometimes we write $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathbf{X})$, when f is a function of a scalar rather than a vector, matrix, or tensor. In this case, we mean to apply f to the array element-wise. For example, if $\mathbf{C} = \sigma(\mathbf{X})$, then $C_{i,j,k} = \sigma(X_{i,j,k})$ for all valid values of i, j, and k.

Chapter 1

Introduction

Inventors have long dreamed of creating machines that think. Ancient Greek myths tell of intelligent objects, such as animated statues of human beings and tables that arrive full of food and drink when called.

When programmable computers were first conceived, people wondered whether they 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 to 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 in terms of its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all of the knowledge that the computer needs. The hierarchy of concepts allows 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 relevant concepts 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 are only recently 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, and 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 about statements in these formal languages automatically using logical inference rules. This is known as the *knowledge base* approach to artificial intelligence. None of these projects has lead 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 introduction of machine learning allowed computers to tackle problems involving knowledge of the real world and make decisions that appear subjective. A simple machine learning algorithm called *logistic regression* can determine whether to recommend cesarean delivery (Mor-Yosef *et al.*, 1990). A simple machine learning algorithm called *naive Bayes* can separate legitimate e-mail from spam e-mail.

The performance of these simple machine learning algorithms depends heavily on the *representation* of the data they are given. For example, when logistic regression is used to recommend cesarean delivery, the AI system does not examine the patient directly. Instead, the doctor tells the system several pieces of relevant information, such as the presence or absence of a uterine scar. Each piece of information included in the representation of the patient is known as a *feature*. Logistic regression learns how each of these features of the patient correlates with various outcomes. However, it cannot influence the way that the features are defined in any way. If logistic regression was given a 3-D MRI image of the patient, rather than the doctor's formalized report, it would not be able to make useful predictions. Individual voxels[1] in an MRI scan have negligible correlation with any complications that might occur during delivery.

This dependence on representations is a general phenomenon that appears throughout computer science and even daily life. In computer science, operations such as searching a collection of data can proceed exponentially faster if the collection is structured and indexed intelligently. People can easily perform arithmetic on Arabic numerals, but find arithmetic on Roman numerals much more time consuming. It is not surprising that the choice of representation has an enormous effect on the performance of machine learning algorithms. For a simple visual example, see Fig. 1.1.

Many artificial intelligence tasks can be solved by designing the right set of features to extract for that task, then providing these features to a simple machine learning algorithm. For example, a useful feature for speaker identification from sound is the pitch. The pitch can be formally specified—it is the lowest frequency major peak of the spectrogram. It is useful for speaker identification because it is determined by the size of the vocal tract, and therefore gives a strong clue as to whether the speaker is a man, woman, or child.

However, for many tasks, it is difficult to know what features should be extracted. For example, suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.

One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as *representation learning*. Learned representations of-

[1] A voxel is the value at a single point in a 3-D scan, much as a pixel as the value at a single point in an image.

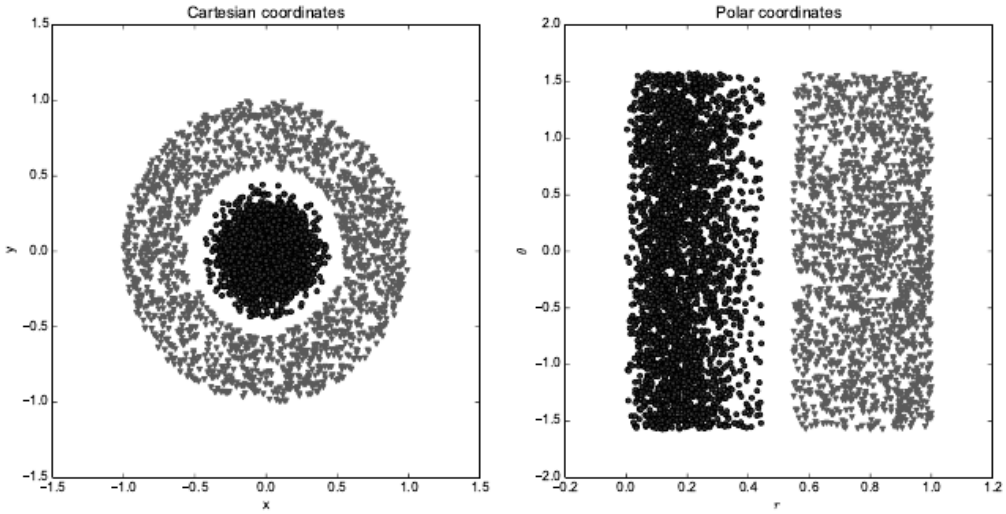


Figure 1.1: Example of different representations: suppose we want to separate two categories of data by drawing a line between them in a scatterplot. In the plot on the left, we represent some data using Cartesian coordinates, and the task is impossible. In the plot on the right, we represent the data with polar coordinates and the task becomes simple to solve with a vertical line. (Figure credit: David Warde-Farley)

ten result in much better performance than can be obtained with hand-designed representations. They also allow AI systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a simple task in minutes, or a complex task in hours to months. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.

The quintessential example of a representation learning algorithm is the *autoencoder*. An autoencoder is the combination of an *encoder* function that converts the input data into a different representation, and a *decoder* function that converts the new representation back into the original format. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties.

When designing features or algorithms for learning features, our goal is usually to separate the *factors of variation* that explain the observed data. In this context, we use the word "factors" simply to refer to separate sources of influence; the factors are usually not combined by multiplication. Such factors are often not quantities that are directly observed but they may exist either as unobserved objects or forces in the physical world that affect observable quantities, or they are 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 and sex, their accent, and the words that 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 only be identified 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 allows the computer to build complex concepts out of simpler concepts. Fig. 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 *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 it allows the computer to learn a multi-step 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. Being able to execute instructions sequentially offers great power because later instructions can refer back to the results of earlier instructions. According to this view of deep learning, not all of the information in a layer's representation of the input necessarily encodes factors of variation that explain the input. The representation is also used to store state information that helps to execute a program that can make sense of the input. This state

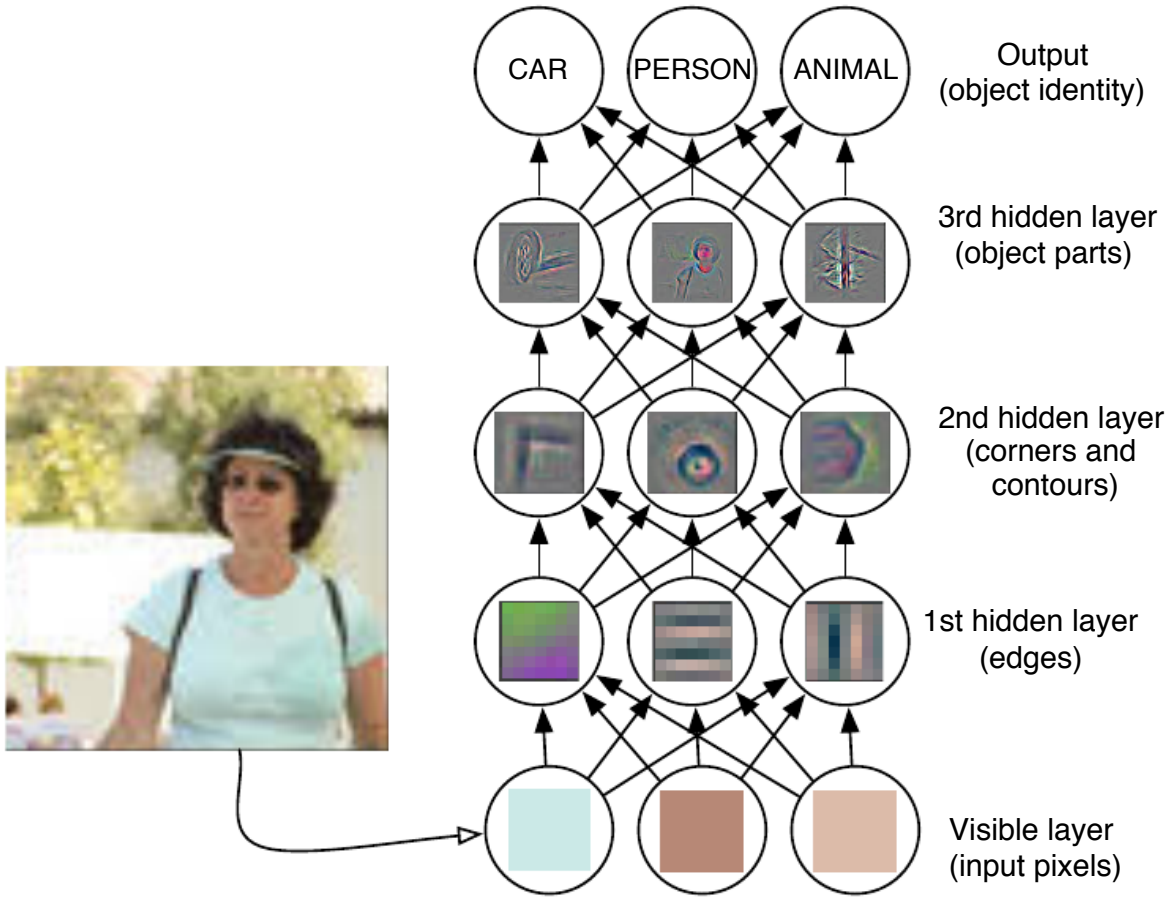


Figure 1.2: Illustration of a deep learning model. It is difficult for a computer to understand the meaning of raw sensory input data, such as this image represented as a collection of pixel values. The function mapping from a set of pixels to an object identity is very complicated. Learning or evaluating 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).

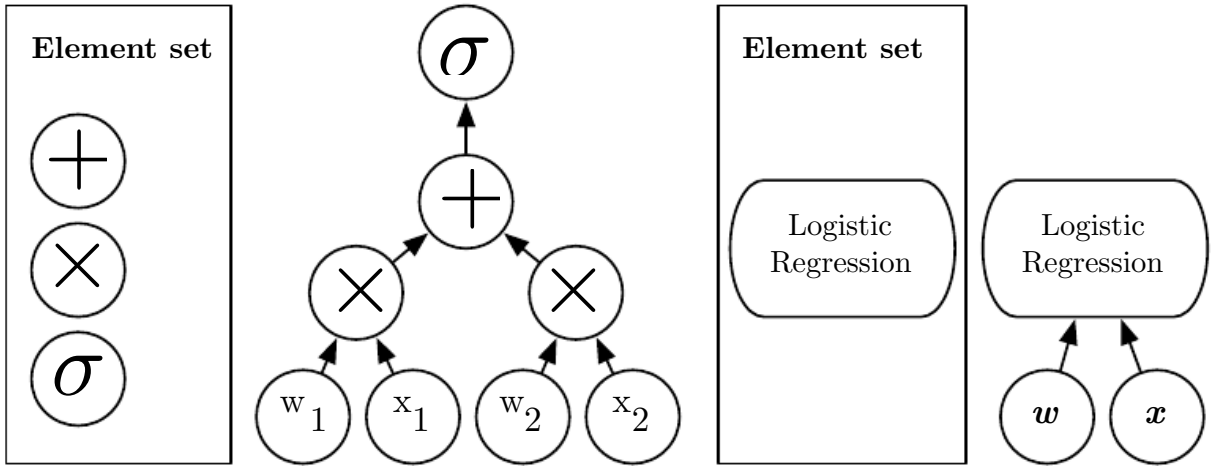


Figure 1.3: Illustration of computational flow graphs mapping an input to an output where each node performs an operation. Depth is the length of the longest path from input to output but depends on the definition of what constitutes a possible computational step. The computation depicted in these graphs 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.

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 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 flow chart with different depths depending on which functions we allow to be used as individual steps in the flow chart. Fig. 1.3 illustrates how this choice of language can give two different measurements for the same architecture.

Another approach, used by deep probabilistic models, examples 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 of the flow-chart 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 only see one eye. After detecting that a face is present, it can

then infer that a second eye is probably present as well. In this case, the graph of concepts only includes 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 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 safely be regarded as the study of models that either involve a greater amount of composition of 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 allows computer systems to improve with experience and data. According to the authors of this book, 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 learning to represent the world as a nested hierarchy of concepts and representations, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones. Fig. 1.4 illustrates the relationship between these different AI disciplines. Fig. 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 main 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. Software engineers working in a wide variety of industries are likely to find deep learning to be useful, as it has already proven successful in many areas 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 in order to best accommodate a variety of readers. Part 1 introduces basic mathematical tools and machine

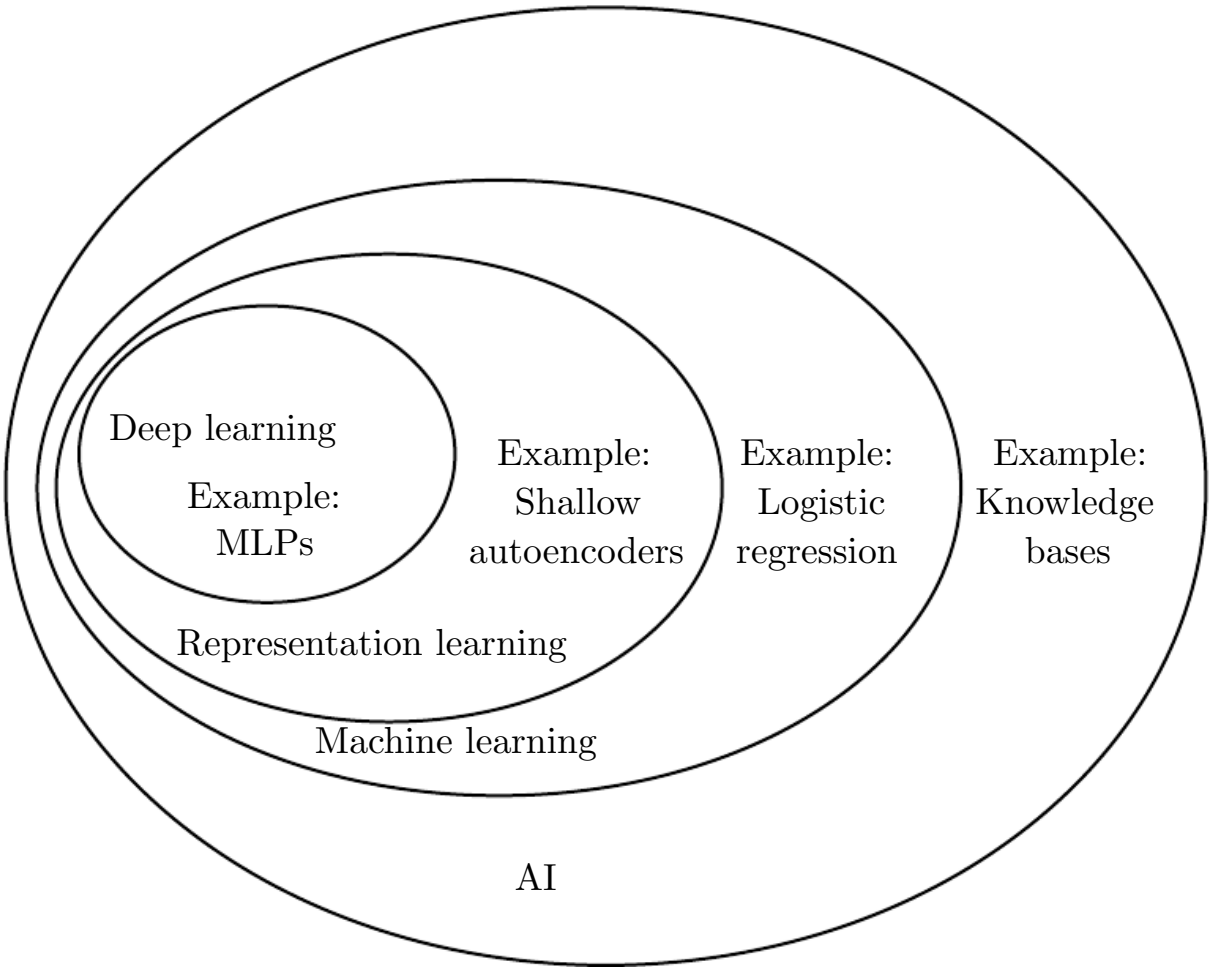


Figure 1.4: A Venn diagram showing how deep learning is a kind of representation learning, which is in turn a kind of machine learning, which is used for many but not all approaches to AI. Each section of the Venn diagram includes an example of an AI technology.

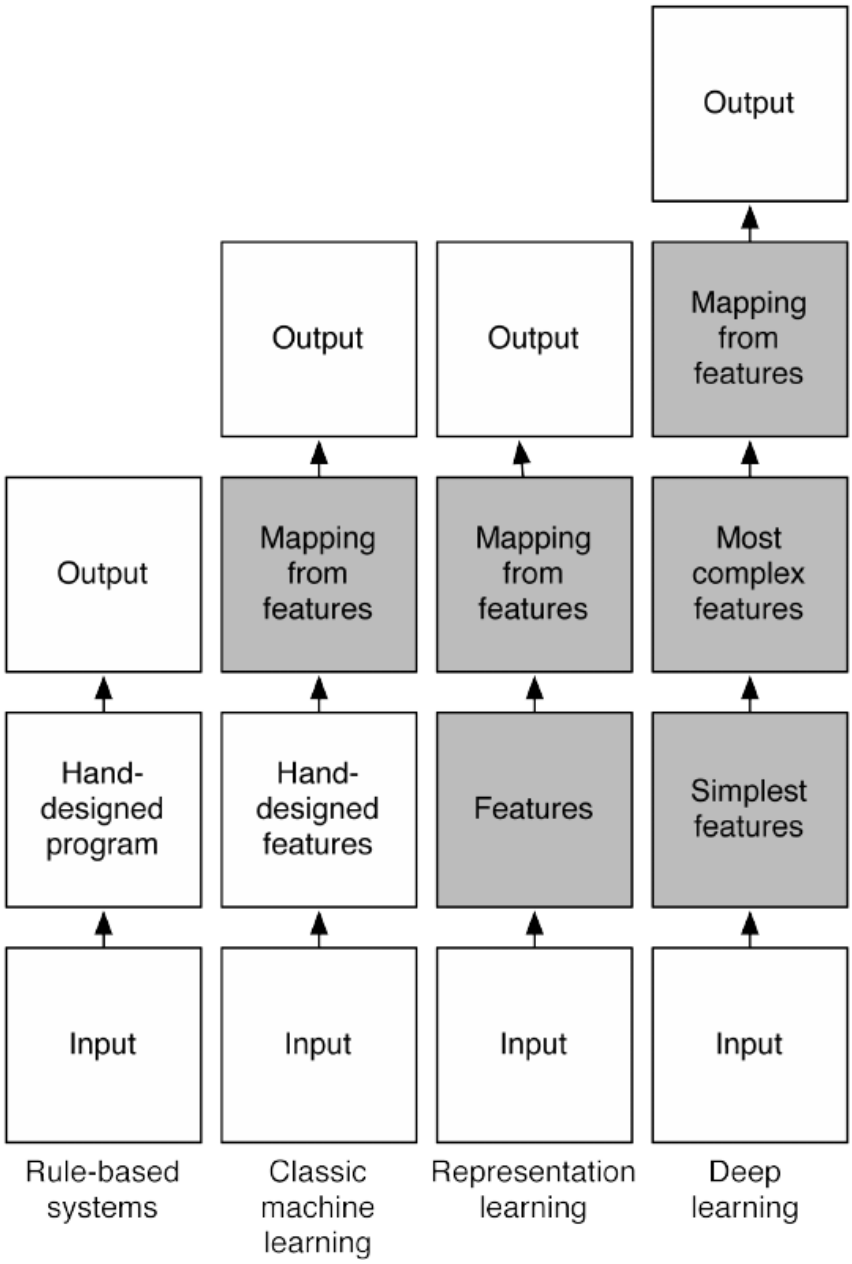


Figure 1.5: Flow-charts showing how the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data.

learning concepts. Part 2 describes the most established deep learning algorithms that are essentially solved technologies. Part 3 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 1, for example, while readers who just want to implement a working system need not read beyond part 2.

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 hardware and software infrastructure 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 has a long and rich history. 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. While the term "deep learning" is relatively new, the field dates back to the 1950s. 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 pedagogical textbook. However, some basic context is useful for understanding deep learning. Broadly speaking, there have been three waves of development of deep learning: 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. See Figure 1.6 for a basic timeline.

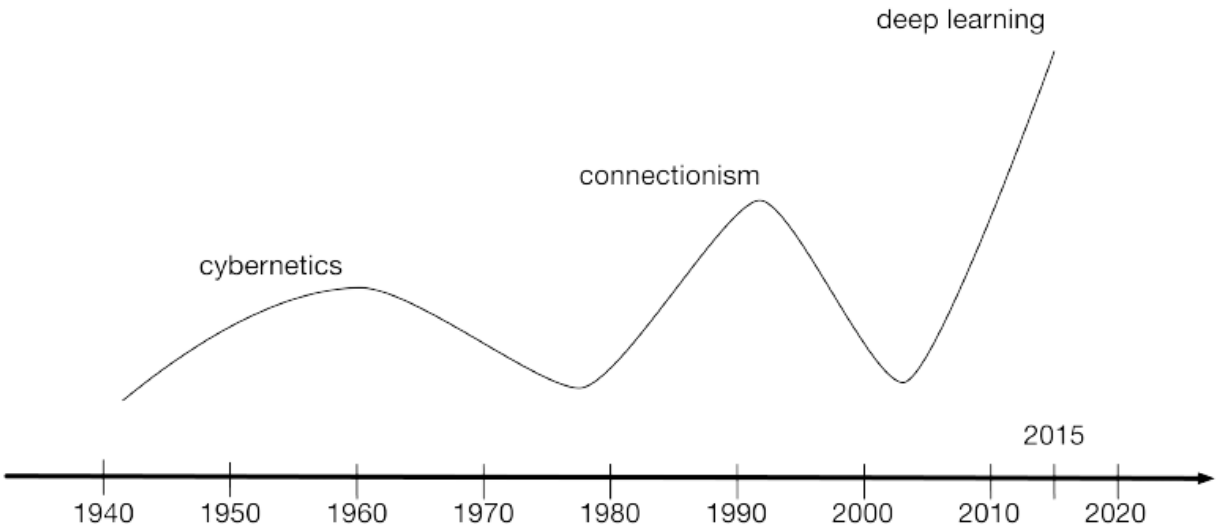


Figure 1.6: The three historical waves of artificial neural nets research, starting with cybernetics in the 1940-1960's, with the perceptron (Rosenblatt, 1958) to train a single neuron, then the connectionist approach of the 1980-1995 period, with back-propagation (Rumelhart *et al.*, 1986a) to train a neural network with one or two hidden layers, and the current wave, deep learning, started around 2006 (Hinton *et al.*, 2006; Bengio *et al.*, 2007a; Ranzato *et al.*, 2007a), which allows us to train very deep networks.

Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, i.e. 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). 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* (see Fig. 1.6).

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 defining the categories given examples of inputs from each category. The Adaptive Linear Element (ADALINE), which dates from the 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 is the first dip in the popularity of neural networks in our broad timeline (Fig. 1.6).

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 of 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.*, 1998a), as we will see in Chapter 9.11. Most neural networks today are based on a model neuron called the *rectified linear unit*. These units were developed from a variety of viewpoints, with (Nair and Hinton, 2010b) and Glorot *et al.* (2011a) citing neuroscience as an influence, and Jarrett *et al.* (2009a) 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 found a machine learning value or interpretation. 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 influence, 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.*, 1986d). 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, 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 and 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*. 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 that activates for each of the nine possible combinations: red truck, red car, red bird, green truck, and so on. This requires nine different neurons, and each neuron must independently learn the concept of color and object identity. One way to improve on this situation is to use a distributed representation, with three neurons describing the color and three neurons describing the object identity. This requires only six neurons total instead of nine, and the neuron describing redness is able to learn about redness from images of cars, trucks, and birds, not only from images of one specific category of objects. The concept of distributed representation is central to this book, and will be described in greater detail in Chapter 16.

Another major accomplishment of the connectionist movement was the successful use of back-propagation to train deep neural networks with internal representations and the popularization of the back-propagation algorithm (Rumelhart *et al.*, 1986a; LeCun, 1987). This algorithm has waxed and waned in popularity but as of this writing is currently the dominant approach to training deep models.

The second wave of neural networks research lasted until the mid-1990s. At that point, the popularity of neural networks declined again. This was in part due to a negative reaction to the failure of neural networks (and AI research in general) to fulfill excessive promises made by a variety of people seeking investment in neural network-based ventures, but also due to improvements in other fields of machine learning: kernel machines (Boser *et al.*, 1992; Cortes and Vapnik, 1995; Schölkopf *et al.*, 1999) and graphical models (Jordan, 1998).

Kernel machines enjoy many nice theoretical guarantees. In particular, training a kernel machine is a *convex optimization problem* (this will be explained in more detail in Chapter 4) which means that the training process can be guaranteed to find the optimal model efficiently. This made kernel machines very amenable to software implementations that "just work" without much need for the human operator to understand the underlying ideas. Soon, most machine learning applications consisted of manually designing good features to provide to a kernel machine for each different application area.

During this time, neural networks continued to obtain impressive performance on some tasks (LeCun *et al.*, 1998b; Bengio *et al.*, 2001a). The Canadian Institute for Advanced Research (CIFAR) helped to keep neural networks research alive via its Neural Computation and Adaptive Perception research initiative. This program united machine research groups led by Geoffrey Hinton at University of Toronto, Yoshua Bengio at University of Montreal, and Yann LeCun at New York University. It had a multi-disciplinary nature that also included neuroscientists and experts in human and computer vision.

At this point in time, deep networks were generally believed to be very difficult to train. We now know that algorithms that have existed since the 1980s work quite well, but this was not apparent circa 2006. The issue is perhaps simply that these algorithms were too computationally costly to allow much experimentation with the hardware available at the time.

The third wave of neural networks research began with a breakthrough in 2006. Geoffrey Hinton showed that a kind of neural network called a deep belief network could be efficiently trained using a strategy called greedy layer-wise pretraining (Hinton *et al.*, 2006), which will be described in more detail in Chapter 16.1. The other CIFAR-affiliated research groups quickly showed that the same strategy could be used to train many other kinds of deep networks (Bengio *et al.*, 2007a; Ranzato *et al.*, 2007a) and systematically helped to improve gen-

eralization on test examples. This wave of neural networks research popularized the use of the term *deep learning* to emphasize that researchers were now able to train deeper neural networks than had been possible before, and to emphasize the theoretical importance of depth (Bengio and LeCun, 2007a; Delalleau and Bengio, 2011; Pascanu *et al.*, 2014a; Montufar *et al.*, 2014). Deep neural networks displaced kernel machines with manually designed features for several important application areas during this time—in part because the time and memory cost of training a kernel machine is quadratic in the size of the dataset, and datasets grew to be large enough for this cost to outweigh the benefits of convex optimization. This third wave of popularity of neural networks continues to the time of this writing, though the focus of deep learning research has changed dramatically within the time of this wave. The third wave began with a focus on new unsupervised learning techniques and the ability of deep models to generalize well from small datasets, but today there is more interest in much older supervised learning algorithms and the ability of deep models to leverage large labeled datasets.

1.2.2 Increasing Dataset Sizes

One may wonder why deep learning has only recently become recognized as a crucial technology if it has existed since the 1950s. Deep learning has been successfully used in commercial applications since the 1990s, but was often regarded as being more of an art than a technology and something that only an expert could use, until recently. It is true that some skill is required to get good performance from a deep learning algorithm. Fortunately, the amount of skill required reduces as the amount of training data increases. The learning algorithms reaching human performance on complex tasks today are nearly identical to the learning algorithms that struggled to solve toy problems in the 1980s, though the models we train with these algorithms have undergone changes that simplify the training of very deep architectures. The most important new development is that today we can provide these algorithms with the resources they need to succeed. Fig. 1.7 shows how the size of benchmark datasets has increased 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 of data—has been considerably lightened. As of 2015, 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 Fig. 1.8, our machine learning models have had a number of connections per neuron that was 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 Fig. 1.9. 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 allow 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, 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.

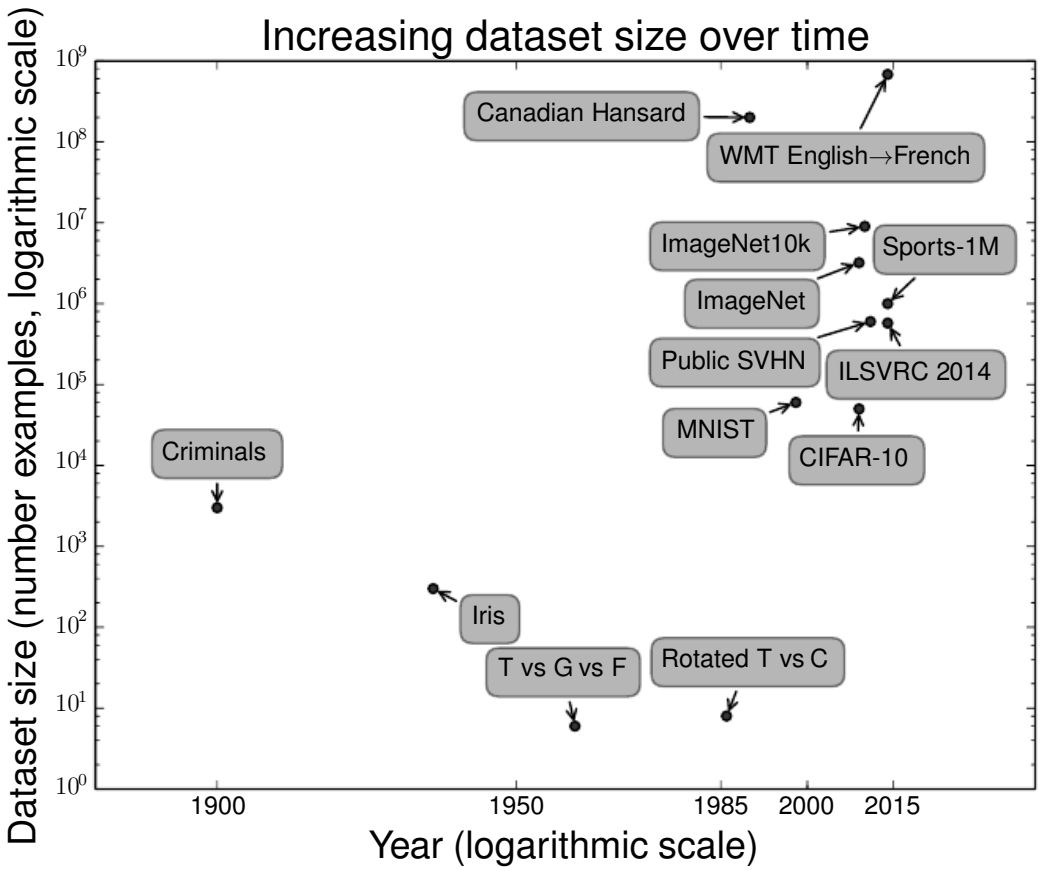


Figure 1.7: Dataset sizes have increased greatly over time. In the early 1900s, statisticians studied datasets using hundreds or thousands of manually compiled measurements (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936). In the 1950s through 1980s, the pioneers of biologically-inspired machine 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 in nature and began to leverage larger datasets containing tens of thousands of examples such as the MNIST dataset 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 dataset (Schwenk, 2014) are typically far ahead of other dataset sizes.

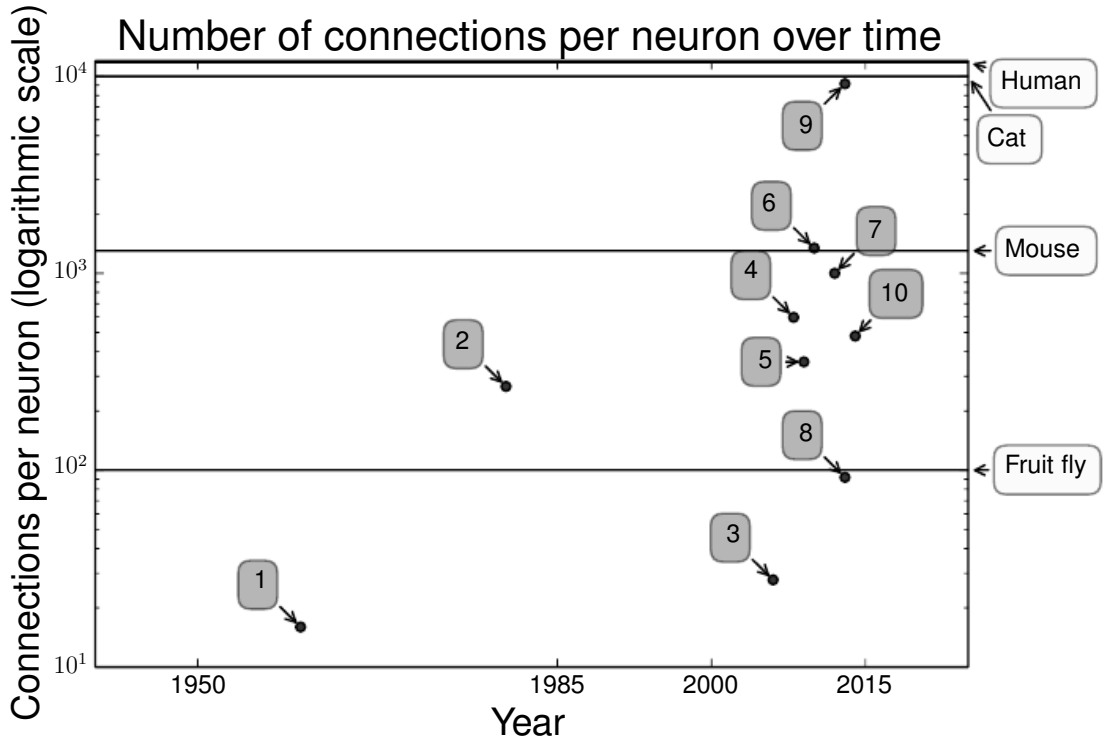


Figure 1.8: Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, 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. The sparse connectivity of biological neural networks means that our artificial networks are able to match the performance of biological neural networks despite limited hardware. Modern neural networks are much smaller than the brains of any vertebrate animal, but we typically train each network to perform just one task, while an animal's brain has different areas devoted to different tasks. 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 machines (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett *et al.*, 2009b)
6. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
7. Distributed autoencoder (Le *et al.*, 2012)
8. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012a)
9. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

1.2.4 Increasing Accuracy, Application Complexity, and Real-World Impact

Since the 1980s, deep learning has consistently improved in its ability to provide accurate recognition or 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(Krizhevsky *et al.*, 2012b). Similarly, the earliest networks could only recognize 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 Competition held each year. A dramatic moment in the meteoric rise of deep learning came when a convolutional network won this challenge for the first time and by a wide margin, bringing down the state-of-the-art error rate from 26.1% to 15.3% (Krizhevsky *et al.*, 2012b). Since then, these competitions are consistently won by deep convolutional nets, and as of this writing, advances in deep learning had brought the latest error rate in this contest down to 6.5% as shown in Fig. 1.10, using even deeper networks (Szegedy *et al.*, 2014a). Outside the framework of the contest, this error rate has now dropped to below 5% (Ioffe and Szegedy, 2015; Wu *et al.*, 2015).

Deep learning has also had a dramatic impact on speech recognition. After improving throughout the 1990s, the error rates for speech recognition stagnated starting in about 2000. The introduction of deep learning (Dahl *et al.*, 2010; Deng *et al.*, 2010b; Seide *et al.*, 2011; Hinton *et al.*, 2012a) to speech recognition resulted in a sudden drop of error rates by up to half! We will explore this history in more detail in Chapter 12.3.1.

Deep networks have also had spectacular successes for pedestrian detection and image segmentation (Sermanet *et al.*, 2013; Farabet *et al.*, 2013a; Couprie *et al.*, 2013) and yielded superhuman performance in traffic sign classification (Ciresan *et al.*, 2012).

At the same time that the scale and accuracy of deep networks has increased, so has the complexity of the tasks that they can solve. Goodfellow *et al.* (2014d) showed that neural networks could learn to output an entire sequence of characters transcribed from an image, rather than just identifying a single object. Previously, it was widely believed that this kind of learning required labeling of the individual elements of the sequence (Gülçehre and Bengio, 2013). Since this time, a neural network designed to model sequences, the Long Short-Term Memory or LSTM

(Hochreiter and Schmidhuber, 1997), has enjoyed an explosion in popularity. LSTMs and related models are now used to model relationships between *sequences* and other *sequences* rather than just fixed inputs. This sequence-to-sequence learning seems to be on the cusp of revolutionizing another application: machine translation (Sutskever *et al.*, 2014a; Bahdanau *et al.*, 2014).

This trend of increasing complexity has been pushed to its logical conclusion with the introduction of the Neural Turing Machine (Graves *et al.*, 2014), a neural network that can learn entire programs. This neural network has been shown to be able to learn how to sort lists of numbers given examples of scrambled and sorted sequences. This self-programming technology is in its infancy, but in the future could in principle be applied to nearly any task.

Many of these applications of deep learning are highly profitable, given enough data to apply deep learning to. Deep learning is now used by many top technology companies including Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA and NEC.

Deep learning has also made contributions back to other sciences. Modern convolutional networks for object recognition provide a model of visual processing that neuroscientists can study (DiCarlo, 2013). Deep learning also provides useful tools for processing massive amounts of data and making useful predictions in scientific fields. It has been successfully used to predict how molecules will interact in order to help pharmaceutical companies design new drugs (Dahl *et al.*, 2014), to search for subatomic particles (Baldi *et al.*, 2014), and to automatically parse microscope 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, it has seen tremendous growth in its popularity and usefulness, due in large part to 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 bring it to new frontiers.

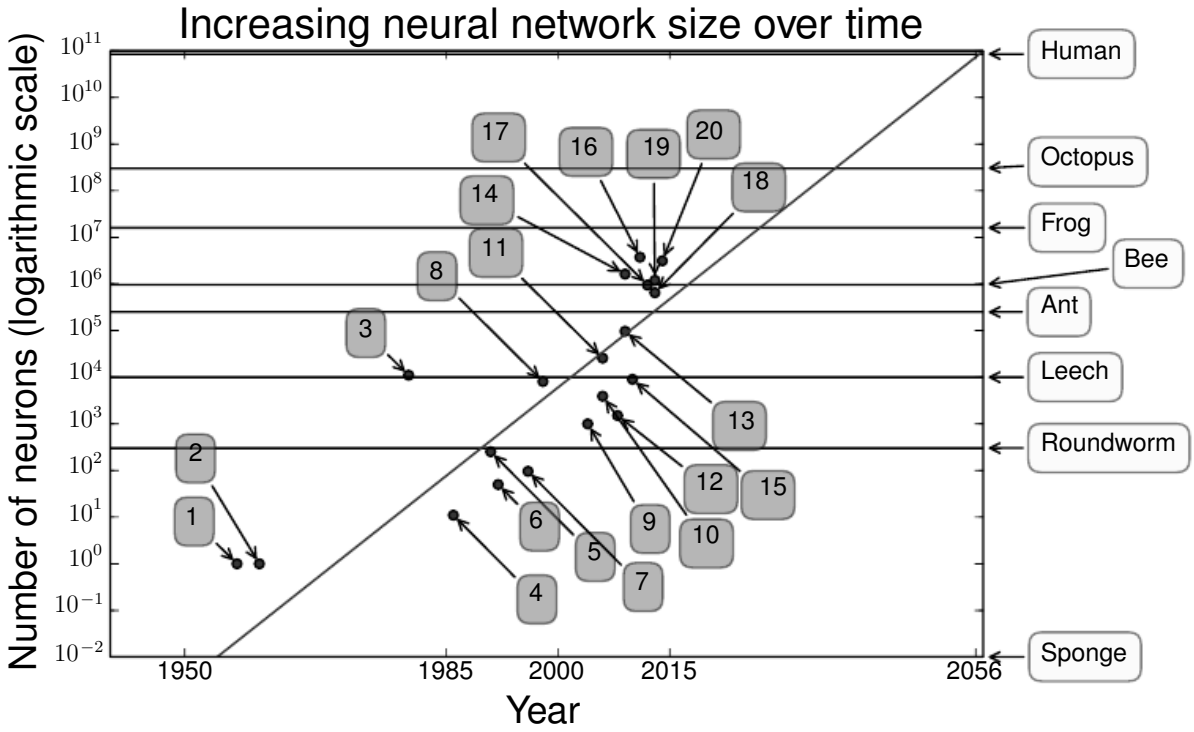


Figure 1.9: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive Linear Element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early backpropagation 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.*, 1998a)
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 machines (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009b)
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.*, 2012a)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

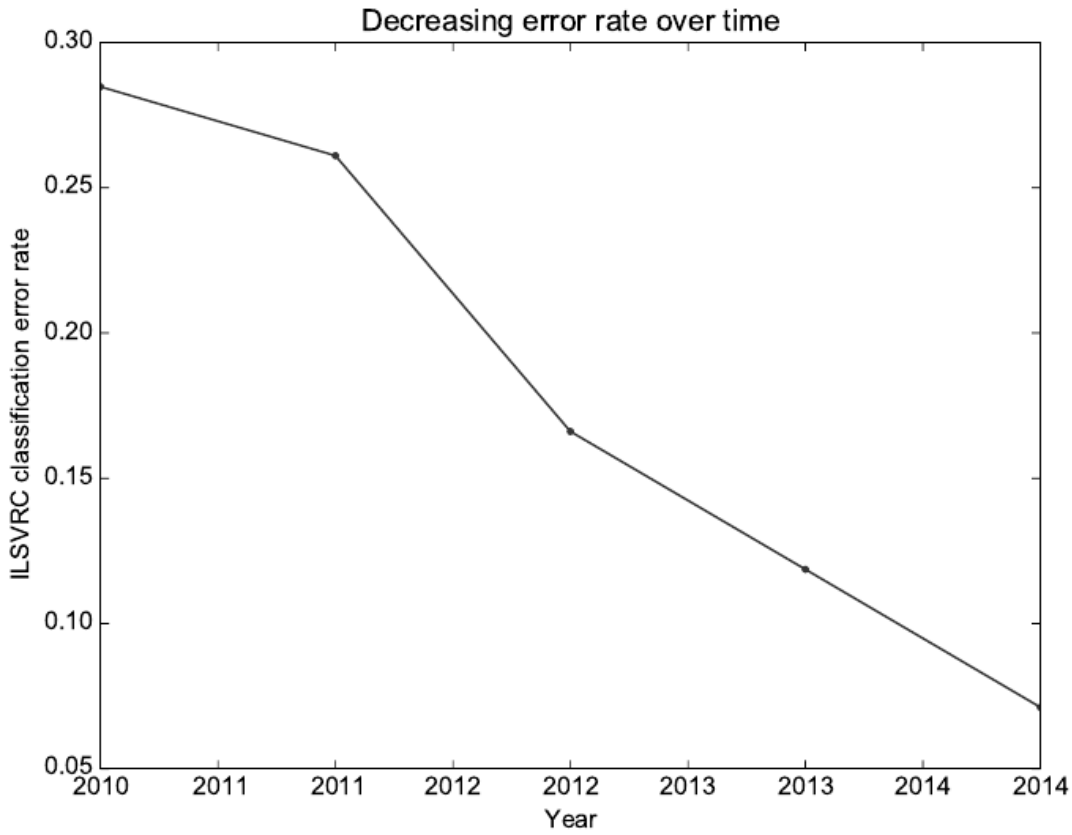


Figure 1.10: Since deep networks reached the scale necessary to compete in the ImageNet Large Scale Visual Recognition, they have consistently won the competition every year, and yielded lower and lower error rates each time. Data from Russakovsky *et al.* (2014b).

Part 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 allow 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.

Chapter 2

Linear Algebra

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. However, 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 begin the technical content of the book with a focused presentation of the key linear algebra ideas that are most important in deep learning.

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 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 will completely omit 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 lower-case 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 have an order to them, and we can identify each individual number by its index in that ordering. Typically we give vectors lower case names written 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:

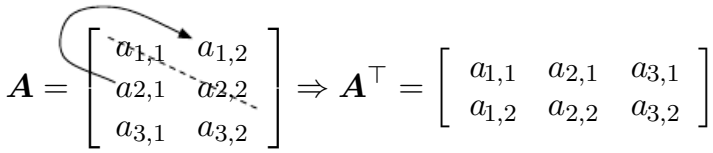
$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

  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 of the 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 upper-case 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 of 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 the i-th *column* of $\boldsymbol{A}$. When we need to explicitly identify the elements of a matrix, we write them as an array enclosed in square brackets:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}.$$



$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \Rightarrow \boldsymbol{A}^\top = \begin{bmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \end{bmatrix}$$

Figure 2.1: The transpose of the matrix can be thought of as a mirror image across the main diagonal.

Sometimes we may need to index matrix-valued expressions that are not just a single letter. In this case, we use subscripts after the expression, but do not convert anything to lower case. For example, $f(\boldsymbol{A})_{i,j}$ gives element (i, j) of the matrix computed by applying the function f to $\boldsymbol{A}$.

- *Tensors*: In some cases we will need an array with more than two axes. In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a *tensor.* We denote a tensor named "A" with this typeface: A. We identify the element of A at coordinates (i, j, k) by writing $A_{i,j,k}$.

By convention, we consider that adding (or subtracting) a scalar and a vector yields a vector with the additive operation performed on each element. The same thing happens with a matrix or a tensor. This is called a *broadcasting* operation in Python's `numpy` library.

One important operation on matrices is the *transpose*. The transpose of a matrix is the mirror image of the matrix across a diagonal line, called the *main diagonal*, running down and to the right, starting from its upper left corner. See Fig. 2.1 for a graphical depiction of this operation. We denote the transpose of a matrix $\boldsymbol{A}$ as $\boldsymbol{A}^\top$, and it is defined such that

$$(\boldsymbol{A}^\top)_{i,j} = A_{j,i}\,.$$

Vectors can be thought of as matrices that contain only one column. The transpose of a vector is therefore a matrix with only one row. Sometimes we define a vector by writing out its elements in the text inline as a row matrix, then using the transpose operator to turn it into a standard column vector, e.g. $\boldsymbol{x} = [x_1, x_2, x_3]^\top$.

We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ where $C_{i,j} = A_{i,j} + B_{i,j}$.

We can also add a scalar to a matrix or multiply a matrix by a scalar, just by performing that operation on each element of a matrix: $\boldsymbol{D} = a \cdot \boldsymbol{B} + c$ where $D_{i,j} = a \cdot B_{i,j} + c$.

2.2 Multiplying Matrices and Vectors

One of the most important operations involving matrices is multiplication of two matrices. The *matrix product* of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is a third matrix $\boldsymbol{C}$. In order for this product to be defined, $\boldsymbol{A}$ must have the same number of columns as $\boldsymbol{B}$ has rows. If $\boldsymbol{A}$ is of shape $m \times n$ and $\boldsymbol{B}$ is of shape $n \times p$, then $\boldsymbol{C}$ is of shape $m \times p$. We can write the matrix product just by placing two or more matrices together, e.g.

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}.$$

The product operation is defined by

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}.$$

Note that the standard product of two matrices is *not* just a matrix containing the product of the individual elements. Such an operation exists and is called the *element-wise product* or *Hadamard product*, and is denoted in this book[1] as $\boldsymbol{A} \odot \boldsymbol{B}$.

The *dot product* between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of the same dimensionality is the matrix product $\boldsymbol{x}^\top \boldsymbol{y}$. We can think of the matrix product $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$ as computing $c_{i,j}$ as the dot product between row i of $\boldsymbol{A}$ and column j of $\boldsymbol{B}$.

Matrix product operations have many useful properties that make mathematical analysis of matrices more convenient. For example, matrix multiplication is distributive:

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{A}\boldsymbol{B} + \boldsymbol{A}\boldsymbol{C}.$$

It is also associative:

$$\boldsymbol{A}(\boldsymbol{B}\boldsymbol{C}) = (\boldsymbol{A}\boldsymbol{B})\boldsymbol{C}.$$

Matrix multiplication is *not* commutative, unlike scalar multiplication.

The transpose of a matrix product also has a simple form:

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

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:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.1}$$

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

[1] The element-wise product is used relatively rarely, so the notation for it is not as standardized as the other operations described in this chapter.

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

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

one of these unknowns to solve for. Each row of $\boldsymbol{A}$ and each element of $\boldsymbol{b}$ provide another constraint. We can rewrite equation 2.1 as:

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

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

$$\cdots$$

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

or, even more explicitly, as:

$$a_{1,1}\, x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1$$

$$a_{2,1}\, x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2$$

$$\cdots$$

$$a_{m,1}\, x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m.$$

Matrix-vector product notations 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 allows us to solve equation 2.1 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 n-dimensional identity matrix as $\boldsymbol{I}_n$. Formally,

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

The structure of the identity matrix is simple: all of the entries along the main diagonal are 1, while all of the other entries are zero. See Fig. 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

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

We can now solve equation 2.1 by the following steps:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$
$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$
$$\boldsymbol{I}_n\,\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$
$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}.$$

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

2.4 Linear Dependence, Span, and Rank

In order for $\boldsymbol{A}^{-1}$ to exist, equation 2.1 must have exactly one solution for every value of $\boldsymbol{b}$. However, 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 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

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

is also a solution for any real α.

To analyze how many solutions the equation has, we can think of the columns of $\boldsymbol{A}$ as specifying different directions we can travel from the *origin* (the point specified by the vector of all zeros), and 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, i.e. x_i specifies how far to move in the direction of column i:

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

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:

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

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

Determining whether $\boldsymbol{Ax} = \boldsymbol{b}$ 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 $\boldsymbol{Ax} = \boldsymbol{b}$ 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. This implies immediately that $\boldsymbol{A}$ must have at least m columns, i.e., $n \geq m$. Otherwise, the dimensionality of the column space must 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 allows 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 equal to each other. 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 have at least m linearly independent columns. This condition is both necessary and sufficient for equation 2.1 to have a solution for every value of $\boldsymbol{b}$.

In order for the matrix to have an inverse, we additionally need to ensure that equation 2.1 has *at most* one solution for each value of $\boldsymbol{b}$. To do so, we need to ensure 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 of the columns must 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, it can still be possible to solve the equation. However, we can not 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:

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

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

2.5 Norms

Sometimes we need to measure the size of a vector. In machine learning, we usually measure the size of vectors using an L^p *norm*:

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

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

Norms, including the L^p norm, are functions mapping vectors to non-negative values, satisfying these properties that make them behave like distances between points:

- $f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$
- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$ (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*. It is simply the Euclidean distance from the origin to the point identified by $\boldsymbol{x}$. This is probably the most common norm used in machine learning. 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, the derivatives of the squared L^2 norm with respect to each element of $\boldsymbol{x}$ each depend only on the corresponding element of $\boldsymbol{x}$, while all of 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 retains mathematical simplicity: the L^1 norm. The L^1 norm may be simplified to

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

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 (and when we use the L^1 norm, we often use it as a proxy for this function). Some authors refer to this function as the "l_0 norm," but this is incorrect terminology, because scaling the vector by α does not change the number of nonzero entries.

One other norm that commonly arises in machine learning is the l_∞ norm, also known as *the max norm*. This norm simplifies to

$$||\boldsymbol{x}||_\infty = \max_i |x_i|,$$

e.g., the absolute value of the element with the largest magnitude in the vector.

Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the otherwise obscure *Frobenius norm*

$$||A||_F = \sqrt{\sum_{i,j} a_{i,j}^2},$$

which is analogous to the L^2 norm of a vector.

The dot product of two vectors can be rewritten in terms of norms. Specifically,

$$\boldsymbol{x}^\top \boldsymbol{y} = ||\boldsymbol{x}||_2 ||\boldsymbol{y}||_2 \cos\theta$$

where θ is the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$.

2.6 Special Kinds of Matrices and Vectors

Some special kinds of matrices and vectors are particularly useful.

Diagonal matrices only have non-zero entries along the main diagonal. Formally, a matrix $\boldsymbol{D}$ is diagonal if and only if $d_{i,j} = 0$ for all $i \neq j$. We've already seen one example of a diagonal matrix: the identity matrix, where all of the diagonal entries are 1. In this book[2], we write $\text{diag}(\boldsymbol{v})$ to denote a square diagonal

[2] There is not a standardized notation for constructing a diagonal matrix from a vector.

matrix whose diagonal entries are given by the entries of the vector $\boldsymbol{v}$. Diagonal matrices are of interest in part because multiplying by a diagonal matrix is very computationally efficient. To compute $\text{diag}(\boldsymbol{v})\boldsymbol{x}$, we only need to scale each element x_i by v_i. In other words, $\text{diag}(\boldsymbol{v})\boldsymbol{x} = \boldsymbol{v} \odot \boldsymbol{x}$. Inverting a square diagonal matrix is also efficient. The inverse exists only if every diagonal entry is nonzero, and in that case, $\text{diag}(\boldsymbol{v})^{-1} = \text{diag}([1/v_1, \ldots, 1/v_n]^\top)$. In many cases, we may derive some very general machine learning algorithm in terms of arbitrary matrices, but obtain a less expensive (and less descriptive) algorithm by restricting some matrices to be diagonal.

Note that not all diagonal matrices need be square. It is possible to construct a rectangular diagonal matrix. Non-square diagonal matrices do not have inverses but it is still possible to multiply by them cheaply. For a non-square diagonal matrix $\boldsymbol{D}$, the product $\boldsymbol{Dx}$ will involving scaling each element of $\boldsymbol{x}$, and either concatenating some zeros to the result if $\boldsymbol{D}$ is taller than it is wide, or discarding some of the last elements of the vector if $\boldsymbol{D}$ is wider than it is tall.

A *symmetric* matrix is any matrix that is equal to its own transpose:

$$\boldsymbol{A} = \boldsymbol{A}^\top .$$

Symmetric matrices often arise when the entries are generated by some function of two arguments that does not depend on the order of the arguments. For example, if $\boldsymbol{A}$ is a matrix of distance measurements, with $a_{i,j}$ giving the distance from point i to point j, then $a_{i,j} = a_{j,i}$ because distance functions are symmetric.

A *unit vector* is a vector with *unit norm*:

$$||\boldsymbol{x}||_2 = 1.$$

A vector $\boldsymbol{x}$ and a vector $\boldsymbol{y}$ are *orthogonal* to each other if $\boldsymbol{x}^\top \boldsymbol{y} = 0$. If both vectors have nonzero norm, this means that they are at 90 degree angles to each other. In $\mathbb{R}^n$, at most n vectors may be mutually orthogonal with nonzero norm. If the vectors are not only orthogonal but also have unit norm, we call them *orthonormal*.

An *orthogonal matrix* is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

$$\boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{I}.$$

This implies that

$$\boldsymbol{A}^{-1} = \boldsymbol{A}^\top,$$

so orthogonal matrices are of interest because their inverse is very cheap to compute. Pay careful attention to the definition of orthogonal matrices. Counterintuitively, their rows are not merely orthogonal but fully orthonormal. There

is no special term for a matrix whose rows or columns are orthogonal but not orthonormal.

2.7 Eigendecomposition

Many mathematical objects can be understood better by breaking them into constituent parts, or finding some properties of them that are universal, not caused by the way we choose to represent them.

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, such as that 12 is not divisible by 5, or 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 non-zero vector v such that multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$:

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

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

Note that 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 only look for unit eigenvectors.

We can represent the matrix $\boldsymbol{A}$ using an *eigendecomposition*, with eigenvectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ and corresponding eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$ by concatenating the eigenvectors into a matrix $\boldsymbol{V} = [\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}]$, (i.e. one column per eigenvector) and concatenating the eigenvalues into a vector $\boldsymbol{\lambda}$. Then the matrix

$$\boldsymbol{A} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$$

has the desired eigenvalues and eigenvectors. If we make $\boldsymbol{V}$ an orthogonal matrix, then we can think of $\boldsymbol{A}$ as scaling space by λ_i in direction $\boldsymbol{v}^{(i)}$. See Fig. 2.3 for an example.

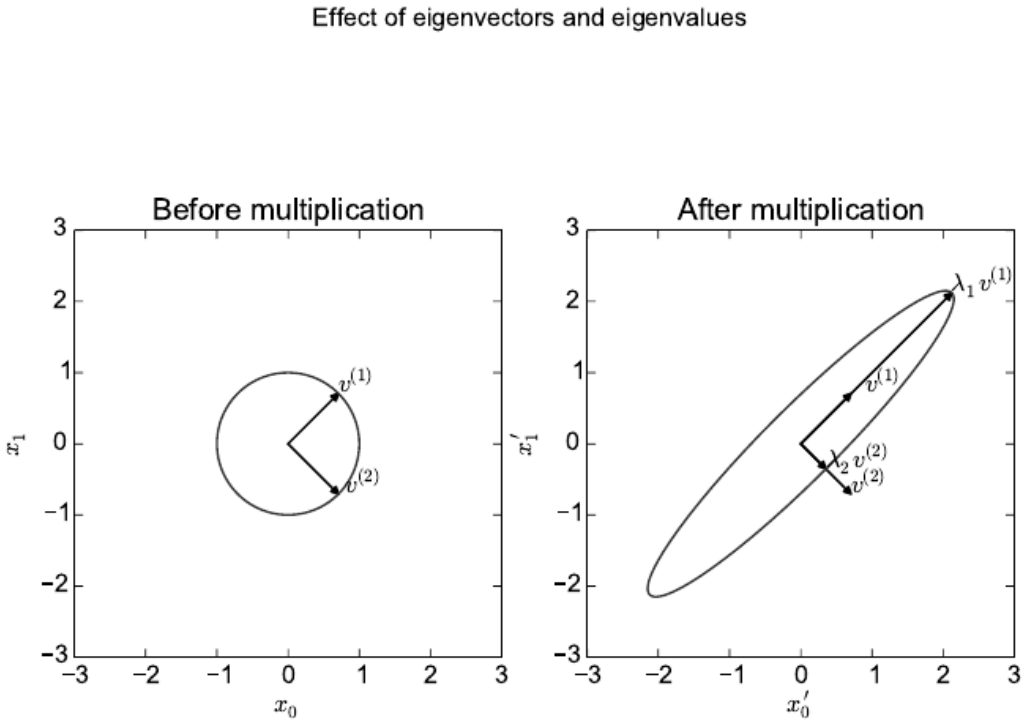


Figure 2.3: An example of the effect of eigenvectors and eigenvalues. Here, we have a matrix $\boldsymbol{A}$ with two orthonormal eigenvectors, $\boldsymbol{v}^{(1)}$ with eigenvalue λ_1 and $\boldsymbol{v}^{(2)}$ with eigenvalue λ_2. *Left)* We plot the set of all unit vectors $\boldsymbol{u} \in \mathbb{R}^2$ as a unit circle. *Right)* We plot the set of all points $\boldsymbol{Au}$. 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.

We have seen that *constructing* matrices with specific eigenvalues and eigenvectors allows us to stretch space in desired directions. However, we often want to *decompose* matrices into their eigenvalues and eigenvectors. Doing so can help us to 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 may involve 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:

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

where $\boldsymbol{Q}$ is an orthogonal matrix composed of eigenvectors of $\boldsymbol{A}$, and $\boldsymbol{\Lambda}$ is a diagonal matrix, with $\lambda_{i,i}$ being the eigenvalue corresponding to $\boldsymbol{Q}_{:,i}$.

While any real symmetric matrix $\boldsymbol{A}$ is guaranteed to have an eigendecomposition, the eigendecomposition is not 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 of the eigenvalues are unique.

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 0. The eigendecomposition 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 Sec. 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 allows us to discover

some of the same kind of information as the eigendecomposition. 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

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

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

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

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 non-zero 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 non-square 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

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

by left-multiplying each side to obtain

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

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

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

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

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

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 non-zero elements then taking the transpose of the resulting matrix.

When $\boldsymbol{A}$ has more rows than columns, then solving a linear equation using 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 columns than rows, 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 of the diagonal entries of a matrix:

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_i a_{i,i}.$$

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:

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

The trace operator also has many useful properties that make it easy to manipulate expressions involving the trace operator. For example, the trace operator is invariant to the transpose operator:

$$\mathrm{Tr}(\boldsymbol{A}) = \mathrm{Tr}(\boldsymbol{A}^\top).$$

The trace of a square matrix composed of many factors is also invariant to moving the last factor into the first position:

$$\text{Tr}(\boldsymbol{ABC}) = \text{Tr}(\boldsymbol{CAB}) = \text{Tr}(\boldsymbol{BCA})$$

or more generally,

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

Another useful fact to keep in mind is that a scalar is its own trace, i.e. $a = Tr(a)$. This can be useful when wishing to manipulate inner products. Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be two column vectors in $\mathbb{R}^n$

$$\boldsymbol{a}^\top \boldsymbol{b} = \text{Tr}(\boldsymbol{a}^\top \boldsymbol{b}) = \text{Tr}(\boldsymbol{b}\boldsymbol{a}^\top).$$

2.11 Determinant

The determinant of a square matrix, denoted $\det(\boldsymbol{A})$, is a function mapping matrices to real scalars. The determinant is equal to the product of all the matrix's eigenvalues. The absolute value of the determinant can be thought of as a measure of how much multiplication by the matrix expands or contracts space. If the determinant is 0, then space is contracted completely along at least one dimension, causing it to lose all of its volume. If the determinant is 1, then the transformation is volume-preserving.

2.12 Example: Principal Components Analysis

One simple machine learning algorithm, *principal components analysis (PCA)* can be derived using only knowledge of basic linear algebra.

Suppose we have a collection of m points $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ in $\mathbb{R}^n$. Suppose we would like to apply lossy compression to these points, i.e. we would like to find a way of storing the points that requires less memory but may lose some precision. We would like to lose as little precision as possible.

One way we can encode these points is to represent a lower-dimensional version of them. For each point $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ we will find a corresponding code vector $\boldsymbol{c}^{(i)} \in \mathbb{R}^l$. If l is smaller than n, it will take less memory to store the code points than the original data. We can use matrix multiplication to map the code back into $\mathbb{R}^n$. Let the reconstruction $r(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}$, where $\boldsymbol{D} \in \mathbb{R}^{n \times l}$ is the matrix defining the decoding.

To simplify the computation of the optimal encoding, we constrain the columns of $\boldsymbol{D}$ to be orthogonal to each other. (Note that $\boldsymbol{D}$ is still not technically "an orthogonal matrix" unless $l = n$)

With the problem as described so far, many solutions are possible, because we can increase the scale of $\boldsymbol{D}_{:,i}$ if we decrease c_i proportionally for all points. To give the problem a unique solution, we constrain all of the columns of $\boldsymbol{D}$ to have unit norm.

In order to turn this basic idea into an algorithm we can implement, the first thing we need to do is figure out how to generate the optimal code point $\boldsymbol{c}^*$ for each input point $\boldsymbol{x}$. One way to do this is to minimize the distance between the input point $\boldsymbol{x}$ and its reconstruction, $r(\boldsymbol{c})$. We can measure this distance using a norm. In the principal components algorithm, we use the L^2 norm:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - r(\boldsymbol{c})||_2.$$

We can switch to the squared L^2 norm instead of the L^2 norm itself, because both are minimized by the same value of $\boldsymbol{c}$. This is because the L^2 norm is non-negative and the squaring operation is monotonically increasing for non-negative arguments.

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - r(\boldsymbol{c})||_2^2$$

The function being minimized simplifies to

$$(\boldsymbol{x} - r(\boldsymbol{c}))^\top (\boldsymbol{x} - r(\boldsymbol{c}))$$

(by the definition of the L^2 norm)

$$= \boldsymbol{x}^\top \boldsymbol{x} - \boldsymbol{x}^\top r(\boldsymbol{c}) - r(\boldsymbol{c})^\top \boldsymbol{x} + r(\boldsymbol{c})^\top r(\boldsymbol{c})$$

(by the distributive property)

$$= \boldsymbol{x}^\top \boldsymbol{x} - 2\boldsymbol{x}^\top r(\boldsymbol{c}) + r(\boldsymbol{c})^\top r(\boldsymbol{c})$$

(because a scalar is equal to the transpose of itself).

We can now change the function being minimized again, to omit the first term, since this term does not depend on $\boldsymbol{c}$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top r(\boldsymbol{c}) + r(\boldsymbol{c})^\top r(\boldsymbol{c}).$$

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

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

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

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

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

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

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

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

$$\boldsymbol{c} = \boldsymbol{D}^\top \boldsymbol{x}. \tag{2.2}$$

This is good news: we can optimally encode $\boldsymbol{x}$ just using a vector-product operation.

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. However, since we will use the same matrix $\boldsymbol{D}$ to decode all of 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:

$$\boldsymbol{D}^* = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j} (x_j^{(i)} - r_j^{(i)})^2} \text{ subject to } \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l \tag{2.3}$$

To derive the algorithm for finding $\boldsymbol{D}^*$, we will start by considering the case where $l = 1$. In this case, $\boldsymbol{D}$ is just a single vector, $\boldsymbol{d}$. Substituting equation 2.2 into equation 2.3 and simplifying $\boldsymbol{D}$ into $\boldsymbol{d}$, the problem reduces to

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

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

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

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

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

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

(by the alternate definition of the Frobenius norm)

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

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

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

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

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

(because the trace is invariant to transpose)

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

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

At this point, we re-introduce the constraint:

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

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

(due to the constraint)

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

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

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

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.

In the general case, where $l > 1$, $\boldsymbol{D}$ 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.

Chapter 3

Probability and Information Theory

In this chapter, we describe probability theory. Probability theory is a mathematical framework for representing uncertain statements. It provides a means of quantifying uncertainty and 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. If you are already familiar with probability theory, feel free to skip this chapter. If you have absolutely no prior experience with probability, 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).

3.1 Why Probability?

Many branches of computer science deal mostly with entities that are entirely deterministic and certain. A programmer can usually safely assume that a CPU will execute each machine instruction flawlessly. Errors in hardware do occur, but are rare enough that most software applications do not need to be designed to account for them. Given that many computer scientists and software engineers work in a relatively clean and certain environment, it can be surprising that

machine learning makes heavy use of probability theory.

This is because machine learning must always deal with uncertain quantities, and sometimes may also need to deal with stochastic quantities. Uncertainty and stochasticity can arise from many sources. Researchers have made compelling arguments for quantifying uncertainty using probability since at least the 1980s. Many of the arguments presented here are summarized from or inspired by (Pearl, 1988). Much earlier work in probability and engineering introduced and developed the underlying fundamental notions, such as the notion of exchangeability (de Finetti, 1937), Cox's theorem as the foundations of Bayesian inference (Cox, 1946), and the theory of stochastic processes (Doob, 1953).

Nearly all activities require some ability to reason in the presence of uncertainty. In fact, beyond mathematical statements that are true by definition, it is difficult to think of any proposition that is absolutely true or any event that is absolutely guaranteed to occur.

One source of uncertainty is incomplete observability. When we cannot observe something, we are uncertain about its true nature. In machine learning, it is often the case that we can observe a large amount of data, but there is not a data instance for every situation we care about. We are also generally not able to observe directly what process generates the data. Since we are uncertain about what process generates the data, we are also uncertain about what happens in the situations for which we have not observed data points. Lack of observability can also give rise to apparent stochasticity. Deterministic systems can appear stochastic when we cannot observe all of the variables that drive the behavior of the system. For example, consider a game of Russian roulette. The outcome is deterministic if you know which chamber of the revolver is loaded. If you do not know this important information, then it is a game of chance. In many cases, we are able to observe some quantity, but our measurement is itself uncertain. For example, laser range finders may have several centimeters of random error.

Uncertainty can also arise from the simplifications we make in order to model real-world processes. For example, if we discretize space, then we immediately become uncertain about the precise position of objects: each object could be anywhere within the discrete cell that we know it occupies.

Conceivably, the universe itself could have stochastic dynamics, but we make no claim on this subject.

In many cases, it is more practical to use a simple but uncertain rule rather than a complex but certain one, even if our modeling system has the fidelity to accommodate a complex rule. For example, the simple rule "Most birds fly" is cheap to develop and is broadly useful, while a rule of the form, "Birds fly, except for very young birds that have not yet learned to fly, sick or injured birds that have lost the ability to fly, flightless species of birds including the cassowary, ostrich,

and kiwi. . . " is expensive to develop, maintain, and communicate, and after all of this effort is still very brittle and prone to failure.

Given that we need a means of representing and reasoning about uncertainty, it is not immediately obvious that probability theory can provide all of the tools we want for artificial intelligence applications. Probability theory was originally developed to analyze the frequencies of events. It is easy to see how probability theory can be used to study events like drawing a certain hand of cards in a game of poker. These kinds of events are often repeatable, and when we say that an outcome has a probability p of occurring, it means that if we repeated the experiment (e.g., draw a hand of cards) infinitely many times, then proportion p of the repetitions would result in that outcome. This kind of reasoning does not seem immediately applicable to propositions that are not repeatable. If a doctor analyzes a patient and says that the patient has a 40% chance of having the flu, this means something very different—we can not make infinitely many replicas of the patient, nor is there any reason to believe that different replicas of the patient would present with the same symptoms yet have varying underlying conditions. In the case of the doctor diagnosing the patient, we use probability to represent a *degree of belief*, with 1 indicating absolute certainty, and 0 indicating absolute uncertainty. The former kind of probability, related directly to the rates at which events occur, is known as *frequentist probability*, while the latter, related to qualitative levels of certainty, is known as *Bayesian probability*.

It turns out that if we list several properties that we expect common sense reasoning about uncertainty to have, then the only way to satisfy those properties is to treat Bayesian probabilities as behaving exactly the same as frequentist probabilities. For example, if we want to compute the probability that a player will win a poker game given that she has a certain set of cards, we use exactly the same formulas as when we compute the probability that a patient has a disease given that she 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 lower case letter in plain type-

face, and the values it can take on with lower case 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 probability mass function to use based on the identity of the random variable, rather than 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. $P(x)$ denotes the probability that $\mathrm{x} = x$, 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: $x \sim P(\mathrm{x})$.

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 probability mass function on a set of random variables x, a function f must meet the following properties:

- The domain of f must be the set of all possible states of x.
- The range of f must be a subset of the real interval $[0, 1]$. (No state can be more likely than a guaranteed event or less likely than an impossible event.)
- $\sum_{x \in \mathrm{x}} f(x) = 1$. ($f$ must guarantee that *some* state occurs.)

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 probability mass function to

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

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 $\sum_i P(\mathrm{x} = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1$, 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. A probability density function must satisfy the following properties:

- It must map from the domain of the random variable whose distribution it describes to the real numbers.
- $\forall x, p(x) \geq 0$. Note that we do not require $p(x) \leq 1$.
- $\int p(x)dx = 1$.

A probability density function 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 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 probability density function 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 nonnegative everywhere. Additionally, it integrates to 1. We often denote that x follows the uniform distribution on $[a, b]$ by writing $x \sim U(a, b)$.

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*.

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*:

$$\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_{y} P(\mathrm{x} = x, \mathrm{y} = y).$$

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:

$$p(x) = \int p(x, y)dy.$$

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

$$P(\mathrm{y} = y \mid \mathrm{x} = x) = P(\mathrm{y} = y, \mathrm{x} = x)/P(\mathrm{x} = x).$$

Note that this 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:

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

This observation is known as the *chain rule* or *product rule* of probability. It follows immediately from the definition of conditional probability. For example, applying the definition twice, we get

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

Note how every statement about probabilities remains true if we add conditions (stuff on the right-hand side of the vertical bar) consistently on all the "P"'s in the statement. We can use this to derive the same thing differently:

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

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:

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

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:

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

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:

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

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

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

When the identity of the distribution is clear from the context, we may simply write the name of the random variable that the expectation is over, e.g. $\mathbb{E}_{\mathrm{x}}[f(x)]$. If it is clear which random variable the expectation is over, we may omit the subscript entirely, e.g. $\mathbb{E}[f(x)]$. By default, we can assume that $\mathbb{E}[\cdot]$ averages over the values of all the random variables inside the brackets. Likewise, when there is no ambiguity, we may omit the square brackets.

Expectations are linear, for example, $\mathbb{E}[\alpha f(x)+\beta g(x)] = \alpha\mathbb{E}[f(x)]+\beta\mathbb{E}[g(x)]$, when α and β are fixed (not random and not depending on x).

The *variance* gives a measure of how much the different values of a function are spread apart:

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

When the variance is low, the values of $f(x)$ cluster near their expected value. The square root of the variance is known as the *standard deviation*.

The *covariance* gives some sense of how much two values are linearly related to each other, as well as the scale of these variables:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}\left[(f(x) - \mathbb{E}\left[f(x)\right])\left(g(y) - \mathbb{E}\left[g(y)\right]\right)\right].$$

High absolute values of the covariance mean that the values change a lot and are both far from their respective means at the same time. If the sign of the covariance is positive, then the values tend to change in the same direction, while if it is negative, they tend to change in opposite directions. Other measures such as *correlation* normalize the contribution of each variable in order to measure only how much the variables are related, rather than also being affected by the scale of the separate variables.

The notions of covariance and dependence are conceptually related, but are in fact distinct concepts. Two random variables that have non-zero covariance are dependent. However, they may have zero covariance without being independent.

For example, suppose we first generate x, then generate $s \in \{-1, 1\}$ with each state having probability 0.5, then generate y as $s(x - \mathbb{E}[x])$. Clearly, x and y are not independent, because y only has two possible values given x. However, $\text{Cov}(x, y) = 0$.

The *covariance matrix* of a random vector $\boldsymbol{x} \in \mathbb{R}^n$ is an $n \times n$ matrix, such that

$$\text{Cov}(\mathbf{x})_{i,j} = \text{Cov}(\mathrm{x}_i, \mathrm{x}_j).$$

Note that the diagonal elements give $\text{Cov}(\mathrm{x}_i, \mathrm{x}_i) = \text{Var}(\mathrm{x}_i)$.

3.9 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 quantify similarity between probability distributions. For more detail on information theory, see (Cover and Thomas, 2006; 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. Specifically,

- 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.

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

$$I(x) = -\log P(x). \tag{3.1}$$

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*[1]:

$$H(\mathrm{x}) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log P(x)]. \tag{3.2}$$

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 actually gives a lower bound on the number of bits (if the logarithm is base 2, otherwise the units are different) needed in 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 Fig. 3.1 for a demonstration. When x is continous, 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 the *Kullback-Leibler (KL) divergence*:

$$D_{\mathrm{KL}}(Q\|P) = \mathbb{E}_{x\sim Q}\left[\log \frac{Q(x)}{P(x)}\right]. \tag{3.3}$$

In the case of discrete variables, it is the extra amount of information needed to send a message containing symbols drawn with probability Q, when we incorrectly believe that they were drawn with probability P, i.e., it measures how much it hurts to use P as a model when Q is the ground truth. The KL divergence has

[1] Shannon entropy is named for Claude Shannon, the father of information theory (Shannon, 1948, 1949). For an interesting biographical account of Shannon and some of his contemporaries, see *Fortune's Formula* by William Poundstone (Poundstone, 2005).

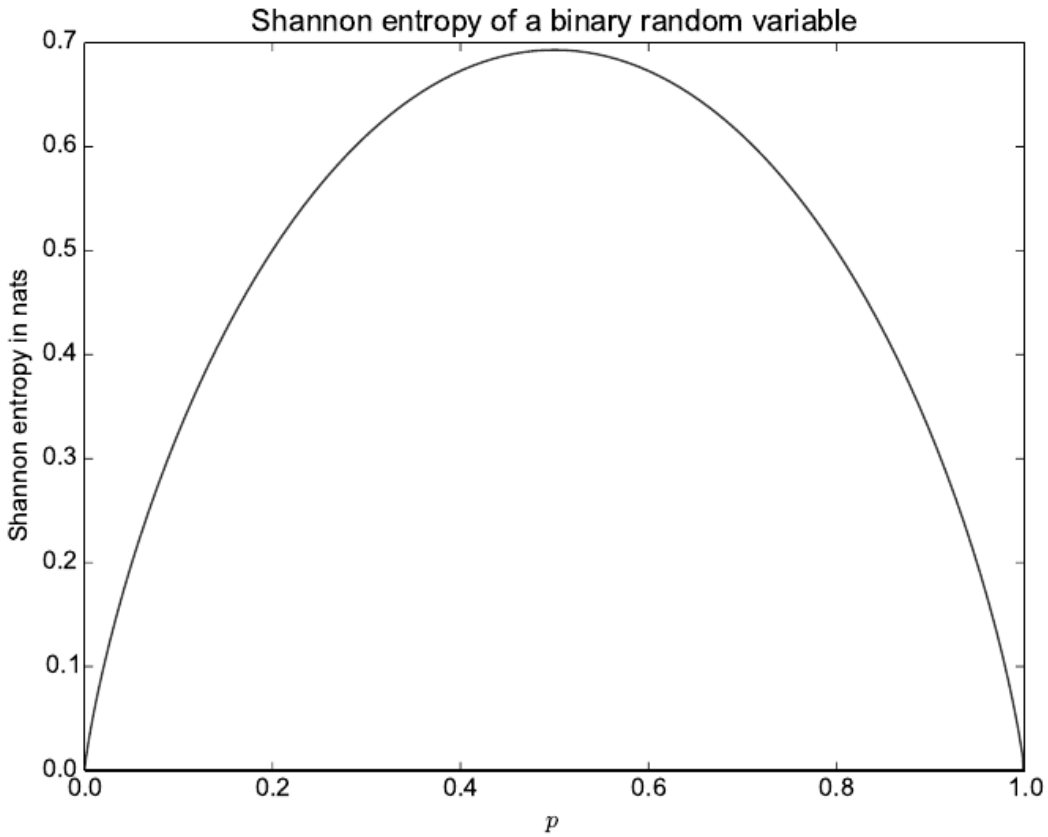


Figure 3.1: This plot shows how distributions that are closer to deterministic have low Shannon entropy while distributions that are close to uniform have high Shannon entropy. On the horizontal axis, we plot p, the probability of a binary random variable being equal to 1. When p is near 0, the distribution 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.

many useful properties, most notably that it is 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 (see section 3.13 for details). 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. However, it is not a true distance measure because it is not symmetric, i.e. $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$ for some P and Q.

When computing many of these quantities, it is common to encounter expressions of the form $0 \log 0$. By convention, in the context of information theory, we treat these expressions as $\lim_{x \to 0} x \log x = 0$.

3.10 Common Probability Distributions

Several simple probability distributions are useful in many contexts in machine learning.

3.10.1 Bernoulli Distribution

The *Bernoulli* distribution is a distribution over a single binary random variable. It is controlled by a single parameter $\phi \in [0, 1]$, which gives the probability of the random variable being equal to 1. It has the following properties:

$$P(\mathrm{x} = 1) = \phi$$

$$P(\mathrm{x} = 0) = 1 - \phi$$

$$P(\mathrm{x} = x) = \phi^{x}(1 - \phi)^{1-x}$$

$$\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = \phi$$

$$\mathrm{Var}_{\mathrm{x}}(\mathrm{x}) = \phi(1 - \phi)$$

$$H(\mathrm{x}) = (\phi - 1)\log(1 - \phi) - \phi \log \phi.$$

3.10.2 Multinoulli Distribution

The *multinoulli* or *categorical* distribution is a distribution over a single discrete variable with k different states, where k is finite[2]. The multinoulli distribution

[2] "Multinoulli" is a recently coined term. The multinoulli distribution is a special case of the *multinomial* distribution. A multinomial distribution is the distribution over vectors in $\{0, \ldots, k\}^n$ representing how many times each of the k categories is visited when n samples are drawn from a multinoulli distribution. Many texts use the term "multinomial" to refer to multinoulli distributions without clarifying that they refer only to the $n = 1$ case.

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, etc. 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. This is because they model discrete variables for which it is feasible to simply enumerate all of 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.

3.10.3 Gaussian Distribution

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

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

See Fig. 3.2 for a schematic.

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

Note that 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 $\beta \in \mathbb{R}^+$ to control the *precision* or inverse variance of the distribution:

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

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.

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.

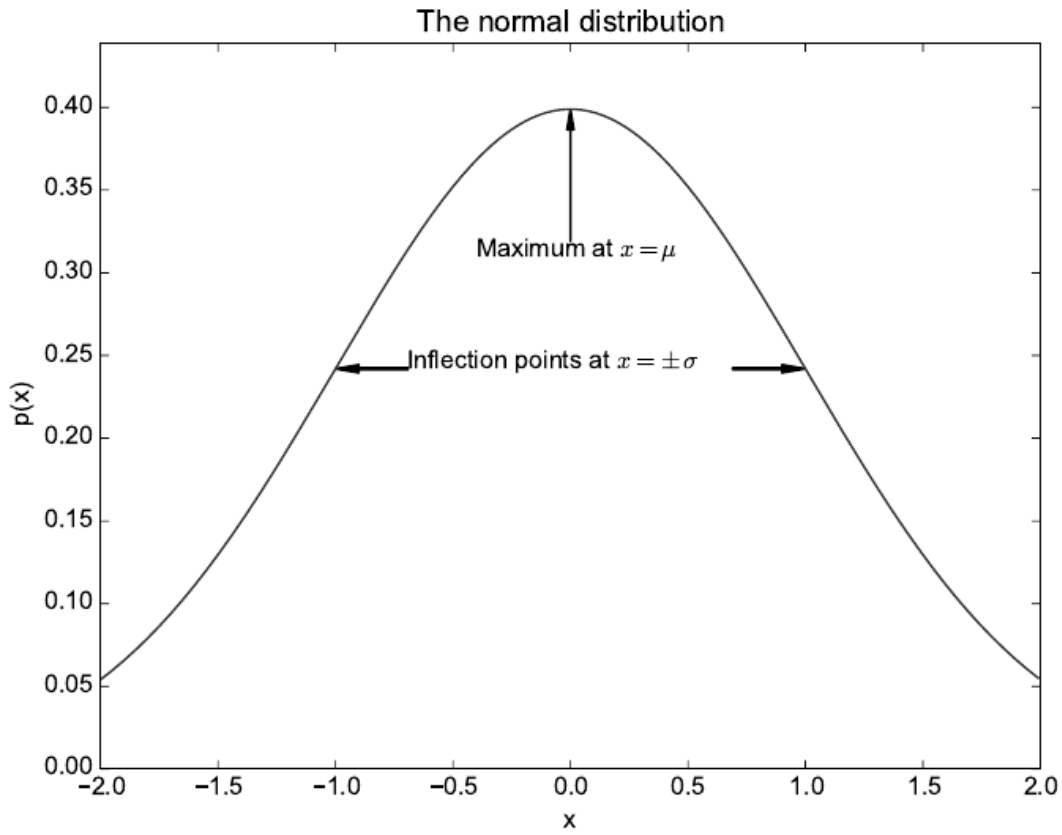


Figure 3.2: *The normal distribution*: The normal distribution $\mathcal{N}(x \mid \mu, \sigma^2)$ exhibits a classic "bell curve" shape, with the x coordinate of its central peak given by μ, and the width of its peak controlled by σ. In this example, we depict the *standard normal distribution*, with $\mu = 0$ and $\sigma = 1$.

Second, the normal distribution in some sense makes the fewest assumptions of any distribution over the reals, so choosing to use it inserts the least amount of prior knowledge into a model. Out of all distributions with the same variance, the normal distribution has the highest entropy. It is not possible to place a uniform distribution on all of $\mathbb{R}$. The closest we can come to doing so is to use a normal distribution with high variance.

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 $\boldsymbol{\Sigma}$:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sqrt{\frac{1}{(2\pi)^n \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) \right).$$

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, the covariance is not necessarily the most 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* $\boldsymbol{\beta}$:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \sqrt{\frac{\det(\boldsymbol{\beta})}{(2\pi)^n}} \exp\left(-\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\beta} (\boldsymbol{x} - \boldsymbol{\mu}) \right).$$

3.10.4 Dirac Distribution

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

$$p(x) = \delta(x - \mu).$$

The Dirac delta function is defined such that it is zero-valued everywhere but 0, yet integrates to 1. 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 the so-called *empirical distribution*,

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \delta(x - x_i) \tag{3.4}$$

which puts probability mass $\frac{1}{n}$ on each of the n points $x_1, \ldots x_n$ 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 to 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 5.8). Many machine learning algorithms can be configured to have arbitrarily high capacity. If given enough capacity, these algorithms will simply learn the empirical distribution. This is a bad outcome because the model does not generalize at all and assigns infinitesimal probability to any point in space that did not occur in the training set. A central problem in machine learning is studying how to limit the capacity of a model in a way that prevents it from simply learning the empirical distribution while also allowing it to learn complicated functions.

The empirical distribution is a particular form of *mixture*, discussed next.

3.10.5 Mixtures of Distributions and Gaussian Mixture

It is also common to define probability distributions by composing 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 generates the sample is determined by sampling a component identity from a multinoulli distribution:

$$P(\mathrm{x}) = \sum_i P(\mathrm{c} = i) P(\mathrm{x} \mid \mathrm{c} = i)$$

where $P(\mathrm{c})$ is the multinoulli distribution over component identities. In chapter 13, we explore the art of building complex probability distributions from simple ones in more detail. Note that we can think of the variable c as a non-observed (or latent) random variable that is related to x through their joint distribution $P(\mathrm{x}, \mathrm{c}) = P(\mathrm{x} \mid \mathrm{c}) P(\mathrm{c})$. Latent variables are discussed further in Section 13.4.2.

A very powerful and common type of mixture model is the *Gaussian mixture* model, in which the components $P(\mathrm{x} \mid \mathrm{c} = i)$ are Gaussians, each with its mean μ_i and covariance Σ_i. Some mixtures can have more constraints, for example, the covariances could be shared across components, i.e., $\Sigma_i = \Sigma_j = \Sigma$, or the covariance matrices could be constrained to be diagonal or simply equal to a scalar times the identity. A Gaussian mixture model is a *universal approximator* of densities, in the sense that any smooth density can be approximated to a particular precision by a Gaussian mixture model with enough components. Gaussian mixture models

have been used in many settings, and are particularly well known for their use as acoustic models in speech recognition (Bahl *et al.*, 1987).

3.11 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*:

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

The logistic sigmoid is commonly used to produce the ϕ parameter of a Bernoulli distribution because its range is $(0, 1)$, which lies within the valid range of values for the ϕ parameter. See Fig. 3.3 for a graph of the sigmoid function.

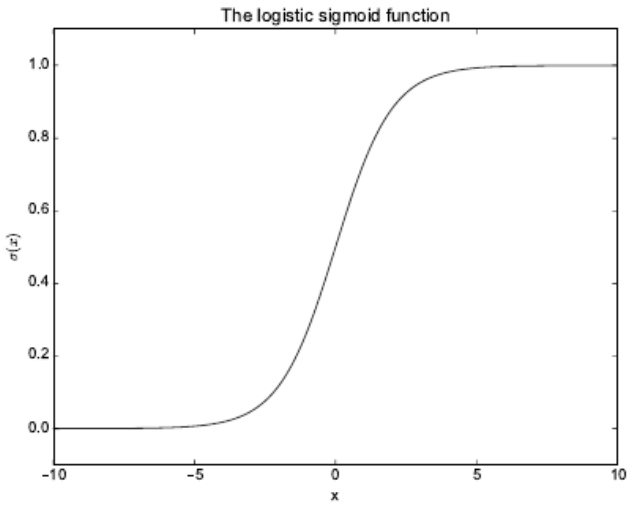


Figure 3.3: The logistic sigmoid function.

Another commonly encountered function is the *softplus* function (Dugas *et al.*, 2001):

$$\zeta(x) = \log\left(1 + \exp(x)\right).$$

The softplus function can be useful for producing the β or σ parameter of a normal distribution because its range is $\mathbb{R}^+$. It also arises commonly when manipulating expressions involving sigmoids, as it is the primitive of the sigmoid, i.e., the integral from $-\infty$ to x of the sigmoid. The name of the softplus function comes from the fact that it is a smoothed or "softened" version of

$$x^+ = \max(0, x).$$

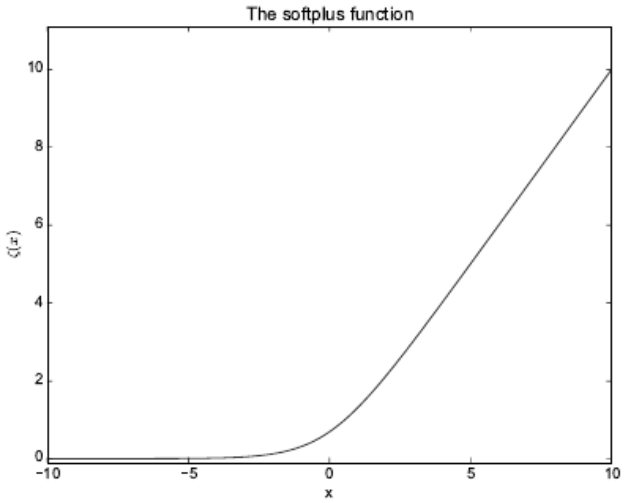


Figure 3.4: The softplus function.

See Fig. 3.4 for a graph of the softplus function.

The following properties are all useful enough that you may wish to memorize them:

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$1 - \sigma(x) = \sigma(-x)$$

$$\log \sigma(x) = -\zeta(-x)$$

$$\frac{d}{dx}\zeta(x) = \sigma(x)$$

$$\forall x \in (0, 1),\ \sigma^{-1}(x) = \log\left(\frac{x}{1 - x}\right)$$

$$\forall x > 0,\ \zeta^{-1}(x) = \log\left(\exp(x) - 1\right)$$

$$\zeta(x) - \zeta(-x) = x$$

The function $\sigma^{-1}(x)$ is called the *logit* in statistics, but this term is more rarely used in machine learning. The final property provides extra justification for the name "softplus", since $x^+ - x^- = x$.

3.12 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*:

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

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.

3.13 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 $\boldsymbol{x}$ lying in some set S is given by the integral of $p(\boldsymbol{x})$ over the set S. Some choices of set S can produce paradoxes. For example, it is possible to construct two sets S_1 and S_2 such that $P(S_1) + P(S_2) > 1$ but $S_1 \cap S_2 = \emptyset$. 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[3]. One of the key contributions of measure theory is to provide a characterization of the set of sets that we can compute the probability of without encountering paradoxes. In this book, we only integrate 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. However, it is useful to understand the intuition that a set of measure zero occupies no volume in the space we are measuring. For example, within $\mathbb{R}^2$,

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

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 of 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 only hold "almost everywhere" for continuous values.

One other detail we must be aware of relates to handling random variables that are deterministic functions of one another. Suppose we have two random variables, $\boldsymbol{x}$ and $\boldsymbol{y}$, such that $\boldsymbol{y} = g(\boldsymbol{x})$. You might think that $p_y(\boldsymbol{y}) = p_x(g^{-1}(\boldsymbol{y}))$. This is actually not the case.

Suppose $\boldsymbol{y} = \frac{\boldsymbol{x}}{2}$ and $\boldsymbol{x} \sim U(0, 1)$. If we use the rule $p_y(\boldsymbol{y}) = p_x(2\boldsymbol{y})$ then p_y will be 0 everywhere except the interval $[0, \frac{1}{2}]$, and it will be 1 on this interval. This means

$$\int p_y(\boldsymbol{y}) d\boldsymbol{y} = \frac{1}{2}$$

which violates the definition of a probability distribution.

This common mistake is wrong because it fails to account for the distortion of space introduced by the function $g(\boldsymbol{x})$. 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 or contract space, the infinitesimal volume surrounding $\boldsymbol{x}$ in $\boldsymbol{x}$ space may have different volume in $\boldsymbol{y}$ space. To correct the problem, we need to preserve the property

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

Solving from this, we obtain

$$p_y(\boldsymbol{y}) = p_x(g^{-1}(\boldsymbol{y}))|\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{y}}|$$

or equivalently

$$p_x(\boldsymbol{x}) = p_y(g(\boldsymbol{x}))|\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}}|. \tag{3.5}$$

In higher dimensions, the absolute value of the derivative generalizes to the determinant of the *Jacobian matrix* — the matrix with $J_{i,j} = \frac{\partial x_i}{\partial y_j}$.

3.14 Structured Probabilistic Models

Machine learning algorithms often involve probability distributions over a very large number of random variables. Often, these probability distributions involve

direct interactions between relatively few variables. Using a single function to describe the entire joint probability distribution can be very inefficient (both computationally and statistically).

Instead of using a single function to represent a probability distribution, we can split a probability distribution into many factors that we multiply together. For example, suppose we have three random variables, a, b, and c. Suppose that a influences the value of b and b influences the value of c, but that a and c are independent given b. We can represent the probability distribution over all three variables as a product of probability distributions over two variables:

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{a}).$$

These factorizations can greatly reduce the number of parameters needed to describe the distribution. Each factor uses a number of parameters that is exponential in the number of variables in the factor. This means that we can greatly reduce the cost of representing a distribution if we are able to find a factorization into distributions over fewer variables.

We can describe these kinds of factorizations using graphs. Here we use the word "graph" in the sense of graph theory, i.e. a set of vertices that may be connected to each other with edges. When we represent the factorization of a probability distribution with a graph, we call it a *structured probabilistic model* or *graphical model*.

There are two main kinds of structured probabilistic models: directed and undirected. Both kinds of graphical models use a graph in which each node in the graph corresponds to a random variable, and an edge connecting two random variables means that the probability distribution is able to represent direct interactions between those two random variables.

Directed models use graphs with directed edges, and they represent factorizations into conditional probability distributions, as in the example above. Specifically, a directed model contains one factor for every random variable x_i in the distribution, and that factor consists of the conditional distribution over x_i given the parents of x_i:

$$p(\mathbf{x}) = \prod_i p\left(\mathrm{x}_i \mid Pa_{\mathcal{G}}\left(\mathrm{x}_i\right)\right).$$

See Fig. 3.5 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

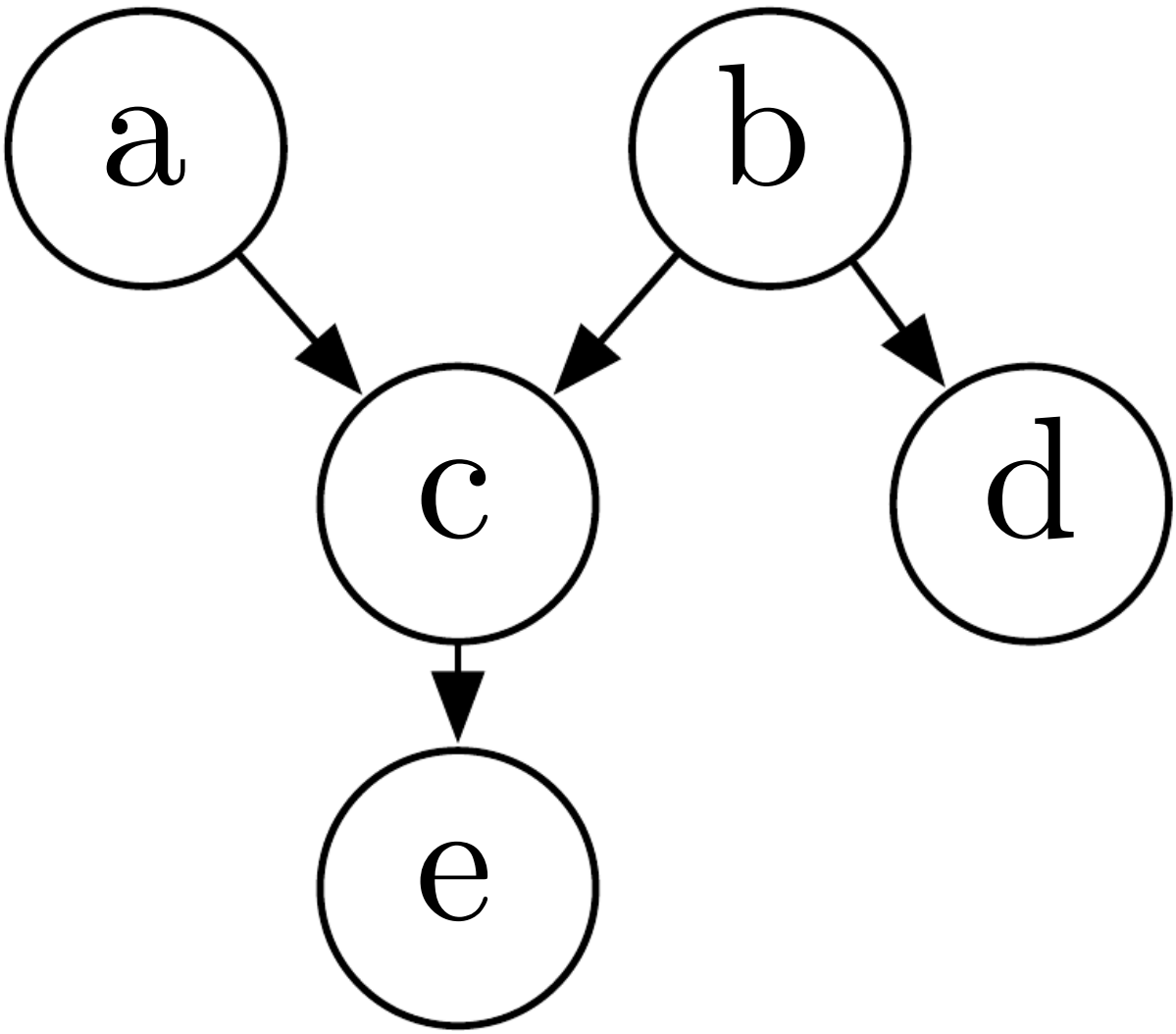


Figure 3.5: A directed graphical model over random variables a, b, c, d, and e. This graph corresponds to probability distributions that can be factored as $p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = p(\mathrm{a})p(\mathrm{b})p(\mathrm{c} \mid \mathrm{a}, \mathrm{b})p(\mathrm{d} \mid \mathrm{b})p(\mathrm{e} \mid \mathrm{c})$. This graph allows 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.

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 of 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:

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

See Fig. 3.6 for an example of an undirected graph and the factorization of probability distributions it represents.

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 part I and part II of this book, we will 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 will explore structured probabilistic models in much greater detail.

3.15 Example: Naive Bayes

We now know enough probability theory that we can perform some simple applications with a probabilistic model. In this example, we will show how to infer the probability that a patient has the flu using a simple probabilistic model. For now, we will assume that we just know the correct model somehow. Later chapters will cover the concepts needed to learn the model from data.

The *Naive Bayes* model is a simple probabilistic model that is often used to recognize patterns. The model consists of one random variable c representing a category, and a set of random variables $\mathbb{F} = \{\mathrm{f}^{(1)}, \ldots, \mathrm{f}^{(n)}\}$ representing features of objects in each category. In this example, we'll use Naive Bayes to diagnose patients as having the flu or not. The random variable c can thus have two values: c_0 representing the category of patients who do not have the flu, and c_1

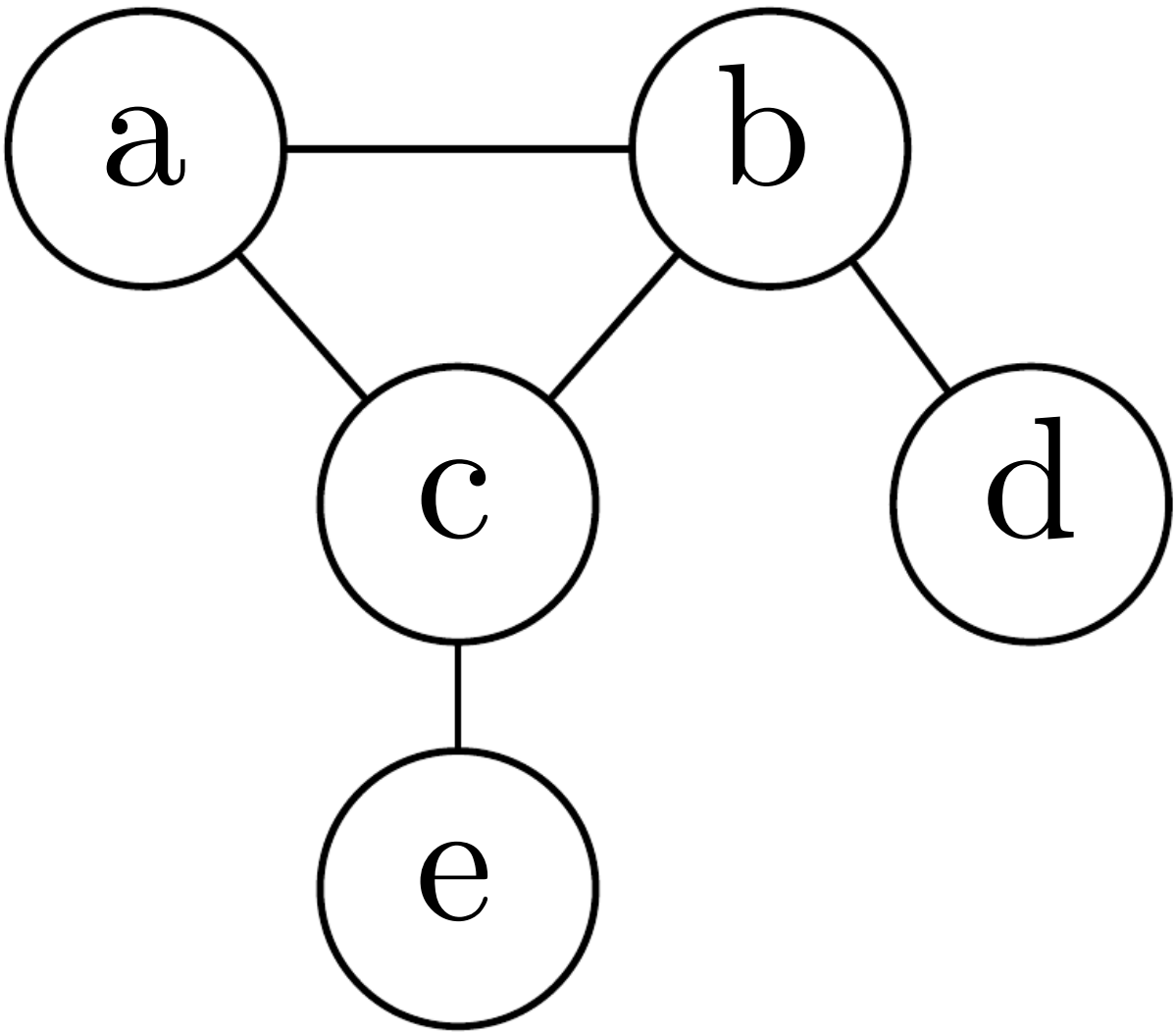


Figure 3.6: An undirected graphical model over random variables a, b, c, d, and e. This graph corresponds to probability distributions that can be factored as $p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = \frac{1}{Z}\phi^{(1)}(\mathrm{a}, \mathrm{b}, \mathrm{c})\phi^{(2)}(\mathrm{b}, \mathrm{d})\phi^{(3)}(\mathrm{c}, \mathrm{e})$. This graph allows 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.

representing the category of patients who do. Suppose $\mathrm{f}^{(1)}$ is the random variable representing whether the patient has a sore throat, with $f_0^{(1)}$ representing no sore throat, and $f_1^{(1)}$ representing a sore throat. Suppose $\mathrm{f}^{(2)} \in \mathbb{R}$ is the patient's temperature in degrees Celsius.

When using the Naive Bayes model, we assume that all of the features are independent from each other given the category:

$$P(\mathrm{c}, \mathrm{f}^{(1)}, \ldots, \mathrm{f}^{(n)}) = P(\mathrm{c}) \prod_i P(\mathrm{f}^{(i)} \mid \mathrm{c}).$$

See Fig. 3.7 for a directed graphical model that expresses these conditional independence assumptions. These assumptions are very strong and unlikely to be true in naturally occuring situations, hence the name "naive". Surprisingly, Naive Bayes often produces good predictions in practice (even though the assumptions do not hold precisely), and is a good baseline model to start with when tackling a new problem.

Beyond these conditional independence assumptions, the Naive Bayes framework does not specify anything about the probability distribution. The specific choice of distributions is left up to the designer. In our flu example, let's make $P(\mathrm{c})$ a Bernoulli distribution, with $P(\mathrm{c} = c_1) = \phi^{(\mathrm{c})}$. We can also make $P(\mathrm{f}^{(1)} \mid \mathrm{c})$ a Bernoulli distribution, with

$$P(\mathrm{f}^{(1)} = f_1^{(1)} \mid \mathrm{c} = c) = \phi_c^{\mathrm{f}}.$$

In other words, the Bernoulli parameter changes depending on the value of c. Finally, we need to choose the distribution over $\mathrm{f}^{(2)}$. Since $\mathrm{f}^{(2)}$ is real-valued, a normal distribution is a good choice. Because $\mathrm{f}^{(2)}$ is a temperature, there are hard limits to the values it can take on—it cannot go below $0K$, for example. Fortunately, these values are so far from the values measured in human patients that we can safely ignore these hard limits. Values outside the hard limits will receive extremely low probability under the normal distribution so long as the mean and variance are set correctly. As with $\mathrm{f}^{(1)}$, we need to use different parameters for different values of c, to represent that patients with the flu have different temperatures than patients without it:

$$\mathrm{f}^{(2)} \sim N(\mathrm{f}^{(2)} \mid \mu_c, \sigma_c^2).$$

Now we are ready to determine how likely a patient is to have the flu. To do this, we want to compute $P(\mathrm{c} \mid \mathbb{F})$, but we know $P(\mathrm{c})$ and $P(\mathbb{F} \mid \mathrm{c})$. This suggests that we should use Bayes' rule to determine the desired distribution. The word "Bayes" in the name "Naive Bayes" comes from this frequent use of Bayes' rule in conjunction with the model. We begin by applying Bayes' rule:

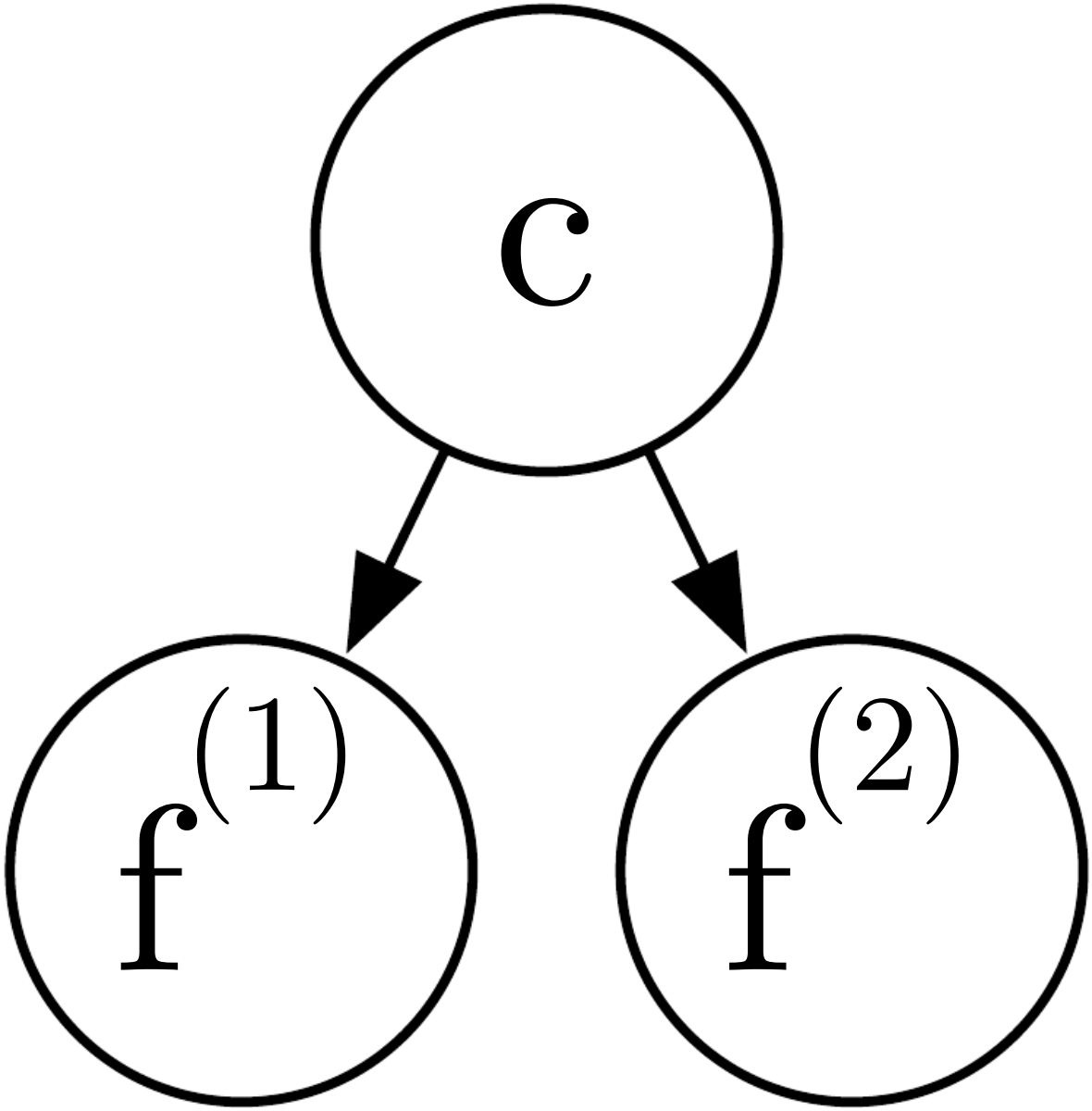


Figure 3.7: A directed graphical model depicting the conditional independence assumptions used by the Naive Bayes model.

$$P(\mathrm{c} \mid \mathbb{F}) = \frac{P(\mathrm{c})P(\mathbb{F} \mid \mathrm{c})}{P(\mathbb{F})}. \tag{3.6}$$

We do not know $P(\mathbb{F})$. Fortunately, it is easy to compute:

$$P(\mathbb{F}) = \sum_{c \in \mathrm{c}} P(\mathrm{c} = c, \mathbb{F}) \text{ (by the sum rule)}$$

$$= \sum_{c \in \mathrm{c}} P(\mathrm{c} = c) P(\mathbb{F} \mid \mathrm{c} = c) \text{ (by the chain rule).}$$

Substituting this result back into equation 3.6, we obtain

$$P(\mathrm{c} \mid \mathbb{F}) = \frac{P(\mathrm{c})P(\mathbb{F} \mid \mathrm{c})}{\sum_{c \in \mathrm{c}} P(\mathrm{c} = c) P(\mathbb{F} \mid \mathrm{c} = c)}$$

$$= \frac{P(\mathrm{c}) \Pi_i P(\mathrm{f}^{(i)} \mid \mathrm{c})}{\sum_{c \in \mathrm{c}} P(\mathrm{c} = c) \Pi_i P(\mathrm{f}^{(i)} \mid \mathrm{c} = c)}$$

by the Naive Bayes assumptions. This is as far as we can simplify the expression for a general Naive Bayes model.

We can simplify the expression further by substituting in the definitions of the particular probability distributions we have defined for our flu diagnosis example:

$$P(\mathrm{c} = c \mid \mathrm{f}^{(1)} = f_1, \mathrm{f}^{(2)} = f_2) = \frac{g(c)}{\sum_{c' \in \mathrm{c}} g(c')}$$

where

$$g(c) = P(\mathrm{c} = c) P(\mathrm{f}^{(1)} = f^{(1)} \mid \mathrm{c} = c) P(\mathrm{f}^{(2)} = f^{(2)} \mid \mathrm{c} = c).$$

Since c only has two possible values in our example, we can simplify this to:

$$P(\mathrm{c} = 1 \mid \mathrm{f}^{(1)} = f_1, \mathrm{f}^{(2)} = f_2) = \frac{g(1)}{g(0) + g(1)}$$

$$= \frac{1}{1 + \frac{g(0)}{g(1)}}$$

$$= \frac{1}{1 + \exp\left(\log g(0) - \log g(1)\right)}$$

$$= \sigma\left(\log g(1) - \log g(0)\right). \tag{3.7}$$

To go further, let's simplify $\log g(i)$:

$$\log g(i) = \log \quad \phi^{(\mathrm{c})i} (1 - \phi^{(\mathrm{c})})^{1-i} \phi_1^{(\mathrm{f})f_1} (1 - \phi_1^{(\mathrm{f})})^{1-f_1} \quad \frac{1}{2\pi\sigma_i^2} \exp \quad -\frac{1}{2\sigma_i^2} (f_2 - \mu_i)^2$$

$$\left(\qquad \qquad \sqrt{\qquad \qquad \left(\qquad \qquad \right)} \right)$$

$$= i \log \phi^{(\mathrm{c})} + (1-i) \log\Big(1 - \phi^{(\mathrm{c})}\Big) + f_1 \log \phi_i^{(\mathrm{f})} + (1-f_1) \log(1-\phi_i^{(\mathrm{f})}) + \frac{1}{2} \log \frac{1}{2\pi\sigma_i^2} - \frac{1}{2\sigma_i^2} (f_2 - \mu_i)^2 .$$

Substituting this back into equation 3.7, we obtain

$$P(\mathrm{c} = c \mid \mathrm{f}^{(1)} = f_1, \mathrm{f}^{(2)} = f_2) =$$

$$\sigma \Bigg(\log \phi^{(\mathrm{c})} - \log(1 - \phi^{(\mathrm{c})}) + f_1 \log \phi_1^{(\mathrm{f})} + (1 - f_1) \log(1 - \phi_1^{(\mathrm{f})}) - f_1 \log \phi_0^{(\mathrm{f})} + (1 - f_1) \log(1 - \phi_0^{(\mathrm{f})}) - \frac{1}{2} \log 2\pi\sigma_1^2 + \frac{1}{2} \log 2\pi\sigma_0^2 - \frac{1}{2\sigma_1^2} (f_2 - \mu_1)^2 + \frac{1}{2\sigma_0^2} (f_2 - \mu_0)^2 \Bigg) .$$

From this formula, we can read off various intuitive properties of the Naive Bayes classifier's behavior on this example problem, regarding the *inference* that can be drawn from a trained model. The probability of the patient having the flu grows like a sigmoidal curve. We move farther to the left as f_2, the patient's temperature, moves farther away from μ_1, the average temperature of a flu patient.

Chapter 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 iteratively update estimates of the solution, rather than analytically deriving a formula providing a symbolic expression for the correct solution. Common operations include solving systems of linear equations and finding the value of an argument that minimizes a function. 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, otherwise 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 further arithmetic).

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 this infinite values into not-a-number values.

For an example of the need to design software implementations to deal with overflow and underflow, consider the softmax function, typically used to predict the probabilities associated with a Multinoulli distribution:

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_j^n \exp(x_j)}.$$

Consider what happens when all of the x_i are equal to some constant c. Analytically, we can see that all of 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 $\text{softmax}(\boldsymbol{z})$ 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 $\log \text{softmax}(\boldsymbol{x})$ 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 $\log \text{softmax}$ in a numerically stable way. The $\log \text{softmax}$ 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 of the numerical considerations involved in implementing the various algorithms described in this book. Implementors should keep numerical issues in mind when developing implementations. Many numerical issues can be avoided by using Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012), 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 per-

turbed 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

$$\max_{i,j} |\frac{\lambda_i}{\lambda_j}|,$$

i.e. 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.

Note that this 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. With iterative algorithms such as solving a linear system (or the worked-out example of linear least square by gradient descent, Section 4.5) ill-conditioning (in that case of the linear system matrix) yields very slow convergence of the iterative algorithm, i.e., more iterations are needed to achieve some given degree of approximation to the final solution.

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*. When we are minimizing it, we may also call it the *cost function*, *loss function*, or *error function*.

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

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

Suppose we have a function $y = f(x)$, where both x and y are real numbers. The *derivative* of this function is denoted as $f'(x)$ or as $\frac{dy}{dx}$. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x. In other words, it specifies how to scale a small change in the input in order to obtain the corresponding change in the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y. For example, we

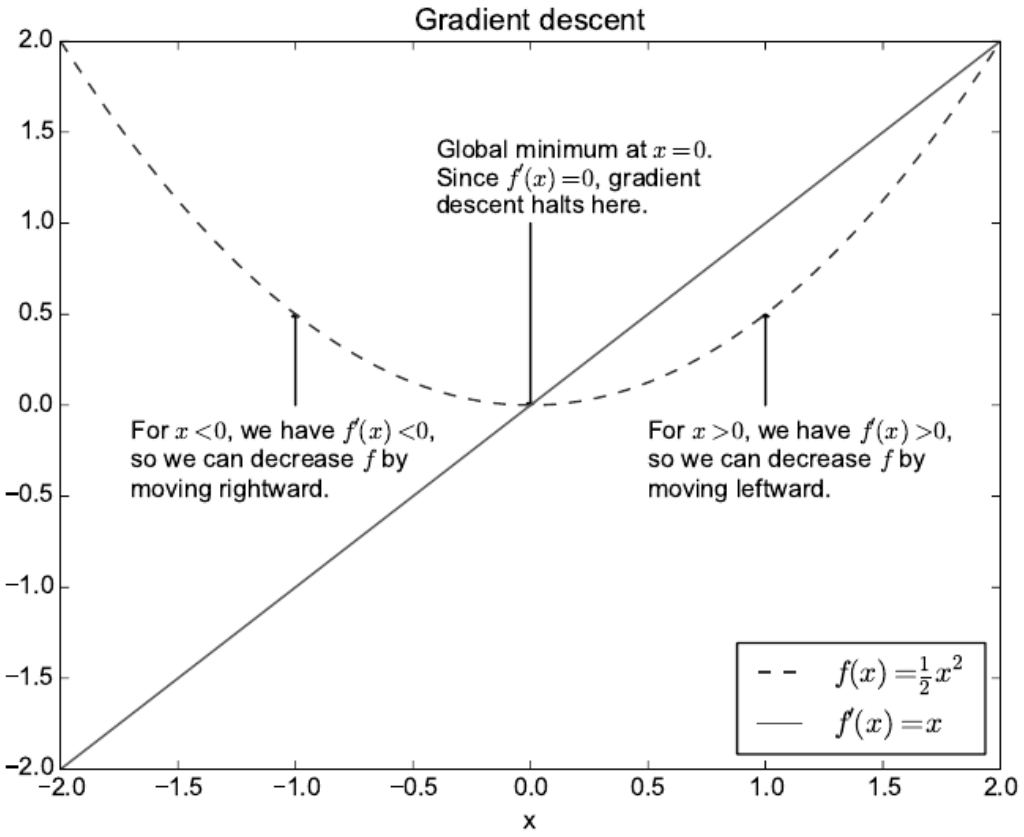


Figure 4.1: An illustration of how the derivatives of a function can be used to follow the function downhill to a minimum. This technique is called *gradient descent*.

know that $f(x - \epsilon \operatorname{sign}(f'(x)))$ is less than $f(x)$ for small enough ϵ. We can thus reduce $f(x)$ by moving x in small steps with opposite sign of the the derivative. This technique is called gradient descent (Cauchy, 1847a). See Fig. 4.1 for an example of this technique.

When $f'(x) = 0$, the derivative provides no information about which direction to move. Points where $f'(x) = 0$ are known as *critical points* or *stationary points*. A *local minimum* is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A *local maximum* is a point where $f(x)$ is higher than at all neighboring points, so it is not possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither maxima nor minima. These are known as *saddle points*. See Fig. 4.2 for examples of each type of critical point.

A point that obtains the absolute lowest value of $f(x)$ is a *global minimum*. It is possible for there to be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally

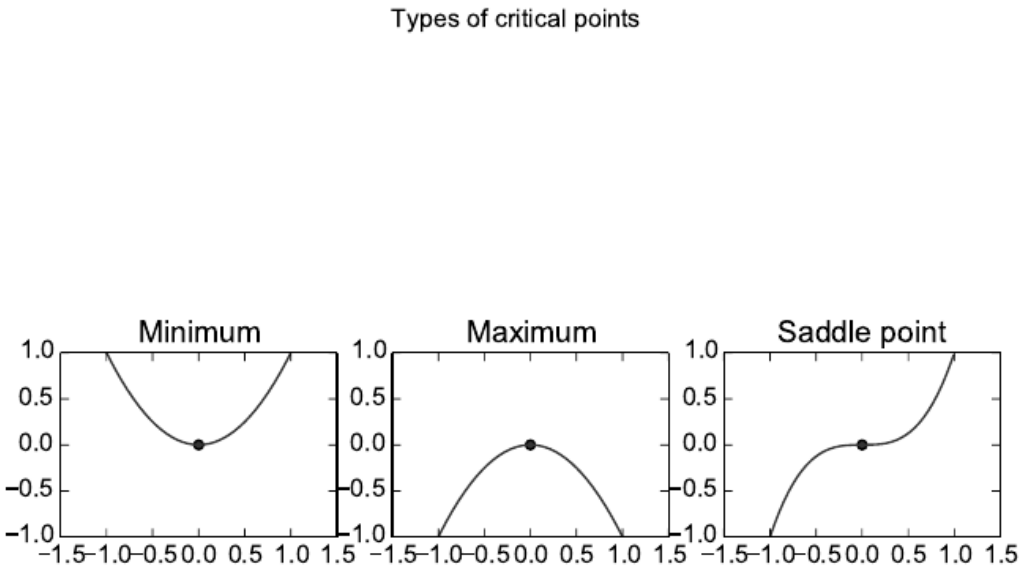


Figure 4.2: Examples of each of the three types of critical points in 1-D. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points, a local maximum, which is higher than the neighboring points, or a saddle point, which has neighbors that are both higher and lower than the point itself. The situation in higher dimension is qualitatively different, especially for saddle points: see Figures 4.4 and 4.5.

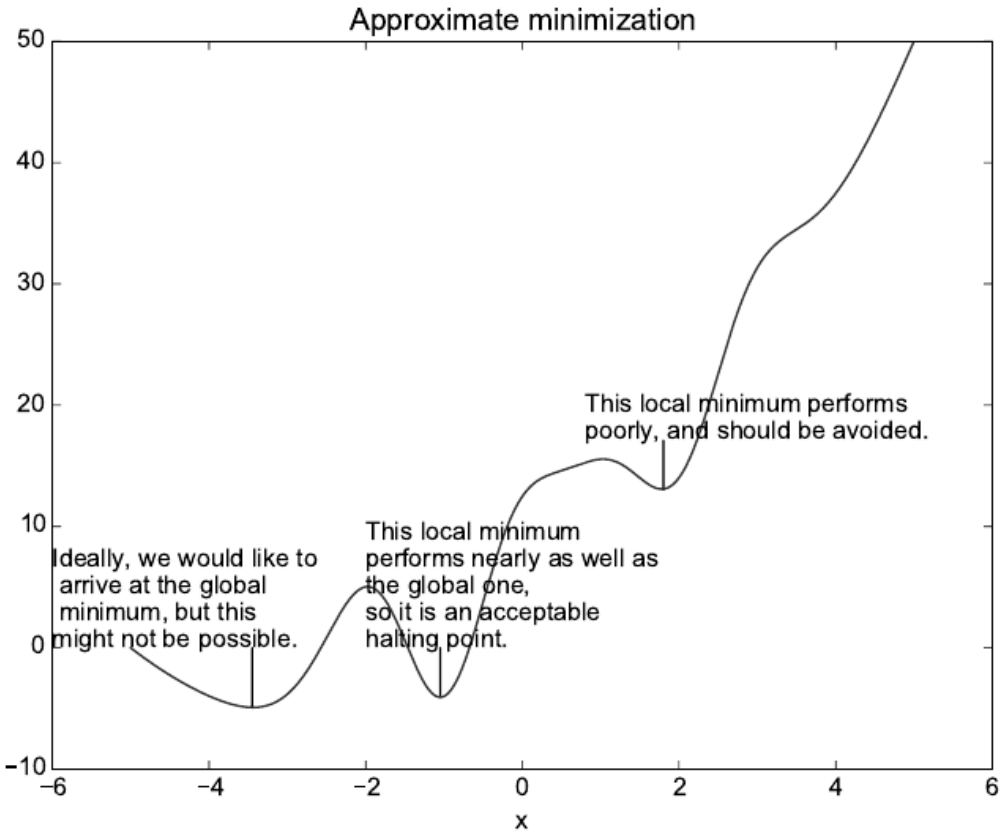


Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function. Optimizing MLPs was believed to suffer from the presence of many local minima, but this idea is questioned in recent work (Dauphin *et al.*, 2014), with saddle points being considered as the more serious issue.

optimal. In the context of deep learning, we optimize functions that may have many local minima that are not optimal, and many saddle points surrounded by very flat regions. All of this makes optimization very difficult, especially when the input to the function is multidimensional. We therefore usually settle for finding a value of f that is very low, but not necessarily minimal in any formal sense. See Fig. 4.3 for an example. The figure caption also raises the question of whether local minima or saddle points and plateaus are more to blame for the difficulties one may encounter in training deep networks, a question that is discussed further in Chapter 8, in particular Section 8.2.3.

We often minimize functions that have multiple inputs: $f : \mathbb{R}^n \to \mathbb{R}$. Note that for the concept of "minimization" to make sense, there must still be only

one output.

For these functions, 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: f is the vector containing all of 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 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 this is $\boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

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

$$\begin{aligned} &\min_{\boldsymbol{u}, \boldsymbol{u}^\top \boldsymbol{u}=1} \boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &= \min_{\boldsymbol{u}, \boldsymbol{u}^\top \boldsymbol{u}=1} ||\boldsymbol{u}||_2 ||\nabla_{\boldsymbol{x}} f(\boldsymbol{x})||_2 \cos\theta \end{aligned}$$

where θ is the angle between $\boldsymbol{u}$ and the gradient. Substituting in $||\boldsymbol{u}||_2 = 1$ and ignoring factors that don't 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

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

where ϵ is 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 $\boldsymbol{x}$.

Sometimes we need to find all of the partial derivatives of all of the elements of a vector-valued function. 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 means it can be useful for determining whether a critical point is a local maximum, a local minimum, or saddle point. Recall that on a critical point, $f'(x) = 0$. When $f''(x) > 0$, this means that $f'(x)$ increases as we move to the right, and $f'(x)$ 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 of the second derivatives of the function. 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

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

Equivalently, the Hessian is the Jacobian of the gradient.

Anywhere that the second partial derivatives are continuous, the differential operators are commutative, i.e. their order can be swapped:

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

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.

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[1], the point is a local minimum. This can be seen by observing

[1] all its eigenvalues are positive

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[2], 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 Fig. 4.4 for an example. Finally, the multidimensional second derivative test can be inconclusive, just like the univariate version. The test is inconclusive whenever all of the non-zero 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.

The Hessian can also be useful for understanding the performance of gradient descent. When the Hessian has a poor condition number, gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change in the derivative so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer. See Fig. 4.5 for an example.

This issue can be resolved by using information from the Hessian matrix to guide the search. The simplest method for doing so is known as *Newton's method*. Newton's method is based on using a second-order *Taylor series expansion* to approximate $f(\boldsymbol{x})$ near some point $\boldsymbol{x}_0$, ignoring derivatives of higher order:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + (\boldsymbol{x} - \boldsymbol{x}_0)^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^\top H(f)(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0).$$

If we then solve for the critical point of this function, we obtain:

$$\boldsymbol{x}^* = \boldsymbol{x}_0 - H(f)(\boldsymbol{x}_0)^{-1} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_0).$$

When the function can be locally approximated as quadratic, iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would. This is a useful property near a local minimum, but it can be a harmful property near a saddle point. As discussed in Section 8.2.3, Newton's method is only appropriate when the nearby critical point is a minimum (all the eigenvalues of the Hessian are positive), whereas gradient descent can in principle escape a saddle point, although it may take a lot of time if the negative eigenvalues are very small in magnitude, producing a kind of plateau around the saddle point.

Optimization algorithms such as gradient descent that use only the gradient are called *first-order optimization algorithms*. Optimization algorithms such as

[2] all its eigenvalues are negative

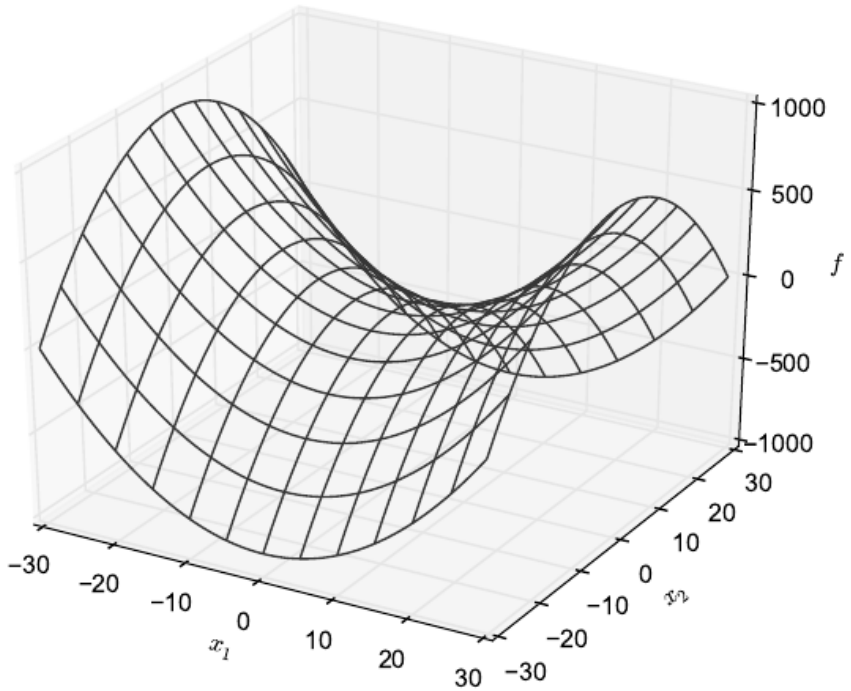


Figure 4.4: A saddle point containing both positive and negative curvature. The function in this example is $f(\boldsymbol{x}) = x_1^2 - x_2^2$. Along the axis corresponding to x_1, the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding 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. Note that in more than one dimension, it is not necessary to have an eigenvalue of 0 in order to get a saddle point: it is only necessary to have both positive and negative eigenvalues. See Section 8.2.3 for a longer discussion of saddle points in deep nets.

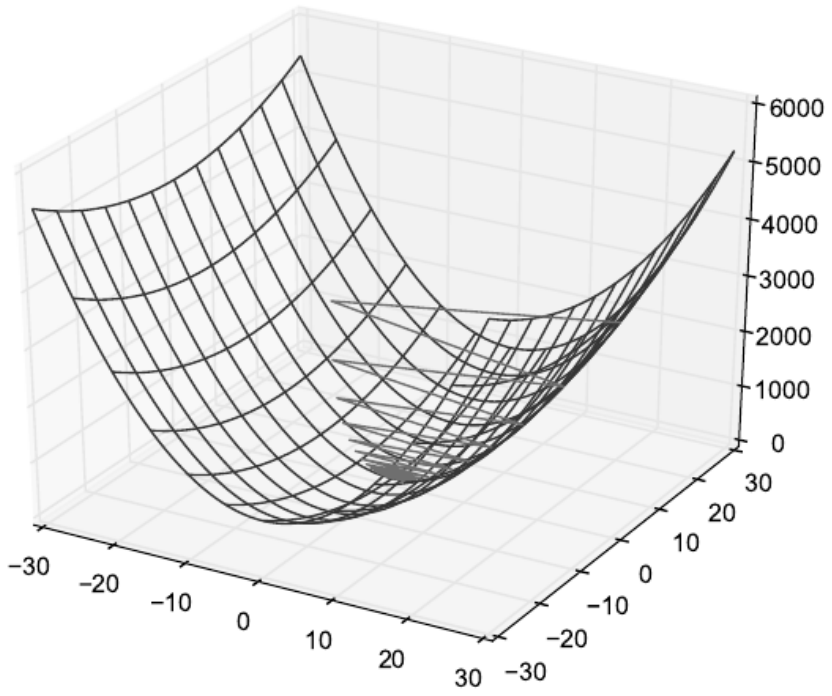


Figure 4.5: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent on a quadratic function whose Hessian matrix has condition number 5 (curvature is 5 times larger in one direction than in some other direction). The lines above the mesh indicate the path followed by gradient descent. This very elongated quadratic function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature. Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration. The large positive eigenvalue of the Hessian corresponding to the eigenvector pointed in this direction indicates that this directional derivative is rapidly increasing, so an optimization algorithm based on the Hessian could predict that the steepest direction is not actually a promising search direction in this context. Note that some recent results suggest that the above picture is not representative for deep highly non-linear networks. See Section 8.2.4 for more on this subject.

Newton's method that also use the Hessian matrix are called *second-order optimization algorithms* (Nocedal and Wright, 2006).

The optimization algorithms employed in most contexts in this book are applicable to a wide variety of functions, but come with almost no guarantees. This is because the family of functions used in deep learning is quite complicated. In many other fields, the dominant approach to optimization is to design optimization algorithms for a limited family of functions. Perhaps the most successful field of specialized optimization is *convex optimization*. Convex optimization algorithms are able to provide many more guarantees, but are applicable only to functions for which the Hessian is positive definite everywhere. Such functions are well-behaved because they lack saddle points and all of 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. However, 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.

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 S. If we use a line search (see previous section), 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 $g(\theta) = f([\cos\theta, \sin\theta]^T)$ 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* [3] 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 S in terms of equations and inequalities. We want a description of S in terms of m functions g_i and n functions h_j so that $S = \{\boldsymbol{x} \mid \forall i, g_i(\boldsymbol{x}) = 0 \text{ and } \forall j, h_j(\boldsymbol{x}) \leq 0\}$. 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

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) + \sum_j \alpha_j h_j(\boldsymbol{x}).$$

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

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

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

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

This follows because any time the constraints are satisfied,

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

while any time a constraint is violated,

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

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

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

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} -f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) + \sum_j \alpha_j h_j(\boldsymbol{x}).$$

[3] The KKT approach generalizes the method of *Lagrange multipliers* which only allows equality constraints

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

$$\max_{\boldsymbol{x}} \min_{\boldsymbol{\lambda}} \min_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) - \sum_j \alpha_j h_j(\boldsymbol{x}).$$

Note that 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 is the same whether or not that constraint exists. 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, $\boldsymbol{\alpha} \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$. In other words, for all i, we know that at least one of the constraints $\alpha_i \geq 0$ and $h_i(\boldsymbol{x}) \leq 0$ 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.

The properties that the gradient of the generalized Lagrangian is zero, all constraints on both $\boldsymbol{x}$ and the KKT multipliers are satisfied, and $\boldsymbol{\alpha} \odot \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$ are called the *Karush-Kuhn-Tucker* (KKT) conditions (Karush, 1939; Kuhn and Tucker, 1951). Together, these properties describe the optimal points of constrained optimization problems.

In the case where there are no inequality constraints, the KKT approach simplifies to the method of Lagrange multipliers. 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

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

There are specialized linear algebra algorithms that 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:

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

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.

Set ϵ, the step size, and δ, the tolerance, 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 $\boldsymbol{x}^\top \boldsymbol{x} \leq 1$. To do so, we introduce the Lagrangian

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

We can now solve the problem

$$\min_{\boldsymbol{x}} \max_{\lambda, \lambda \geq 0} L(\boldsymbol{x}, \lambda).$$

The solution to the unconstrained least squares problem is given by $\boldsymbol{x} = \boldsymbol{A}^+ \boldsymbol{b}$. 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

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

This tells us that the solution will take the form

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

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

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

When the norm of $\boldsymbol{x}$ exceeds 1, this derivative is positive, so to ascend the gradient and increase the Lagrangian with respect to λ, we increase λ. This will in turn shrink the optimal $\boldsymbol{x}$. The process continues until $\boldsymbol{x}$ has the correct norm and the derivative on λ is 0.

Chapter 5

Machine Learning Basics

Deep learning is a specific kind of machine learning. In order to understand deep learning well, one must have a solid understanding of the basic principles of machine learning. This chapter provides a brief course in the most important general principles that will be applied throughout the rest of the book. Novice readers or those that want a wider perspective are encouraged to consider machine learning textbooks with a more comprehensive coverage of the fundamentals, such as Murphy (2012) or Bishop (2006). If you are already familiar with machine learning basics, feel free to skip ahead to Section 5.13. That section covers some perspectives on traditional machine learning techniques that have strongly influenced the development of deep learning algorithms.

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? A popular definition of learning in the context of computer programs is "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" (Mitchell, 1997). One can imagine a very wide variety of experiences E, tasks T, and performance measures P, and we do not make any attempt in this book to provide a formal definition of what may be used for each of these entities. Instead, the following sections 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 is mostly interesting because of the tasks we can accomplish with it. From an engineering point of view, machine learning allows 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 understanding it allows us to understand the principles that underlie intelligent behavior, and intelligent behavior is defined as being able to accomplish certain tasks.

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.

Many kinds of tasks can be solved with machine learning. Some of the most common machine learning tasks include the following:

- *Classification*: In this type of task, the computer program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \to \{1, \ldots, k\}$ which may then be applied to any input. Here the output of $f(\boldsymbol{x})$ can be interpreted as an estimate of the category that $\boldsymbol{x}$ belongs to. There are other variants of the classification task, for example, where f outputs a probability distribution over classes. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image. For example, the Willow Garage PR2 robot is able to act as a waiter that can recognize different kinds of drinks and deliver them to people on command (Goodfellow *et al.*, 2010). Modern object recognition is best accomplished with deep learning (Krizhevsky *et al.*, 2012a; Ioffe and Szegedy, 2015). Object recognition is the same basic technology that allows computers to recognize faces (Taigman *et al.*, 2014), which can be used to automatically tag people in photo collections and allow computers to interact more naturally with their users.

- *Classification with missing inputs* : Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. In order to solve the classification task, the learning algorithm only has to define a *single* function mapping from a vector input to a categorical output. When some of the inputs may be missing, rather than providing a single classification function, the learning algorithm must learn a *set* of functions. Each function corresponds

to classifying $\boldsymbol{x}$ with a different subset of its inputs missing. This kind of situation arises frequently in medical diagnosis, because many kinds of medical tests are expensive or invasive. One way to efficiently define such a large set of functions is to learn a probability distribution over all of the relevant variables, then solve the classification task by marginalizing out the missing variables. With n input variables, we can now obtain all 2^n different classification functions needed for each possible set of missing inputs, but we only need to learn a single function describing the joint probability distribution. See Goodfellow *et al.* (2013b) for an example of a deep probabilistic model applied to such a task in this way. Many of the other tasks described in this section can also be generalized to work with missing inputs; classification with missing inputs is just one example of what machine learning can do.

- *Regression* : In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \to \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premia), or the prediction of future prices of securities. These kinds of predictions are also used for algorithmic trading.

- *Transcription* : In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form. For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text a sequence of characters (e.g. in ASCII or Unicode format). Google Street View uses deep learning to process address numbers in this way Goodfellow *et al.* (2014d). Another example is speech recognition, where the computer program is provided an audio waveform and emits a sequence of characters or word ID codes describing the words that were spoken in the audio recording. Deep learning is a crucial component of modern speech recognition systems used at major companies including Microsoft, IBM and Google (Hinton *et al.*, 2012b).

- indexTranslation *Translation*: In a translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. This is commonly applied to natural languages, such as to translate from English to French. Deep learning has recently begun to have an important impact on this kind of task (Sutskever *et al.*, 2014a; Bahdanau *et al.*, 2014).

- *Structured output* tasks involve any task where the output is a vector containing important relationships between the different elements. This is a broad category, and includes the transcription and translation tasks described above, but also many other tasks. One example is parsing—mapping a natural language sentence into a tree that describes its grammatical structure and the relative role of its constituents. See Collobert (2011) for an example of deep learning applied to a parsing task. Another example is pixel-wise segmentation of images, where the computer program assigns every pixel in an image to a specific category. For example, deep learning can be used to annotate the locations of roads in aerial photographs (Mnih and Hinton, 2010). The output need not have its form mirror the structure of the input as closely as in these annotation-style tasks. For example, in an image captioning, the computer program observes an image and outputs a natural language sentence describing the image (Kiros *et al.*, 2014a,b; Mao *et al.*, 2015; Vinyals *et al.*, 2015; Donahue *et al.*, 2014; Karpathy and Li, 2015; Fang *et al.*, 2015; Xu *et al.*, 2015a). These tasks are called structured output tasks because the program must output several values that are all tightly inter-related. For example, the words produced by an image captioning program must form a valid sentence.

- *Anomaly detection*: In this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection. By modeling your purchasing habits, a credit card company can detect misuse of your cards. If a thief steals your credit card or credit card information, the thief's purchases will often come from a different probability distribution over purchase types than your own. The credit card company can prevent fraud by placing a hold on an account as soon as that card has been used for an uncharacteristic purchase.

- *Synthesis and sampling*: In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. This can be useful for media applications where it can be expensive or boring for an artist to generate large volumes of content by hand. For example, video games can automatically generate textures for large objects or landscapes, rather than requiring an artist to manually label each pixel (Luo *et al.*, 2013). In some cases, we want the sampling or synthesis procedure to generate some specific kind of output given the input. For example, in a speech synthesis task, we provide a written sentence and ask the program to emit an audio waveform containing a spoken version of that sentence. This is a kind of structured output task, but with the added qualification

that there is no single correct output for each input, and we explicitly desire a large amount of variation in the output, in order for the output to seem more natural and realistic.

- *Imputation of missing values*: In this type of task, the machine learning algorithm is given a new example $\boldsymbol{x} \in \mathbb{R}^n$, but with some entries x_i of $\boldsymbol{x}$ missing. The algorithm must provide a prediction of the values of the missing entries.

- *Denoising*: In this type of task, the machine learning algorithm is given in input a *corrupted example* $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$ obtained by an unknown corruption process from a *clean example* $\boldsymbol{x} \in \mathbb{R}^n$. The learner must predict the clean example $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$, or more generally predict the conditional probability distribution $P(\boldsymbol{x} \mid \tilde{\boldsymbol{x}})$.

- *Density or probability function estimation*: In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{\text{model}}(\boldsymbol{x})$ can be interpreted as a probability density function (if $\mathbf{x}$ is continuous) or a probability function (if $\mathbf{x}$ is discrete) on the space that the examples were drawn from. To do such a task well (we will specify exactly what that means when we discuss performance measures P), the algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur. Most of the tasks described above require that the learning algorithm has at least implicitly captured the structure of the probability distribution. Density estimation allows us to explicitly capture that distribution. In principle, we can then perform computations on that distribution in order to solve the other tasks as well. For example, if we have performed density estimation to obtain a probability distribution $p(\boldsymbol{x})$, we can use that distribution to solve the missing value imputation task. If a value x_i is missing, then we know the distribution over it is given by $p(x_i \mid x_{-i})$. In practice, density estimation does not always allow us to solve all of these related tasks, because in many cases the required operations on $p(\boldsymbol{x})$ are computationally intractable.

Of course, many other tasks and types of tasks are possible. The types of tasks we listed here are only intended to provide examples of what machine learning can do, not to define a rigid taxonomy of tasks.

5.1.2 The Performance Measure, P

In order to evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task T being carried out by the system.

For tasks such as classification, classification with missing inputs, and transcription, we often measure the *accuracy* of the model. In the simplest case this is just the proportion of examples for which the model produces the correct output. For tasks such as density estimation, we can measure the probability the model assigns to some examples.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a *test set* of data that is separate from the data used for training the machine learning system.

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system.

In some cases, this is because it is difficult to decide what should be measured. For example, when performing a transcription task, should we measure the accuracy of the system at transcribing entire sequences, or should we use a more fine-grained performance measure that gives partial credit for getting some elements of the sequence correct? When performing a regression task, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes? These kinds of design choices depend on the application.

In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. For example, this arises frequently in the context of density estimation. Many of the best probabilistic models represent probability distributions only implicitly. Computing the actual probability value assigned to a specific point in space is intractable. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion.

5.1.3 The Experience, E

Machine learning algorithms can be broadly categorized as *unsupervised* or *supervised* by what kind of experience they are allowed to have during the learning process.

Most of the learning algorithms in this book can be understood as being allowed to experience an entire *dataset*. A dataset is a collection of many objects called *examples*, with each example containing many *features* that have been objectively measured. Sometimes we will also call examples *data points*.

One of the oldest datasets studied by statisticians and machine learning researchers is the Iris dataset (Fisher, 1936). It is a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one exam-

ple. The features within each example are the measurements of each of the parts of the plant: the sepal length, sepal width, petal length, and petal width. The dataset also records which species each plant belonged to. Three different species are represented in the dataset.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like dividing the dataset into clusters of similar examples.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a *label* or *target*. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.

Roughly speaking, unsupervised learning involves observing several examples of a random vector $\mathbf{x}$, and attempting to implicitly or explicitly learn the probability distribution $p(\mathbf{x})$, or some interesting properties of that distribution, while supervised learning involves observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, and learning to predict $\mathbf{y}$ from $\mathbf{x}$, e.g. estimating $p(\mathbf{y} \mid \mathbf{x})$. The term *supervised learning* originates from the view of the target $\mathbf{y}$ being provided by an instructor or teacher that shows the machine learning system what to do. In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

Unsupervised learning and supervised learning are not formally defined terms, and the lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $\mathbf{x} \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(\mathbf{x}) = \Pi_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \ldots, \mathrm{x}_{i-1}).$$

This decomposition means that we can solve the ostensibly unsupervised problem of modeling $p(\mathbf{x})$ by splitting it into n supervised learning problems. Alternatively, we can solve the supervised learning problem of learning $p(y \mid \mathbf{x})$ by using traditional unsupervised learning technologies to learn the joint distribution $p(\mathbf{x}, y)$ and inferring

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}.$$

Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help to roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression,

classification, and structured output problems as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

Some machine learning algorithms do not just experience a fixed dataset. For example, *reinforcement learning* algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences. Such algorithms are beyond the scope of this book.

Most machine learning algorithms simply experience a dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples. Each example is a collection of observations called *features* collected from a different time or place. If we wish to make a system for recognizing objects from photographs, we might use a machine learning algorithm where each example is a photograph, and the features within the example are the brightness values of each of the pixels within the photograph. If we wish to perform speech recognition, we might collect a dataset where each example is a recording of a person saying a word or sentence, and each of the features is the amplitude of the sound wave at a particular moment in time.

One common way of describing a dataset is with a *design matrix*. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\boldsymbol{X} \in \mathbb{R}^{150\times 4}$, where $X_{i,1}$ is the sepal length of plant i, $X_{i,2}$ is the sepal width of plant i, etc. We will describe most of the learning algorithms in this book in terms of how they operate on design matrix datasets.

Of course, to describe a dataset as a design matrix, it must be possible to describe each example as a vector, and each of these vectors must be the same size. This is not always possible. For example, if you have a collection of photographs with different widths and heights, then different photographs will contain different numbers of pixels, so not all of the photographs may be described with the same length of vector. Different sections of this book describe how to handle different types of heterogeneous data. In cases like these, rather than describing the dataset as a matrix with m rows, we will describe it as a set containing m elements, e.g. $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$. This notation does not imply that any two example vectors $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(j)}$ have the same size.

In the case of supervised learning, the example contains a label or target as well as a collection of features. For example, if we want to use a learning algorithm to perform object recognition from photographs, we need to specify which object appears in each of the photos. We might do this with a numeric code, with 0 signifying a person, 1 signifying a car, 2 signifying a cat, etc. Often when working with a dataset containing a design matrix of feature observations $\boldsymbol{X}$, we also

provide a vector of labels $\boldsymbol{y}$, with y_i providing the label for example i.

Of course, sometimes the label may be more than just a single number. For example, if we want to train a speech recognition system to transcribe entire sentences, then the label for each example sentence is a sequence of words.

Just as there is no formal definition of supervised and unsupervised learning, there is no rigid taxonomy of datasets or experiences. The structures described here cover most cases, but it is always possible to design new ones for new applications.

5.2 Example: Linear Regression

In the previous section, we saw that a machine learning algorithm is an algorithm that is capable of improving a computer program's performance at some task via experience. Now it is time to define some specific machine learning algorithms.

Let's begin with an example of a simple machine learning algorithm: *linear regression*. In this section, we will only describe what the linear regression algorithm does. We wait until later sections of this chapter to justify the algorithm and show more formally that it actually works.

As the name implies, linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $\boldsymbol{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. In the case of linear regression, the output is a linear function of the input. Let $\hat{y}$ be the value that our model predicts y should take on. We define the output to be

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a vector of *parameters*.

Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features. We can think of $\boldsymbol{w}$ as a set of *weights* that determine how each feature affects the prediction. If a feature x_i receives a positive weight w_i, then increasing the value of that feature increases the value of our prediction $\hat{y}$. If a feature receives a negative weight, then increasing the value of that feature decreases the value of our prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction.

We thus have a definition of our task T: to predict y from $\boldsymbol{x}$ by outputting $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$. Next we need a definition of our performance measure, P.

Let's suppose that we have a design matrix of m example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of y for each of

these examples. Because this dataset will only be used for evaluation, we call it the *test set*. Let's refer to the design matrix of inputs as $\boldsymbol{X}^{(\text{test})}$ and the vector of regression targets as $\boldsymbol{y}^{(\text{test})}$.

One way of measuring the performance of the model is to compute the *mean squared error* of the model on the test set. If $\hat{\boldsymbol{y}}^{(\text{test})}$ is the predictions of the model on the test set, then the mean squared error is given by

$$\text{MSE}_{\text{test}} = \frac{1}{m}\sum_i (\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})})_i^2.$$

Intuitively, one can see that this error measure decreases to 0 when $\hat{\boldsymbol{y}}^{(\text{test})} = \boldsymbol{y}^{(\text{test})}$. We can also see that

$$\text{MSE}_{\text{test}} = \frac{1}{m}||\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})}||_2^2,$$

so the error increases whenever the Euclidean distance between the predictions and the targets increases.

To make a machine learning algorithm, we need to design an algorithm that will improve the weights $\boldsymbol{w}$ in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$. One intuitive way of doing this (which we will justify later) is just to minimize the mean squared error on the training set, $\text{MSE}_{\text{train}}$.

To minimize $\text{MSE}_{\text{train}}$, we can simply solve for where its gradient is $\mathbf{0}$:

$$\begin{aligned}
&\nabla_{\boldsymbol{w}}\text{MSE}_{\text{train}} = 0 \\
\Rightarrow\ &\nabla_{\boldsymbol{w}}\frac{1}{m}||\hat{\boldsymbol{y}}^{(\text{train})} - \boldsymbol{y}^{(\text{train})}||_2^2 = 0 \\
\Rightarrow\ &\frac{1}{m}\nabla_{\boldsymbol{w}}||\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}||_2^2 = 0 \\
\Rightarrow\ &\nabla_{\boldsymbol{w}}(\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})})^\top(\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}) = 0 \\
\Rightarrow\ &\nabla_{\boldsymbol{w}}(\boldsymbol{w}^\top\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - 2\boldsymbol{w}^\top\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} + \boldsymbol{y}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})}) = 0 \\
\Rightarrow\ &2\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - 2\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})} = 0 \\
\Rightarrow\ &\boldsymbol{w} = (\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})})^{-1}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})}
\end{aligned} \tag{5.1}$$

The system of equations defined by Eq. 5.1 is known as the *normal equations*. Solving these equations constitutes a simple learning algorithm. For an example of the linear regression learning algorithm in action, see Fig. 5.1.

It's worth noting that the term *linear regression* is often used to refer to a slightly more sophisticated model with one additional parameter—an intercept term b. In this model

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

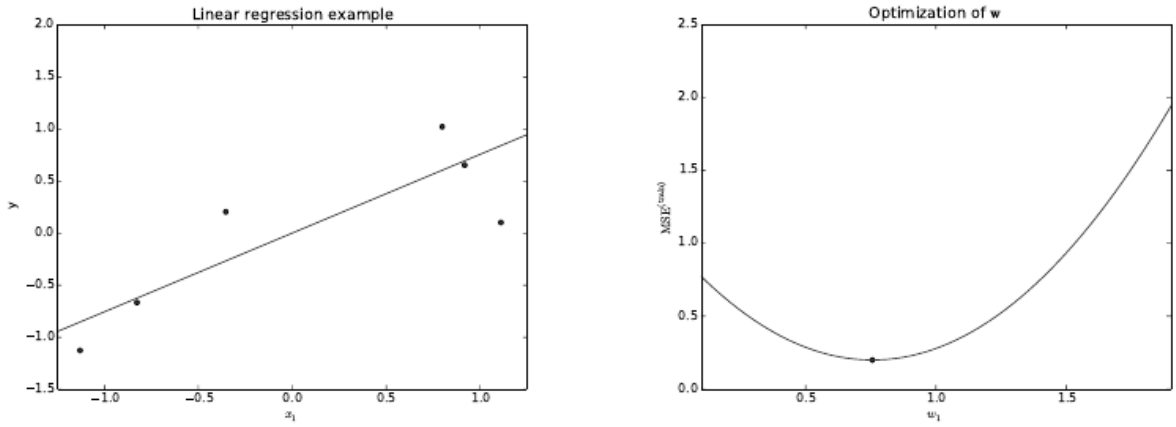


Figure 5.1: Consider this example linear regression problem, with a training set consisting of 5 data points, each containing one feature. This means that the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)* Observe that linear regression learns to set w_1 such that the line $y = w_1 x$ comes as close as possible to passing through all the training points. *(Right)* The plotted point indicates the value of w_1 found by the normal equations, which we can see minimizes the mean squared error on the training set.

so the mapping from parameters to predictions is still a linear function but the mapping from features to predictions is now an affine function. This extension to affine functions means that the plot of the model's predictions still looks like a line, but it need not pass through the origin. We will frequently use the term "linear" when referring to affine functions throughout this book.

Linear regression is of course an extremely simple and limited learning algorithm, but it provides an example of how a learning algorithm can work. In the subsequent sections we will describe some of the basic principles underlying learning algorithm design and demonstrate how these principles can be used to build more complicated learning algorithms.

5.3 Generalization, Capacity, Overfitting and Underfitting

The central challenge in machine learning is that we must perform well on *new, previously unseen* inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called *generalization*.

Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the *training error*, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is

that we want the *generalization error* to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a *test set* of examples that were collected separate from the training set.

In our linear regression example, we trained the model by minimizing the training error,

$$\frac{1}{m^{(\text{train})}} ||\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}||_2^2,$$

but we actually care about the test error, $\frac{1}{m^{(\text{test})}} ||\boldsymbol{X}^{(\text{test})}\boldsymbol{w} - \boldsymbol{y}^{(\text{test})}||_2^2$.

How can we affect performance on the test set when we only get to observe the training set? The field of *statistical learning theory* provides some answers. If the training and the test set are collected arbitrarily, there is indeed little we can do. If we are allowed to make some assumptions about how the training and test set are collected, then we can make some progress.

We typically make a set of assumptions known collectively as the *i.i.d. assumptions*. These assumptions are that the examples in each dataset are *independent* from each other, and that the train set and test set are *identically distributed*, drawn from the same probability distribution as each other. We call that shared underlying distribution the *data generating distribution*, or *data generating process* (which is particularly relevant if the examples are not independent). This probabilistic framework allows us to mathematically study the relationship between training error and test error.

One immediate connection we can observe between the training and test error is that for a randomly selected model, the two have the same expected value. Suppose we have a probability distribution $p(\boldsymbol{x}, y)$ and we sample from it repeatedly to generate the train set and the test set. For some fixed value $\boldsymbol{w}$, then the expected training set error under this sampling process is exactly the same as the expected test set error under this sampling process. The only difference between the two conditions is the name we assign to the dataset we sample. From this observation, we can see that it is natural for there to be some relationship between training and test error under these assumptions.

Of course, when we use a machine learning algorithm, we do not fix the parameters ahead of time, then sample both datasets. We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform are its ability to:

1. Make the training error small.

2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: *underfitting* and *overfitting*. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its *capacity*. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit, i.e., memorize properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its *hypothesis space*, the set of functions that the learning algorithm is allowed to choose as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model's capacity.

A polynomial of degree one gives us the linear regression model with which we are already familiar, with prediction

$$\hat{y} = b + wx.$$

By introducing x^2 as another feature provided to the linear regression model, we can learn a model that is quadratic as a function of x:

$$\hat{y} = b + w_1 x + w_2 x^2.$$

Note that this is still a linear function of the parameters, so we can still use the normal equations to train the model in closed form. We can continue to add more powers of x as additional features, for example to obtain a polynomial of degree 9:

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i.$$

Machine learning algorithms will generally perform best when their capacity is appropriate in regard to the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Model with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task they may overfit.

Fig. 5.2 shows this principle in action. We compare a linear, quadratic, and degree-9 predictor attempting to fit a problem where the true underlying function is quadratic. The linear function is unable to capture the curvature in the true underlying problem, so it underfits. The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely many other functions that pass exactly through the training points, because we have more parameters than training examples. We have little chance of choosing a solution that generalizes well when so many wildly different solutions exist. In this example, the quadratic model is perfectly matched the true structure of the task so it generalizes well to new data.

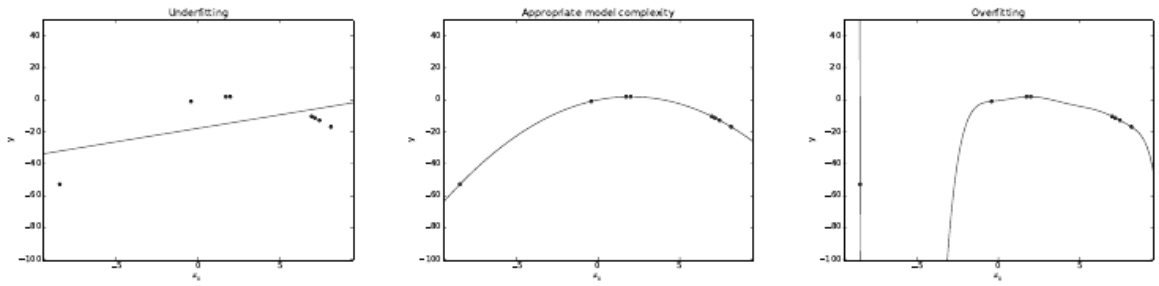


Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. (*Left*) A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. (*Center*) A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. (*Right*) A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudo-inverse to solve the underdetermined normal equations. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

Here we have only described changing a model's capacity by changing the number of input features it has (and simultaneously adding new parameters associated with those features). There are many other ways of controlling the capacity of a machine learning algorithm, which we will explore in the sections ahead.

Many of these ideas date back to *Occam's razor* (c. 1287-1347), also known as the principle of parsimony, which states that among competing hypotheses (here, read functions that could explain the observed data), one should choose the "simpler" one. This idea was formalized and made more precise in the 20th century by the founders of statistical learning theory (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). This body of work provides various

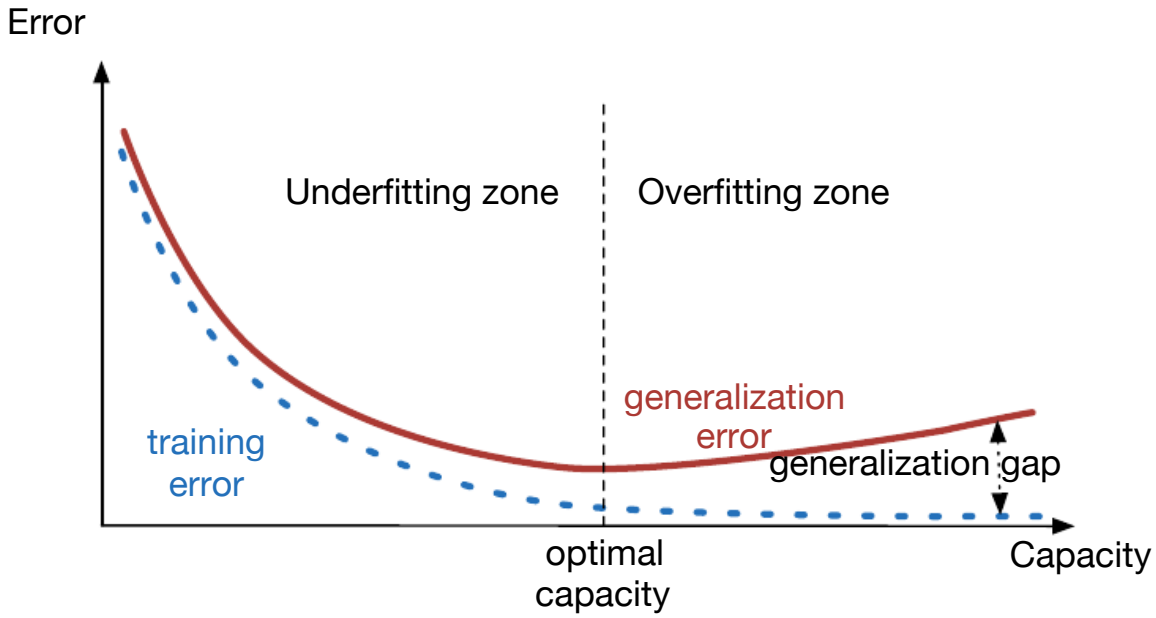


Figure 5.3: Typical relationship between capacity (horizontal axis) and both training (bottom curve, dotted) and generalization (or test) error (top curve, bold). At the left end of the graph, training error and generalization error are both high. This is the *underfitting regime*. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the *overfitting regime*, where capacity is too large, above the *optimal capacity*.

means of quantifying model capacity and showing that the discrepancy between training error and generalization error is bounded by a quantity that grows with the ratio of capacity to number of training examples (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). These bounds provide intellectual justification that machine learning algorithms can work, but they are rarely used in practice when working with deep learning algorithms. This is in part because the bounds are often quite loose, and in part because it can be quite difficult to determine the capacity of deep learning algorithms.

Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming your error measure has a minimum value). Typically, generalization error has a U-shaped curve as a function of model capacity. This is illustrated in Figure 5.3.

Training and generalization error also vary as the size of the training set varies. See Fig. 5.4 for an illustration. The figure introduces the notion of *parametric* and *non-parametric* learning algorithms. Parametric ones have a fixed maximum capacity (their capacity can still be reduced by various means, such as a poor

optimization procedure), and they are called like this because they have a fixed-size parameter vetor. On the other hand, non-parametric learners are allowed to set their capacity based on the given data, i.e., the number of parameters is something that can be determined after the data is observed, and typically more data allows a greater capacity, i.e., more parameters. Note that it is possible for the model to have optimal capacity and yet still have a large gap between training and generalization error. In this situation, we can only reduce this gap by gathering more training examples.

It's worth mentioning that capacity is not just determined by which model we use. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective. This is called the *representational capacity* of the model. In many cases, finding the best function within this family is a very difficult optimization problem. In practice, the learning algorithm does not actually find the best function, just one that significantly reduces the training error. These additional restrictions mean that the model's *effective capacity* may be less than its *representational capacity.*

5.4 The No Free Lunch Theorem

Although learning theory, sketched above, suggests that it is possible to generalize, one should consider a serious caveat, discussed here. Generally speaking, inductive reasoning, or inferring general rules from a limited set of examples, is not logically valid. To logically infer a rule describing every member of a set, one must have information about every member of that set. One may wonder then how the claims that machine learning can generalize well are logically valid.

In part, machine learning avoids this problem by offering only probabilistic rules, rather than the entirely certain rules used in purely logical reasoning. Machine learning promises to find rules that are *probably* correct about *most* members of the set they concern.

Unfortunately, even this does not resolve the entire problem. The *no free lunch theorem* for machine learning (Wolpert, 1996) states that, averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally any better than any other. The most sophisticated algorithm we can conceive of has the same average performance (over all possible tasks) as merely predicting that every point belongs to the same class.

Fortunately, these results hold only when we average over *all* possible data generating distributions. If we make assumptions about the kinds of probability distributions we encounter in real-world applications, then we can design learning

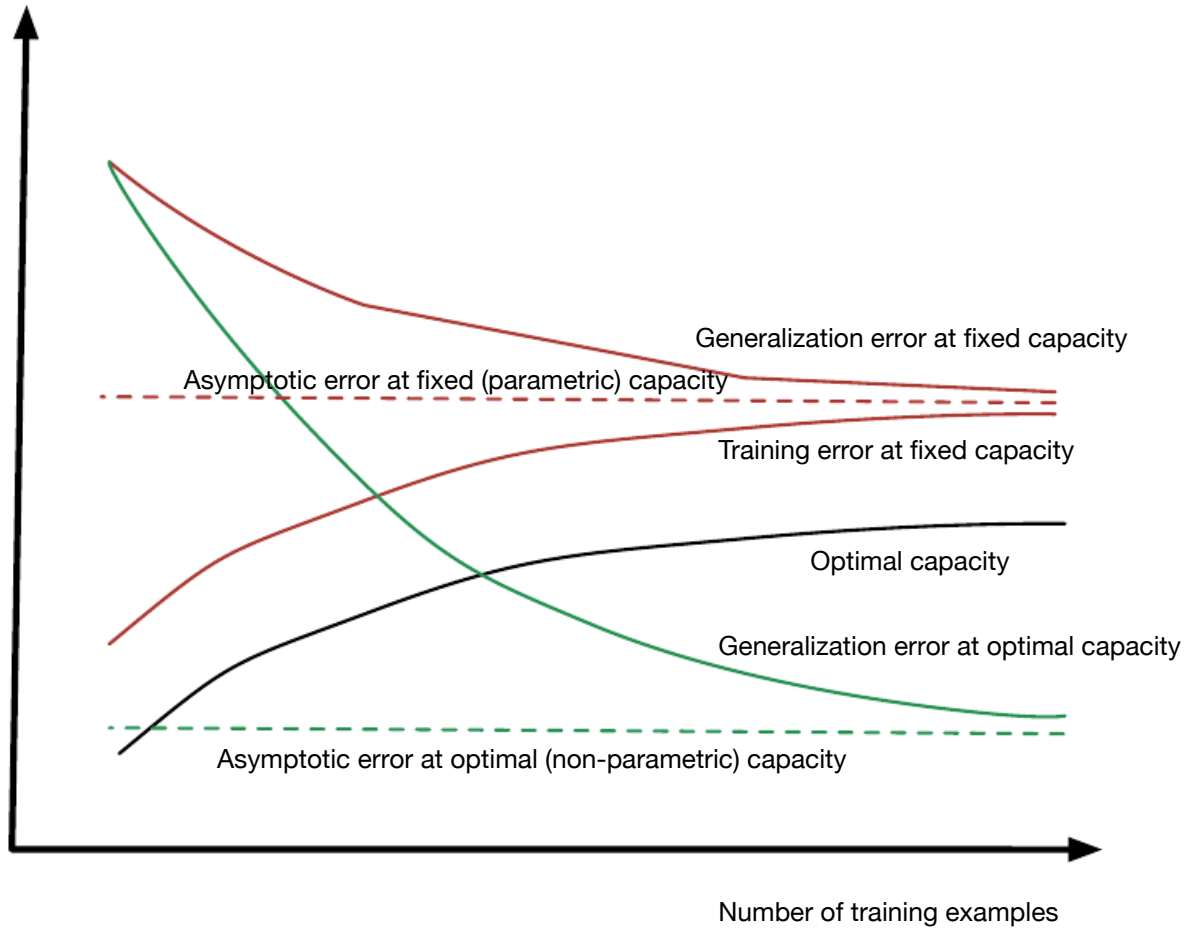


Figure 5.4: This plot shows the effect of the dataset size on the train and test error of the model, as well as on the optimal model capacity. Note that the y-axis is used to show two different values which are not actually comparable—error, and capacity. If we choose a single model with fixed capacity (red), known as a *parametric learning algorithm* and retrain it with different amounts of training data, then the training error will increase as the size of the training set increases. This is because larger datasets are harder to fit. Simultaneously, the test error will decrease, because fewer incorrect hypotheses will be consistent with the training data. Ultimately the train and test error will converge. If we instead consider a learning algorithm that can adapt its capacity with training set size, then the optimal capacity (black) increases with the number of training examples, and reaches an asymptote which only depends on the required complexity for the task to be learned. This kind of learning algorithm is called *non-parametric*. The generalization error of the optimal capacity model (green) decreases and approaches an asymptote, called the *Bayes error* (the error made by an oracle that knows the data generating distribution). Usually the asymptotic error is greater than zero because there is some noise in the true distribution that the model is asked to capture.

algorithms that perform well on these distributions.

This means that the goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, our goal is to understand what kinds of distributions are relevant to the "real world" that an AI agent experiences, and what kinds of machine learning algorithms perform well on data drawn from the kinds of data generating distributions we care about.

5.5 Regularization

The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better.

So far, the only method of modifying a learning algorithm we have discussed is to increase or decrease the model's capacity by adding or removing functions from the hypothesis space of solutions the learning algorithm is able to choose. We gave the specific example of increasing or decreasing the degree of a polynomial for a regression problem. The view we have described so far is oversimplified.

The behavior of our algorithm is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. The learning algorithm we have studied so far, linear regression, has a hypothesis space consisting of the set of linear functions of its input. These linear functions can be very useful for problems where the relationship between inputs and outputs truly is close to linear. They are less useful for problems that behave in a very non-linear fashion. For example, linear regression would not perform very well if we tried to use it to predict $\sin(x)$ from x. We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

We can also give a learning algorithm a preference for one solution in its hypothesis space to another. This means that both functions are eligible, but one is preferred. The unpreferred solution may only be chosen if it fits the training data significantly better than the preferred solution.

For example, we can modify the training criterion for linear regression to include *weight decay*. To perform linear regression with weight decay, we minimize not only the mean squared error on the training set, but instead a criterion $J(\boldsymbol{w})$ that expresses a preference for the weights to have smaller squared L^2 norm. Specifically,

$$J(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w},$$

where λ is a value chosen ahead of time that controls the strength of our preference

for smaller weights. When $\lambda = 0$, we impose no preference, and larger λ forces the weights to become smaller. Minimizing $J(\boldsymbol{w})$ results in a choice of weights that make a tradeoff between fitting the training data and being small. This gives us solutions that have a smaller slope, or put weight on fewer of the features. As an example of how we can control a model's tendency to overfit or underfit via weight decay, we can train a high-degree polynomial regression model with different values of λ. See Fig. 5.5 for the results.

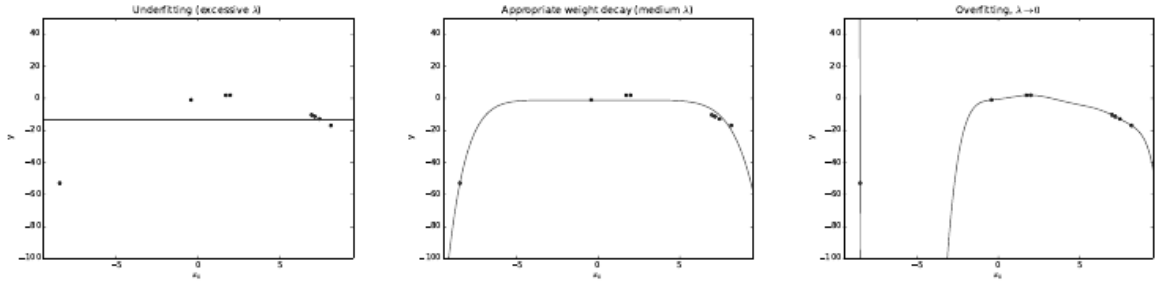


Figure 5.5: We fit a high-degree polynomial regression model to our example training set from Fig. 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. (*Left*) With very large λ, we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. (*Center*) With a medium value of λ, the learning algorithm recovers a curve with the right general shape. Even though the model is capable of representing functions with much more complicated shape, weight decay has encouraged it to use a simpler function described by smaller coefficients. (*Right*) With weight decay approaching zero (i.e., using the Moore-Penrose pseudo-inverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in Fig. 5.2.

Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function.

In our weight decay example, we expressed our preference for linear functions defined with smaller weights explicitly, via an extra term in the criterion we minimize. There are many other ways of expressing preferences for different solutions, both implicitly and explicitly. Together, these different approaches are known as *regularization*. **Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.** Regularization is one of the central concerns of the field of machine learning, rivalled in its importance only by optimization. KEY IDEA

The no free lunch theorem has made it clear that there is no best machine learning algorithm, and in particular, no best form of regularization. Instead we

must choose a form of regularization that is well-suited to the particular task we want to solve. The philosophy of deep learning in general and this book in particular is that a very wide range of tasks (such as all of the intellectual tasks that people can do) may all be solved effectively using very general-purpose forms of regularization.

5.6 Hyperparameters, Validation Sets and Cross-Validation

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called *hyperparameters*. The values of hyperparameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure where one learning algorithm learns the best hyperparameters for another learning algorithm).

In the polynomial regression example we saw in Fig. 5.2, there is a single hyperparameter: the degree of the polynomial, which acts as a *capacity* hyperparameter. The λ value used to control the strength of weight decay is another example of a hyperparameter.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because it is difficult to optimize. More frequently, we do not learn the hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting (refer to Figure 5.3). For example, we can always fit the training set better with a higher degree polynomial, and a weight decay setting of $\lambda = 0$..

To solve this problem, we need a *validation set* of examples that the training algorithm does not observe.

Earlier we discussed how a held-out test set, composed of examples coming from the same distribution as the training set, can be used to estimate the generalization error of a learner, after the learning process has completed. It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set.

For this reason, we always construct the validation set from the *training* data. Specifically, we split the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly. The subset of data used to learn the parameters is still typically called the training set, even though this may be confused with the larger pool of data used for the entire training process.

The subset of data used to guide the selection of hyperparameters is called the validation set. Since the validation set is used to "train" the hyperparameters, the validation set error will underestimate the test set error, though typically by a smaller amount than the training error. Typically, one uses about 80% of the data for training and 20% for validation.

In practice, when the same test set has been used repeatedly to evaluate performance of different algorithms over many years, and especially if we consider all the attempts from the scientific community at beating the reported state-of-the-art performance on that test set, we end up having optimistic evaluations with the test set as well. Benchmarks can thus become stale and then do not reflect the true field performance of a trained system. Thankfully, the community tends to move on to new (and usually more ambitious and larger) benchmark datasets.

5.6.1 Cross-Validation

One issue with the idea of splitting the dataset into train/test or train/validation/test subsets is that only a small fraction of examples are used to evaluate generalization. The consequence is that there is a lot of statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task.

With large datasets with hundreds of thousands of examples or more, this is not a serious issue, but when the dataset is too small, there are alternative procedures, which allow one to use all of the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the training / testing computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the k-fold cross-validation procedure, in which a partition of the dataset is formed by splitting it in k non-overlapping subsets. Then k train/test splits can be obtained by keeping each time the i-th subset as a test set and the rest as a training set. The average test error across all these k training/testing experiments can then be reported. One problem is that there exists no unbiased estimators of the variance of such average error estimators (Bengio and Grandvalet, 2004), but approximations are typically used.

If model selection or hyperparameter optimization is required, things get more computationally expensive: one can recurse the k-fold cross-validation idea, inside the training set. So we can have an outer loop that estimates test error and provides a "training set" for a hyperparameter-free learner, calling it k times to "train". That hyperparameter-free learner can then split its received training set by k-fold cross-validation into internal training/validation subsets (for example, splitting into $k-1$ subsets is convenient, to reuse the same test blocks as the outer loop), call a hyperparameter-specific learner for each choice of hyperparam-

eter value on each of the training partition of this inner loop, and compute the validation error by averaging across the $k-1$ validation sets the errors made by the $k-1$ hyperparameter-specific learners trained on each of the internal training subsets.

5.7 Estimators, Bias, and Variance

The field of statistics gives us many tools that can be used to achieve the machine learning goal of solving a task not only on the training set but also to generalize. Foundational concepts such as parameter estimation, bias and variance are useful to formally characterize notions of generalization, underfitting and overfitting.

5.7.1 Point Estimation

Point estimation is the attempt to provide the single "best" prediction of some quantity of interest. In general the quantity of interest can be a single parameter or a vector of parameters in some parametric model, such as the weights in our linear regression example in Section 5.2, but it can also be a whole function.

In order to distinguish estimates of parameters from their true value, our convention will be to denote a point estimate of a parameter θ by $\hat{\theta}$.

Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ be a set of m *independent and identically distributed* (i.i.d.) data points. A **point estimator** is any function of the data:

$$\hat{\theta}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}). \tag{5.2}$$

In other words, any statistic[1] is a point estimate. Notice that no mention is made of any correspondence between the estimator and the parameter being estimated. There is also no constraint that the range of $g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ should correspond to that of the true parameter.

This definition of a point estimator is very general and allows the designer of an estimator great flexibility. What distinguishes "just any" function of the data from most of the estimators that are in common usage is their properties. For now, we take the frequentist perspective on statistics. That is, we assume that the true parameter value θ is fixed but unknown, while the point estimate $\hat{\theta}$ is a function of the data. Since the data is drawn from a random process, any function of the data is random. Therefore $\hat{\theta}$ is a random variable.

Point estimation can also refer to the estimation of the relationship between input and target variables. We refer to these types of point estimates as function estimators.

[1] A statistic is a function of the data, typically of the whole training set, such as the mean.

Function Estimation As we mentioned above, sometimes we are interested in performing function estimation (or function approximation). Here we are trying to predict a variable (or vector) $\boldsymbol{y}$ given an input vector $\boldsymbol{x}$ (also called the covariates). We consider that there is a function $f(\boldsymbol{x})$ that describes the relationship between $\boldsymbol{y}$ and $\boldsymbol{x}$. For example, we may assume that $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ stands for the part of $\boldsymbol{y}$ that is not predictable from $\boldsymbol{x}$.

In function estimation, we are interested in approximating f with a model or estimate $\hat{f}$. Note that we are really not adding anything new here to our notion of a point estimator, the function estimator $\hat{f}$ is simply a point estimator in function space.

The linear regression example we discussed above in Section. 5.2 and the polynomial regression example discussed in Section. 5.3 are both examples of function estimation where we estimate a model $\hat{f}$ of the relationship between an input $\boldsymbol{x}$ and target $\boldsymbol{y}$.

In the following we will review the most commonly studied properties of point estimators and discuss what they tell us about these estimators.

As $\hat{\theta}$ and $\hat{f}$ are random variables (or vectors, or functions), they are distributed according to some probability distribution. We refer to this distribution as the *sampling distribution*. When we discuss properties of the estimator, we are really describing properties of the sampling distribution.

5.7.2 Bias

The bias of an estimator is defined as:

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta \tag{5.3}$$

where the expectation is over the data (seen as samples from a random variable) and θ is the true underlying value of θ according to the data generating distribution. An estimator $\hat{\theta}_m$ is said to be *unbiased* if $\text{bias}(\hat{\theta}_m) = 0$, i.e., if $\mathbb{E}(\hat{\theta}_m) = \theta$. An estimator $\hat{\theta}_m$ is said to be *asymptotically unbiased* if $\lim_{m\to\infty} \text{bias}(\hat{\theta}_m) = 0$, i.e., if $\lim_{m\to\infty} \mathbb{E}(\hat{\theta}_m) = \theta$.

Example: Bernoulli Distribution Consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Bernoulli distribution, $x^{(i)} \in \{0, 1\}$, where $i \in [1, m]$. The Bernoulli *p.m.f.* (probability mass function, or probability function) is given by $P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})}$.

We are interested in knowing if the estimator $\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$ is biased.

$$
\begin{aligned}
\text{bias}(\hat{\theta}_m) &= \mathbb{E}[\hat{\theta}_m] - \theta \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right] - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} \sum_{x^{(i)}=0}^{1} \left(x^{(i)}\, \theta^{x^{(i)}} (1-\theta)^{(1-x^{(i)})}\right) - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} (\theta) - \theta \\
&= \theta - \theta = 0
\end{aligned}
$$

Since $\text{bias}(\hat{\theta}) = 0$, we say that our estimator $\hat{\theta}$ is unbiased.

Example: Gaussian Distribution Estimator of the Mean Now, consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Gaussian (Normal) distribution ($x^{(i)} \sim \text{Gaussian}(\mu, \sigma^2)$, where $i \in [1, m]$). The Gaussian *p.d.f.* (probability density function) is given by $p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x^{(i)}-\mu)^2}{\sigma^2}\right)$.

A common estimator of the Gaussian mean parameter is known as the *sample mean*:

$$\hat{\mu}_m = \frac{1}{n}\sum_{i=1}^{m} x^{(i)} \tag{5.4}$$

To determine the bias of the sample mean, we are again interested in calculating its expectation:

$$
\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}[\hat{\mu}_m] - \mu \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right]\right) - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mu\right) - \mu \\
&= \mu - \mu = 0
\end{aligned}
$$

Thus we find that the sample mean is an unbiased estimator of Gaussian mean parameter.

Example: Gaussian Distribution Estimators of the Variance Sticking with the Gaussian family of distributions. We consider two different estimators of the variance parameter σ^2. We are interested in knowing if either estimator is biased.

The first estimator of σ^2 we consider is known as the *sample variance*:

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2, \tag{5.5}$$

where $\hat{\mu}_m$ is the sample mean, defined above. More formally, we are interested in computing

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2$$

We now simplify the term $\mathbb{E}[\hat{\sigma}_m^2]$

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2 - 2x^{(i)}\hat{\mu}_m + \hat{\mu}_m^2\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[(x^{(i)})^2\right] - 2\mathbb{E}\left[x^{(i)}\frac{1}{m}\sum_{j=1}^{m}x^{(j)}\right] + \mathbb{E}\left[\left(\frac{1}{m}\sum_{j=1}^{m}x^{(j)}\right)\left(\frac{1}{m}\sum_{k=1}^{m}x^{(k)}\right)\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(\left(1-\frac{2}{m}\right)\mathbb{E}\left[(x^{(i)})^2\right] - \frac{2}{m}\sum_{j\neq i}\mathbb{E}\left[x^{(i)}x^{(j)}\right] + \frac{1}{m^2}\sum_{j=1}^{m}\mathbb{E}\left[(x^{(j)})^2\right]\right. \\
&\quad \left. + \frac{1}{m^2}\sum_{j=1}^{m}\sum_{k\neq j}\mathbb{E}\left[x^{(j)}x^{(k)}\right]\right) \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(\left(\frac{m-2}{m}\right)(\mu^2+\sigma^2) - \frac{2(m-1)}{m}(\mu^2) + \frac{1}{m}(\mu^2+\sigma^2) + \frac{(m-1)}{m}(\mu^2)\right) \\
&= \frac{m-1}{m}\sigma^2
\end{aligned}$$

So the bias of $\hat{\sigma}_m^2) = -\sigma^2/m$, and therefore the sample variance is a biased estimator.

We now consider a modified estimator of the variance sometimes called the *unbiased sample variance*:

$$\tilde{\sigma}^{\,2}_{m} = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_{\,m}\right)^2 \tag{5.6}$$

As the name suggests this estimator is unbiased, that is, we find that $\mathbb{E}[\tilde{\sigma}_m^{\,2}\,] = \sigma^2$:

$$\begin{aligned}
\mathbb{E}[\tilde{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_{\,m}\right)^2\right] \\
&= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}_m^2] \\
&= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) \\
&= \sigma^2 .
\end{aligned}$$

We have two estimators: one is biased and the other is not. While unbiased estimators are clearly desirable, they are not always the "best" estimators. As we will see we often use biased estimators that possess other important properties.

5.7.3 Variance

Another property of the estimator that we might want to consider is how much we expect it to vary as a function of the data sample. Just as we computed the expectation of the estimator to determine its bias, we can compute its *variance.*

$$\mathrm{Var}(\,\hat{\theta}) = \mathbb{E}[\hat{\theta}^2\,] - \mathbb{E}[\hat{\theta}]^{\,2} \tag{5.7}$$

The variance of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently resample the dataset from the underlying data generating process. Just as we might like an estimator to exhibit low bias we would also like it to have relatively low variance.

We can also define the standard error (se) of the estimator as

$$\mathrm{se}(\,\hat{\theta}) = \sqrt{\mathrm{Var}[\hat{\theta}]} \tag{5.8}$$

Example: Bernoulli Distribution Let's once again consider a set samples $(\{x^{(1)}, \ldots, x^{(m)}\})$ drawn independently and identically from a Bernoulli distribution (recall $P(x^{(i)};\theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}$). This time we are interested in

computing the variance of the estimator $\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$.

$$\begin{aligned}
\text{Var}\left(\hat{\theta}_m\right) &= \text{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \text{Var}\left(x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \\
&= \frac{1}{m^2} m\theta(1-\theta) \\
&= \frac{1}{m}\theta(1-\theta)
\end{aligned}$$

Note that the variance of the estimator decreases as a function of m, the number of examples in the dataset. This is a common property of popular estimators that we will return to when we discuss consistency (see Sec. 5.7.5).

Example: Gaussian Distribution Estimators of the Variance We again consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ independently and identically distributed according to a Gaussian distribution ($x^{(i)} \sim \text{Gaussian}(\mu, \sigma^2)$, where $i \in [1, m]$).

We now consider the variance of the two estimators of the variance: the sample variance,

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(1)} - \hat{\mu}_m\right)^2, \tag{5.9}$$

and the unbiased sample variance,

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(1)} - \hat{\mu}_m\right)^2. \tag{5.10}$$

In order to determine the variance of these estimators we will take advantage of a known relationship between the sample variance and the Chi Squared distribution, ppecifically, that $\frac{m-1}{\sigma^2}\hat{\sigma}^2$ happens to be χ^2 distributed. We can then use this together with the fact that the variance of a χ^2 random variable with $m-1$ degrees of freedom is $2(m-1)$.

$$\begin{aligned}
\text{Var}\left(\frac{m-1}{\sigma^2}\tilde{\sigma}^2\right) &= 2(m-1) \\
\frac{(m-1)^2}{\sigma^4}\text{Var}\left(\tilde{\sigma}^2\right) &= 2(m-1) \\
\text{Var}\ \tilde{\sigma}^2 &= \frac{2\sigma^4}{(m-1)}
\end{aligned}$$

By noticing that $\hat{\sigma}^2 = \frac{m-1}{m}\tilde{\sigma}^2$, and using $\tilde{\sigma}^2$'s relationship to the χ^2 distribution, it is straightforward to show that $\text{Var}\left(\hat{\sigma}^2\right) = \frac{2(m-1)\sigma^4}{m^2}$.

To derive this last relation, we used the fact that that $\text{Var}\left(\tilde{\sigma}^2\right) = \left(\frac{m}{m-1}\right)^2 \text{Var}\left(\hat{\sigma}^2\right)$, that is $\text{Var}\left(\tilde{\sigma}^2\right) > \text{Var}\left(\hat{\sigma}^2\right)$. So while the bias of $\tilde{\sigma}^2$ is smaller than the bias of $\hat{\sigma}^2$, the variance of $\tilde{\sigma}^2$ is greater.

5.7.4 Trading off Bias and Variance and the Mean Squared Error

Bias and variance measure two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the true value that any particular sampling of the data is likely to cause.

What happens when we are given a choice between two estimators, one with more bias and one with less variance? How do we choose between them? For example, let's imagine that we are interested in approximating the function shown in Fig. 5.2 and we are only offered the choice between a model with large bias and one that suffers from large variance. How do we choose between them?

In machine learning, perhaps the most common and empirically successful way to negotiate this kind of trade-off, in general is by cross-validation, discussed in Section 5.6.1. Alternatively, we can also compare the *mean squared error* (MSE) of the estimates:

$$\begin{aligned} \text{MSE} &= \mathbb{E}[\hat{\theta}_n - \theta]^2 \\ &= \text{Bias}(\hat{\theta}_n)^2 + \text{Var}(\hat{\theta}_n) \end{aligned} \tag{5.11}$$

The MSE measures the overall expected deviation—in a squared error sense—between the estimator and the true value of the parameter θ. As is clear from Eq. 5.11, evaluating the MSE incorporates both the bias and the variance. Desirable estimators are those with small MSE and these are estimators that manage to keep both their bias and variance somewhat in check.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting discussed in Section. 5.3. In the case where generalization error is measured by the MSE (where bias and variance are meaningful components of generalization error), increasing capacity tends to increase variance and decrease bias. This is illustrated in Figure 5.6, where we see again the U-shaped curve of generalization error as a function of of capacity, as in Section 5.3 and Figure 5.3.

Example: Gaussian Distribution Estimators of the Variance In the last section we saw that when we compared the sample variance, $\hat{\sigma}^2$, and the unbiased sample variance, $\tilde{\sigma}^2$, we see that while $\hat{\sigma}^2$ has higher bias, $\tilde{\sigma}^2$ has higher variance.

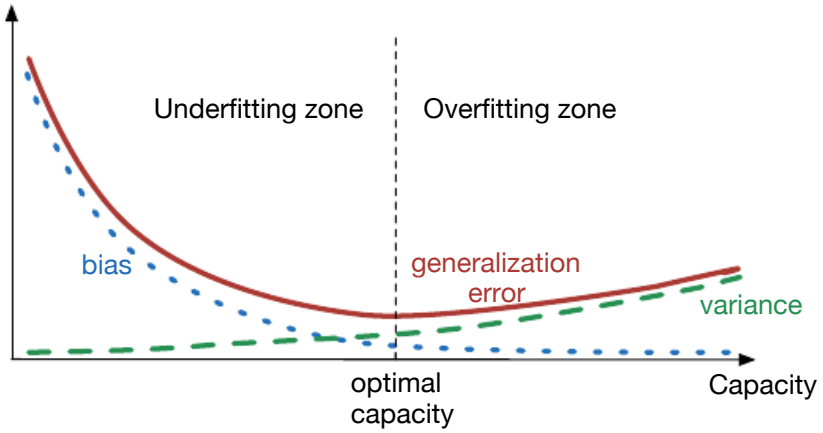


Figure 5.6: As capacity increases (x-axis), bias (dotted) decreases and variance (dashed) increases, yielding another U-shaped curve for generalization error (bold curve). If we vary capacity along one axis, there is an optimal capacity, with underfitting when the capacity is below this optimum and overfitting when it is above.

The mean squared error offers a way of balancing the tradeoff between bias and variance and suggest which estimator we might prefer. For $\hat{\sigma}^2$, the mean squared error is given by:

$$\text{MSE}(\hat{\sigma}_m^2) = \text{Bias}(\hat{\sigma}_m^2)^2 + \text{Var}(\hat{\sigma}_m^2) \tag{5.12}$$

$$= \left(\frac{-\sigma^2}{m}\right)^2 + \frac{2(m-1)\sigma^4}{m^2} \tag{5.13}$$

$$= \left(\frac{1+2(m-1)}{m^2}\right)\sigma^4 \tag{5.14}$$

$$= \left(\frac{2m-1}{m^2}\right)\sigma^4 \tag{5.15}$$

The mean squared error of the unbiased alternative is given by:

$$\text{MSE}(\tilde{\sigma}_m^2) = \text{Bias}(\tilde{\sigma}_m^2)^2 + \text{Var}(\tilde{\sigma}_m^2) \tag{5.16}$$

$$= 0 + \frac{2\sigma^4}{(m-1)} \tag{5.17}$$

$$= \frac{2}{(m-1)}\sigma^4. \tag{5.18}$$

Comparing the two, we see that the MSE of the unbiased sample variance, $\tilde{\sigma}_m^2$, is actually higher than the MSE of the (biased) sample variance, $\hat{\sigma}_m^2$. This implies that despite incurring bias in the estimator $\hat{\sigma}_m^2$, the resulting reduction in variance more than makes up for the difference, at least in a mean squared sense.

5.7.5 Consistency

As we have already discussed, sometimes we may wish to choose an estimator that is biased. For example, in order to minimize the variance of the estimator. However we might still wish that, as the number of data points in our dataset increases, our point estimates converge to the true value of the parameter. More formally, we would like that $\lim_{n\to\infty} \hat{\theta}_n \xrightarrow{p} \theta$?[2] This condition is known as *consistency*[3] and ensures that the bias induced by the estimator is assured to diminish as the number of data examples grows.

Asymptotic unbiasedness is not equivalent to consistency. For example, consider estimating the mean parameter μ of a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with a dataset consisting of n samples: $\{x_1, \ldots, x_n\}$. We could use the first sample x_1 of the dataset as an *unbiased* estimator: $\hat{\theta} = x_1$, In that case, $\mathbb{E}(\hat{\theta}_n) = \theta$ so the estimator is unbiased no matter how many data points are seen. This, of course, implies that the estimate is asymptotically unbiased. However, this is not a consistent estimator as it is *not* the case that $\hat{\theta}_n \to \theta$ as $n \to \infty$.

5.8 Maximum Likelihood Estimation

In the previous section we discussed a number of common properties of estimators but we never mentioned where these estimators come from. In this section, we discuss one of the most common approaches to deriving estimators: via the maximum likelihood principle.

Consider a set of m independent examples $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ with $\boldsymbol{x}^{(i)} \sim P(\mathbf{x})$ independently, where $P(\mathbf{x})$ is the true but unknown *data generating distribution*. More generally, the data may not need to be sampled independently, so we have a *data generating process* which produces the sequence $\boldsymbol{X}$, i.e., $\boldsymbol{X} \sim P(\mathbf{X})$.

Consider a family of probability functions P, parameterized by θ, over the same space, i.e., $P(\boldsymbol{x}; \theta)$ maps any configuration $\boldsymbol{x}$ to a real number estimating the true probability $P(\boldsymbol{x})$, or more generally (in the non-independent case), we have a $P(\boldsymbol{X}; \theta)$ that returns the probability of any whole sequence $\boldsymbol{X}$.

The maximum likelihood estimator for θ is then defined as

$$\theta_{\mathrm{ML}} = \arg\max_{\theta} P(\boldsymbol{X}; \theta). \tag{5.19}$$

[2]The symbol $\xrightarrow{p}$ means that the convergence is in probability, i.e. for any $\epsilon > 0$, $P(|\hat{\theta}_n - \theta| > \epsilon) \to 0$ as $n \to \infty$.

[3]This is sometime referred to as weak consistency, with strong consistency referring to the *almost sure* convergence of $\hat{\theta}$ to θ.

In the i.i.d. scenario, because of the i.i.d. assumptions, we can rewrite

$$P_\theta(\boldsymbol{X}) = \prod_{i=1}^{n} P(\boldsymbol{x}^{(i)}; \theta). \tag{5.20}$$

Note that sometimes we make our model assume that the examples are i.i.d. even though we know they are not, because it simplifies the model (which sometimes mean that better generalization can be achieved), so this is a modeling choice and not necessarily an assumption on the true data generating process. Combining the above two equations and noting that the logarithm of the arg max is the arg max of the logarithm, we obtain the ordinary maximum likelihood estimator under a model that assumes i.i.d. examples:

$$\theta_{\mathrm{ML}} = \arg\max_\theta \sum_{i=1}^{m} \log P(\boldsymbol{x}^{(i)}; \theta). \tag{5.21}$$

This formulation is convenient because it corresponds to an objective function that is additive in the examples, something that is exploited in numerical optimization methods such as stochastic gradient descent (Section 8.3.2), which is heavily used for deep learning. In practice, we will often use numerical optimization to *approximately* maximize the likelihood, so we will not have the true maximum likelihood estimator, but something that approximates it.

There is an interesting connection between the objective function for maximum likelihood, on the right hand side of Eq. 5.21, and the notion KL divergence introduced in Section 3.9 with Eq. 3.3. The KL divergence compares a candidate distribution Q with a target distribution P. If we replace Q by the empirical distribution, we obtain the average negative log-likelihood plus the entropy of Q, which is a constant as far as P is concerned:

$$\begin{aligned}
D_{\mathrm{KL}}(Q\|P) &= \mathbb{E}_{\boldsymbol{x}\sim Q}\left[\log \frac{Q(\boldsymbol{x})}{P(\boldsymbol{x})}\right] && (5.22)\\
&= \mathbb{E}_{\boldsymbol{x}\sim Q}[-\log P(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim Q}[\log Q(\boldsymbol{x})] && (5.23)\\
& && (5.24)\\
&= \mathbb{E}_{\boldsymbol{x}\sim Q}[-\log P(\boldsymbol{x})] - H[Q] && (5.25)\\
&= -\frac{1}{m}\sum_{i=1}^{m} \log P(\boldsymbol{x}^{(i)}) - \log m && (5.26)
\end{aligned}$$

where Eq. 5.22 is the definition of KL divergence, Eq. 5.23 splits it into two terms, the first one being the *cross entropy* between Q and P (with Q as the reference) and second one being the entropy of Q (see Eq. 3.2). Eq. 5.25 is then obtained by taking Q as the empirical distribution (Eq. 3.4), and we obtain Eq. 5.26, which is

the average negative log-likelihood plus a constant. Hence maximizing likelihood is minimizing the cross entropy between the empirical distribution and the model as well as minimizing the KL divergence between these two distributions. Note that when we make Q the data generating distribution, we obtain the *generalization* or *expected* negative log-likelihood.

5.8.1 Conditional Log-Likelihood and Mean Squared Error

The maximum likelihood estimator can readily be generalized to the case where our goal is not to estimate a probability function but rather a *conditional probability*, e.g., $P(\mathbf{y} \mid \mathbf{x}; \theta)$, to predict $\mathbf{y}$ given $\mathbf{x}$. This is actually the most common situation where we do supervised learning (Section 5.10), i.e., the examples are pairs $(\boldsymbol{x}, \boldsymbol{y})$. If $\boldsymbol{X}$ represents all our inputs and $\boldsymbol{Y}$ all our observed targets, then the conditional maximum likelihood estimator is

$$\theta_{\text{ML}} = \arg\max_{\theta} P(\boldsymbol{Y} \mid \boldsymbol{X}; \theta). \tag{5.27}$$

If the examples are assumed to be i.i.d., then this can be decomposed into

$$\theta_{\text{ML}} = \arg\max_{\theta} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \theta). \tag{5.28}$$

Example: Linear Regression Let us consider as an example the special case of linear regression, introduced earlier in Section 5.2. In that case, the conditional density of $\mathbf{y}$, given $\mathbf{x} = \boldsymbol{x}$, is a Gaussian with mean $\mu(\boldsymbol{x})$ that is a learned function of $\boldsymbol{x}$, with unconditional variance σ^2. Since the examples are assumed to be i.i.d., the conditional log-likelihood (Eq. 5.27) becomes

$$\log P(\boldsymbol{Y} \mid \boldsymbol{X}; \theta) = \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \theta) = \sum_{i=1}^{m} \frac{-1}{2\sigma^2} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2 - m \log \sigma - \frac{m}{2} \log(2\pi)$$

where $\hat{\boldsymbol{y}}^{(i)} = \mu(\boldsymbol{x}^{(i)})$ is the output of the linear regression on the i-th input $\boldsymbol{x}^{(i)}$ and m is the dimension of the $\boldsymbol{y}$ vectors. Comparing the above with the mean squared error (Section 5.2) we immediately see that if σ is fixed, maximizing the above is equivalent (up to an additive and a multiplicative constant that do not change the value of the optimal parameter) to minimizing the training set mean squared error, i.e.,

$$MSE_{train} = \frac{1}{m} \sum_{i=1}^{m} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2.$$

Note that the MSE is an average rather than a sum, which is more practical from a numerical point of view (so you can compare MSEs of sets of different sizes

more easily). In practice, researchers reporting log-likelihoods and conditional log-likelihoods also tend to report the per-example average log-likelihood, for the very same reason. The exponential of the average log-likelihood is also called the *perplexity* and is used in language modeling applications.

Whereas in the case of linear regression we have $\mu(\boldsymbol{x}) = \boldsymbol{w}\cdot\boldsymbol{x}$, the above equally applies to other forms of regression, e.g., with a neural network predicting with $\mu(\boldsymbol{x})$ the expected value of $\mathbf{y}$ given $\boldsymbol{x}$.

5.8.2 Properties of Maximum Likelihood

The main appeal of the maximum likelihood estimator is that it can be shown to be the best estimator asymptotically, as the number of examples $m \to \infty$, in terms of its rate of convergence as m increases.

The maximum likelihood estimator has the property of consistency (see Sec. 5.7.5 above), i.e., as more training are considered, the estimator converges to the best one in some sense. There are other inductive principles besides the maximum likelihood estimator, many of which share the property of being consistent estimators. However, there is the question of how many training examples one needs to achieve a particular generalization error, or equivalently what estimation error one gets for a given number of training examples, also called *efficiency*. This is typically studied in the *parametric case* (like in linear regression) where our goal is to estimate the value of a parameter (and assuming it is possible to identify the true parameter), not the value of a function. A way to measure how close we are to the true parameter is by the expected mean squared error, computing the squared difference between the estimated and true parameter values, where the expectation is over m training samples from the data generating distribution. That parametric mean squared error decreases as m increases, and for m large, the Cramér-Rao lower bound (Rao, 1945; Cramér, 1946) shows that no consistent estimator has a lower mean squared error than the maximum likelihood estimator.

For these reasons (consistency and efficiency), the maximum likelihood induction principle is often considered the preferred one in machine learning, modulo slight adjustments such as described in the next Section, to better deal with the non-asymptotic case where the number of examples is small enough to yield overfitting behavior.

5.9 Bayesian Statistics and Prior Probability Distributions

So far we have discussed approaches based on estimating a single value of $\boldsymbol{\theta}$, then making all predictions thereafter based on that one estimate. Another approach is

to consider all possible values of $\boldsymbol{\theta}$ when making a prediction. *Bayesian statistics* provides a natural and theoretically elegant way to carry out this approach.

Historically, statistics has become divided between two communities. One of these communities is known as *frequentist statistics* or *orthodox statistics*. The other is known as *Bayesian statistics*. The difference is mainly one of world view but can have important practical implications.

As discussed in Sec. 5.7.1, the frequentist perspective is that the true parameter value θ is fixed but unknown, while the point estimate $\hat{\theta}$ is a random variable on account of it being a function of the data (which are seen as random).

The Bayesian perspective on statistics is quite different and, in some sense, more intuitive. The Bayesian uses probability to reflect degrees of certainty of states of knowledge. The data is directly observed and so is not random. On the other hand, the true parameter θ is unknown or uncertain and thus is represented as a random variable.

Before observing the data, we represent our knowledge of θ using the *prior probability distribution*, $p(\theta)$ (sometimes referred to as simply 'the prior'). Generally, the prior distribution is quite broad (i.e. with high entropy) to reflect a high degree of uncertainty in the value of θ before observing any data. For example, we might assume *a priori* that θ lies in some finite range or volume, with a uniform distribution. Many priors instead reflect a preference for "simpler" solutions (such as smaller magnitude coefficients, or a function that is closer to being constant).

Now consider that we have a set of data samples $\{x^{(1)}, \ldots, x^{(m)}\}$. We can recover the effect of data on our belief about θ by combining the data likelihood $p(x^{(1)}, \ldots, x^{(m)} \mid \theta)$ with the prior via Bayes' rule:

$$p(\theta \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \theta) p(\theta)}{p(x^{(1)}, \ldots, x^{(m)})} \tag{5.29}$$

If the data is at all informative about the value of θ, the *posterior distribution* $p(\theta \mid x^{(1)}, \ldots, x^{(m)})$ will have less entropy (will be more 'peaky') than the prior $p(\theta)$.

Relative to maximum likelihood estimation, Bayesian estimation offers two important differences. First, unlike the maximum likelihood point estimate of θ, the Bayesian makes decision with respect to a full distribution over θ. For example, after observing m examples, the predicted distribution over the next data sample, $x^{(m+1)}$, is given by

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \theta) p(\theta \mid x^{(1)}, \ldots, x^{(m)}) \, d\theta \tag{5.30}$$

Here each value of θ with positive probability density contributes to the prediction of the next example, with the contribution weighted by the posterior density itself.

After having observed $\{x^{(1)}, \ldots, x^{(m)}\}$, if we are still quite uncertain about the value of θ, then this uncertainty is incorporated directly into any predictions we might make.

In Sec. 5.7, we discussed how the frequentist statistics addresses the uncertainty in a given point estimator of θ by evaluating its variance. The variance of the estimator is an assessment of how the estimate might change will alternative samplings of the observed (or training) data. The Bayesian answer to the question of how to deal with the uncertainty in the estimator is to simply integrate over it, which tends to protect well against overfitting.

The second important difference between the Bayesian approach to estimation and the Maximum Likelihood approach is due to the contribution of the Bayesian prior distribution. The prior has an influence by shifting probability mass density towards regions of the parameter space that are preferred *a priori*. In practice, the prior often expresses a preference for models that are simpler or more smooth. One important effect of the prior is to actually reduce the uncertainty (or entropy) in the posterior density over θ.

We have already noted that combining the prior, $p(\theta)$, with the data likelihood $p(x^{(1)}, \ldots, x^{(m)} \mid \theta)$ results in a distribution that is less entropic (more peaky) than the prior. This is just the result of a basic property of probability distributions: Entropy(product of two densities) $\leq$ Entropy(either density). This implies that the posterior density on θ is also less entropic than the data likelihood alone (when viewed and normalized as a density over θ). The hypothesis space with the Bayesian approach is, to some extent, more constrained than that with an ML approach. Thus we expect a contribution of the prior to be a further reduction in overfitting as compared to ML estimation.

Example: Linear Regression Here we consider the Bayesian estimation approach to learning the linear regression parameters. In linear regression, we learn a linear mapping from an input vector $\boldsymbol{x} \in \mathbb{R}^n$ to predict the value of a scalar $y \in \mathbb{R}$. The prediction is parametrized by the vector $\boldsymbol{w} \in \mathbb{R}^n$:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}.$$

Given a set of m training samples $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, we can express the prediction of y over the entire training set as:

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})} \boldsymbol{w}.$$

Expressed as a Gaussian conditional distribution on $\boldsymbol{y}^{(\text{train})}$, we have

$$p(\boldsymbol{y}^{(\text{train})} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{w}) = \mathcal{N}(\boldsymbol{y}^{(\text{train})}; \boldsymbol{X}^{(\text{train})\top} \boldsymbol{w}, \boldsymbol{I})$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})} \boldsymbol{w})^\top (\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})} \boldsymbol{w})\right),$$

where we will follow the standard MSE formulation in assuming that the Gaussian variance on y is one. In what follows, to reduce the notational burden, we refer to $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$ as simply $(\boldsymbol{X}, \boldsymbol{y})$.

To determine the posterior distribution over the model parameter vector $\boldsymbol{w}$, we first need to specify a prior distribution. The prior should reflect our naive belief about the value of these parameters. While it is sometimes difficult or unnatural to express our prior beliefs in terms of the parameters of the model, in practice we typically assume a fairly broad distribution expressing a high degree of uncertainty about θ in our prior belief.

For real-valued parameters it is common to use a Gaussian as a prior distribution:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \Lambda_0) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right)$$

where $\boldsymbol{\mu}_0$ and Λ_0 are the prior distribution mean vector and covariance matrix (inverse of covariance matrix) respectively.[4]

With the prior thus specified, we can now proceed in determining the *posterior* distribution over the model parameters.

$$\begin{aligned} p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) &\propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) \\ &\propto \exp\left(-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\right) \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right) \\ &\propto \exp\left(-\frac{1}{2}\left(-2\boldsymbol{y}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \Lambda_0^{-1}\boldsymbol{w} - 2\boldsymbol{\mu}_0^\top \Lambda_0^{-1}\boldsymbol{w}\right)\right) \end{aligned}$$

We now make the substitutions $\Lambda_m = \left(\boldsymbol{X}^\top \boldsymbol{X} + \Lambda_0^{-1}\right)^{-1}$ and $\boldsymbol{\mu}_m = \Lambda_m\left(\boldsymbol{X}^\top \boldsymbol{y} + \Lambda_0^{-1}\boldsymbol{\mu}_0\right)$ into the derivation of the posterior (and complete the square) to get:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \Lambda_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) + \frac{1}{2}\boldsymbol{\mu}_m^\top \Lambda_m^{-1}\boldsymbol{\mu}_m\right) \tag{5.31}$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \Lambda_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m)\right). \tag{5.32}$$

In the above, we have dropped all terms that do not include the parameter vector $\boldsymbol{w}$. In Eq. 5.32, we recognize that the posterior distribution has the form of a Gaussian distribution with mean vector $\boldsymbol{\mu}_m$ and covariance matrix Λ_m. It is interesting to note that this justifies our dropping all terms unrelated to $\boldsymbol{w}$, since we know that the posterior distribution must be normalized and, as a Gaussian,

[4] Unless there is a reason to assume a particular covariance structure, we typically assume a diagonal covariance matrix $\Lambda_0 = \text{diag}(\boldsymbol{\lambda}_0)$.

we know what that normalization constant must be (where n is the dimension of the input):

$$p(\boldsymbol{w} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})}) = \frac{1}{\sqrt{(2\pi)^n \, |\Lambda_m|}} \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \Lambda_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) \right). \tag{5.33}$$

5.9.1 Maximum *A Posteriori* (MAP) Estimation

While, in principle, we can use the full Bayesian posterior distribution over the parameter θ as our estimate of this parameter, it is still often desirable to have a single point estimate (for example, most operations involving the Bayesian posterior for most interesting models are intractable and must be heavily approximated). Rather than simply returning to the maximum likelihood estimate, we can still gain some of the benefit of the Bayesian approach by allowing the prior to influence the choice of the point estimate. One rational way to do this is to choose the *maximum a posteriori* (MAP) point estimate. The MAP estimate chooses the point of maximal posterior probability (or maximal probability density in the more common case of continuous θ).

$$\theta_{\text{MAP}} = \arg\max_\theta p(\theta \mid \boldsymbol{x}) = \arg\max_\theta \log p(\boldsymbol{x} \mid \theta) + \log p(\theta) \tag{5.34}$$

We recognize, above on the right hand side, $\log p(\boldsymbol{x} \mid \theta)$, i.e. the standard log-likelihood term and $\log p(\theta)$ corresponding to the prior distribution.

As discussed above the advantage brought by introducing the influence of the prior on the MAP estimate is to leverage information other than that contained in the training data. This additional information helps to reduce the variance in the MAP point estimate (in comparison to the ML estimate). However, it does so at the price of increased bias.

Example: Regularized Linear Regression We discussed above the Bayesian approach to linear regression. Given a set of m training samples of input output pairs: $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, we can express the prediction of y over the entire training set as:

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})} \boldsymbol{w}.$$

where prediction is parametrized by the vector $\boldsymbol{w} \in \mathbb{R}^n$.

Recall from Sec. 5.8.1 that the maximum likelihood estimate for the model parameters is given by:

$$\hat{\boldsymbol{w}}_{\text{ML}} = (\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})})^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.35}$$

For the sake of comparison to the maximum likelihood solution, we will make the simplifying assumption that the prior covariance matrix is scalar: $\Lambda_0 = \lambda_0 \boldsymbol{I}$. As mentioned previously, in practice, this is a very common form of prior distribution. We will also assume that $\boldsymbol{\mu}_0 = 0$. This is also a very common assumption in practice and corresponds to acknowledging that *a priori*, we do not know if the features of $\boldsymbol{x}$ have a positive or negative correlation with y. Adding these assumptions, the MAP estimate of the model parameters (corresponding to the mean of the Gaussian posterior density, in Eq. 5.32) becomes:

$$\hat{\boldsymbol{w}}_{\text{MAP}} = \Lambda_m \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.36}$$

where $\boldsymbol{\mu}_0$ and Λ_0 are the prior mean and covariance respectively and Λ_m is the posterior covariance and is given by:

$$\Lambda_m = \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} + \lambda_0^{-1} \boldsymbol{I} \right)^{-1} \tag{5.37}$$

Comparing Eqs. 5.35 and 5.36, we see that the MAP estimate amounts to a weighted combination of the prior maximum probability value, $\boldsymbol{\mu}_0$, and the ML estimate. As the variance of the prior distribution tends to infinity, the MAP estimate reduces to the ML estimate. As the variance of the prior tends to zero, the MAP estimate tends to zero (actually it tends to $\boldsymbol{\mu}_0$ which here is assumed to be zero).

We can make the model capacity tradeoff between the ML estimate and the MAP estimate more explicit by analyzing the bias and variance of these estimates.

It is relatively easy to show that the ML estimate is unbiased, i.e. that $\mathbb{E}[\hat{\boldsymbol{w}}_{\text{ML}}] = \boldsymbol{w}$ and that it has a variance given by:

$$\text{Var}(\boldsymbol{w}_{\text{ML}}) = (\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})})^{-1} \tag{5.38}$$

In order to derive the bias of the MAP estimate, we need to calculate the expectation:

$$\begin{aligned} E[\hat{\boldsymbol{w}}_{\text{MAP}}] &= \mathbb{E}[\Lambda_m \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})}] \\ &= E\left[\Lambda_m \boldsymbol{X}^{(\text{train})\top} \left(\boldsymbol{X}^{(\text{train})} \boldsymbol{w} + \epsilon \right) \right] \\ &= \Lambda_m \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} \right) + \Lambda_m \boldsymbol{X}^{(\text{train})\top} E[\epsilon] \\ &= \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} + \lambda_0^{-1} \boldsymbol{I} \right)^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w}, \end{aligned} \tag{5.39}$$

We see that while the expected value of the ML estimate is the true parameter value $\boldsymbol{w}$ (i.e. the parameters that we assume generated the data); the expected

value of the MAP estimate is a weighted average of $\boldsymbol{w}$ and the prior mean $\boldsymbol{\mu}$. We comput the bias as:

$$\begin{aligned}\text{Bias}(\hat{\boldsymbol{w}}_{\text{MAP}}) &= E[\hat{\boldsymbol{w}}_{\text{MAP}}] - \boldsymbol{w} \\ &= -\left(\lambda_0 \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \boldsymbol{I}\right)^{-1}\boldsymbol{w}.\end{aligned}$$

Since the bias is not zero, we can conclude that the MAP estimate is biased, and as expected we can see that as the variance of the prior $\lambda_0 \to \infty$, the bias tends to zero. As the variance of the prior $\lambda_0 \to \mathbf{0}$, the bias tends to $\boldsymbol{w}$.

In order to compute the variance, we use the identity $\text{Var}(\hat{\theta}) = \mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2$. So before computing the variance we need to compute $\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right]$:

$$\begin{aligned}\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right] &= \mathbb{E}\Big[\Lambda_m \boldsymbol{X}^{(\text{train})\top}\hat{\boldsymbol{y}}^{(\text{train})}\boldsymbol{y}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\Big] \\ &= \mathbb{E}\Big[\Lambda_m \boldsymbol{X}^{(\text{train})\top}\left(\boldsymbol{X}^{(\text{train})}\boldsymbol{w} + \boldsymbol{\epsilon}\right)\left(\boldsymbol{X}^{(\text{train})}\boldsymbol{w} + \boldsymbol{\epsilon}\right)^{\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\Big] \\ &= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w}\boldsymbol{w}^{\top}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &\quad + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\mathbb{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\top}\right]\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w}\boldsymbol{w}^{\top}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &\quad + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &= E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} + \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\end{aligned}$$

With $\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right]$ thus computed, the variance of the MAP estimate of our linear regression model is given by:

$$\begin{aligned}\text{Var}(\hat{\boldsymbol{w}}_{\text{MAP}}) &= \mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}\hat{\boldsymbol{w}}_{MAP}^{\top}\right] - \mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}\right]\mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}^{\top}\right] \\ &= E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m - E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} \\ &= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &= \left(\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \lambda_0^{-1}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} \\ &\qquad \times \left(\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \lambda_0^{-1}\boldsymbol{I}\right)^{-1} \qquad (5.40)\end{aligned}$$

It is perhaps difficult to compare Eqs. 5.38 and 5.40. But if we assume that $\boldsymbol{w}$ is one-dimensional (along with $\boldsymbol{x}$), it becomes a bit easier to see that, as long as λ_0 is bounded, then $\text{Var}(\hat{w}_{\text{ML}}) = \frac{1}{\sum_{i=1}^{m} x_i^2} > \text{Var}(\hat{w}_{\text{MAP}}) = \frac{\lambda_0 \sum_{i=1}^{m} x_i^2}{\left(1+\lambda_0 \sum_{i=1}^{m} x_i^2\right)^2}$.

From the above analysis we can see that the role of the prior in the MAP estimate is to trade increased bias for a reduction in variance. The goal, of

course, is to try to avoid overfitting. The incurred bias is a consequence of the reduction in model capacity caused by limiting the space of hypotheses to those with significant probability density under the prior.

Many regularized estimation strategies, such as maximum likelihood learning regularized with weight decay, can be interpreted as making the MAP approximation to Bayesian inference. This view applies when the regularization consists of adding an extra term to the objective function that corresponds to $\log p(\theta)$. Not all such regularizer terms correspond to MAP Bayesian inference. For example, some regularizer terms may not be the logarithm of a probability distribution. Other regularization terms depend on the data, which of course a prior probability distribution is not allowed to do.

5.10 Supervised Learning

Supervised learning algorithms are, roughly speaking, learning algorithms that learn to associate some input with some output, given a training set of examples of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$.

5.10.1 Probabilistic Supervised Learning

Most supervised learning algorithms in this book are based on estimating a probability distribution $p(y \mid \boldsymbol{x})$. We can do this simply by using maximum conditional likelihood estimation (Sec. 5.8.1) – or just maximum likelihood for short – to find the best parameter vector $\boldsymbol{\theta}$ for a parametric family of distributions $p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

We have already seen that linear regression corresponds to the family $p(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y \mid \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{I})$. We can generalize linear regression to the classification scenario by defining a different family of probability distributions. If we have two classes, class 0 and class 1, then we need only specify the probability of one of these classes. The probability of class 1 determines the probability of class 0, because these two values must add up to 1.

The normal distribution over real-valued numbers that we used for linear regression is parameterized in terms of a mean. Any value we supply for this mean is valid. A distribution over a binary variable is slightly more complicated, because its mean must always be between 0 and 1. One way to solve this problem is to use the logistic sigmoid function to squash the output of the linear function into the interval (0, 1) and interpret that value as a probability:

$$p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}).$$

This approach is known as *logistic regression* (a somewhat strange name since we use the model for classification rather than regression).

In the case of linear regression, we were able to find the optimal weights by solving the normal equations. Logistic regression is somewhat more difficult. There is no closed-form solution for its optimal weights. Instead, we must search for them by maximizing the log-likelihood. We can do this by minimizing the negative log-likelihood (NLL) using gradient descent.

This same strategy can be applied to essentially any supervised learning problem, by writing down a parametric family of probability of conditional distributions over the right kind of input and output variables.

5.10.2 Support Vector Machines

One of the most influential approaches to supervised learning is the support vector machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995). This model is similar to logistic regression in that it is driven by a linear function $\boldsymbol{w}^\top \boldsymbol{x} + b$. Unlike logistic regression, the support vector machine does not provide probabilities, but only outputs a class identity.

One key innovation associated with support vector machines is the *kernel trick*. The kernel trick consists of observing that many machine learning algorithms can be written exclusively in terms of dot products between examples. For example, it can be shown that the linear function used by the support vector machine can be re-written as

$$\boldsymbol{w}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)}$$

where $\boldsymbol{x}^{(i)}$ is a training example and $\boldsymbol{\alpha}$ is a vector of coefficients. Rewriting the learning algorithm this way allows us to replace $\boldsymbol{x}$ by the output of a given feature function $\phi(\boldsymbol{x})$ and the dot product with a function $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$ called a *kernel*.

We can then make predictions using the function

$$f(x) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}). \tag{5.41}$$

This function is linear in the space that ϕ maps to, but non-linear as a function of $\boldsymbol{x}$.

The kernel trick is powerful for two reasons. First, it allows us to learn models that are non-linear as a function of $\boldsymbol{x}$ using convex optimization techniques that are guaranteed to converge efficiently. This is only possible because we consider ϕ fixed and only optimize $\boldsymbol{\alpha}$, i.e., the optimization algorithm can view the decision function as being linear in a different space. Second, the kernel function k need not be implemented in terms of explicitly applying the ϕ mapping and then applying

the dot product. The dot product in ϕ space might be equivalent to a non-linear but computationally less expensive operation in $\boldsymbol{x}$ space. For example, we could design an *infinite-dimensional* feature mapping $\phi(\boldsymbol{x})$ over the non-negative integers. Suppose that this mapping returns a vector containing $\boldsymbol{x}$ ones followed by infinitely many zeros. Explicitly constructing this mapping, or taking the dot product between two such vectors, costs infinite time and memory. But we can write a kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$ that is exactly equivalent to to this infinite-dimensional dot product. The most commonly used kernel is the *Gaussian kernel*

$$k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2 I) \tag{5.42}$$

where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the standard normal density. This kernel corresponds to the dot product $k(\boldsymbol{u}, \boldsymbol{v}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x})$ on an infinite-dimensional feature space ϕ and also has an interpretation as a similarity function, acting like a kind of template matching.

Support vector machines are not the only algorithm that can be enhanced using the kernel trick. Many linear models can be enhanced in this way. This category of algorithms is known as *kernel machines* or *kernel methods*.

A major drawback to kernel machines is that the cost of learning the $\boldsymbol{\alpha}$ coefficients is quadratic in the number of training examples. A related problem is that the cost of evaluating the decision function is linear in the number of training examples, because the i-th example contributes a term $\alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$ to the decision function. Support vector machines are able to mitigate this by learning an $\boldsymbol{\alpha}$ vector that contains mostly zeros. Classifying a new example then requires evaluating the kernel function only for the training examples that have non-zero α_i. These training examples are known as *support vectors*. Another major drawback of common kernel machines (such as those using the Gaussian kernel) is more statistical and regards their difficulty in generalizing to complex variations far from the training examples, as discussed in Section 5.13.

The analysis of the statistical limitations of support vector machines with general purpose kernels like the Gaussian kernels actually motivated the rebirth of neural networks through deep learning. Support vector machines and other kernel machines have often been viewed as a competitor to deep learning (though some deep networks can in fact be interpreted as support vector machines with learned kernels). The current deep learning renaissance began when deep networks were shown to outperform support vector machines on the MNIST benchmark dataset (Hinton *et al.*, 2006). One of the main reasons for the current popularity of deep learning relative to support vector machines is the fact that the cost of training kernel machines usually scales quadratically with the number of examples in the training set. For a deep network of fixed size, the memory cost of training is constant with respect to training set size (except for the memory

needed to store the examples themselves) and the runtime of a single pass through the training set is linear in training set size. These asymptotic results meant that kernelized SVMs dominated while datasets were small, but deep models currently dominate now that datasets are large.

5.11 Unsupervised Learning

Whereas supervised learning is geared at a very specific task such as predicting a variable $\mathbf{y}$ given a variable $\mathbf{x}$ from the observed data, *unsupervised learning* tries to extract more general-purpose statistical structure from the data. In fact, several more advanced forms unsupervised learning can be thought of as the rather general task of extracting all the possible information from the observed data. Let us call the observed data $\mathbf{z}$ (which could correspond to a pair $(\mathbf{x}, \mathbf{y})$ or maybe to observing just $\mathbf{x}$ alone), corresponding to the joint observation of many individual variables z_1, z_2 , z_3, then some unsupervised learning procedures basically amount to learning what it takes to be able to predict any subset of the variables given any other subset.

Unsupervised learning can also be seen as the types of learning algorithms that extract potentially useful information from *inputs* x *alone*, without any pre-specified label y. The objective is then to use the extracted information later, for some supervised task involving the prediction of y given x. This would be a form of *semi-supervised learning*, in which we combine unlabeled examples (with only examples of x) with labeled examples (with (x, y) pairs).

Learning a representation of data A classic unsupervised learning task is to find the 'best' representation of the data. By 'best' we can mean different things, but generally speaking we are looking for a representation that preserves as much information about x as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than x itself.

There are multiple ways of defining a *simpler* representation, some of the most common include lower dimensional representations, sparse representations and independent representations. Low-dimensional representations attempt to compress as much information about x as possible in a smaller representation. Sparse representations generally embed the dataset into a high-dimensional representation[5] where the number of non-zero entries is small. This results in an overall structure of the representation that tends to distribute data along the axes of the representation space. Independent representations attempt to *disentangle* the sources of variation underlying the data distribution such that the

[5] sparse representations often use over-complete representations: the representation dimension is greater than the original dimensionality of the data.

dimensions of the representation are statistically independent.

Of course these three criteria are certainly not mutually exclusive. Low-dimensional representations often yield elements that are more-or-less mutually independent. This happens because the pressure to encode as much information about the data x as possible into a low-dimensional representation drives the elements of this representation to be more independent. Any dependency between the variables in $f(x)$ is evidence of redundancy and implies that the representation $f(x)$ could have captured more information about x.

The notion of representation is one of the central themes of deep learning and therefore one of the central themes in this book. Chapter 16 discusses some of the qualities we would like in our learned representations, along with specific representation learning algorithms more powerful than the simple one presented next, Principal Components Analysis.

5.11.1 Principal Components Analysis

In the remainder of this section we will consider one of the most widely used unsupervised learning methods: Principle Components Analysis (PCA). PCA is an orthogonal, linear transformation of the data that projects it into a representation where the elements are uncorrelated (shown in Figure 5.7).

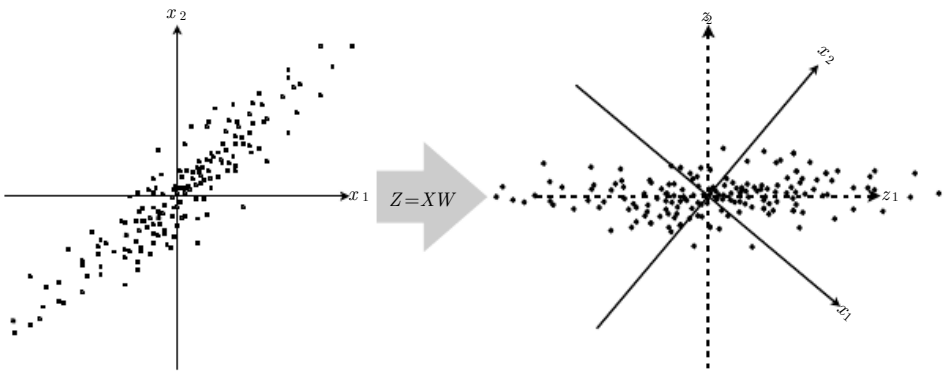


Figure 5.7: Illustration of the data representation learned via PCA.

In section 2.12, we saw that we could learn a one-dimensional representation that best reconstructs the original data (in the sense of mean squared error) and that this representation actually corresponds to the first principal component of the data. Thus we can use PCA as a simple and effective dimensionality reduction method that preserves as much of the information in the data as possible (again, as measured by least-squares reconstruction error). In the following, we will take a look at other properties of the PCA representation. Specifically, we will study how the PCA representation can be said to decorrelate the original data

representation $\boldsymbol{X}$.

Let us consider the $n \times m$-dimensional design matrix $\boldsymbol{X}$. We will assume that the data has a mean of zero, $\mathbb{E}[\boldsymbol{x}] = \mathbf{0}$. If this is not the case, the data can easily be centered (mean removed). The unbiased sample covariance matrix associated with $\boldsymbol{X}$ is given by:

$$\text{Var}[\boldsymbol{x}] = \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X} \tag{5.43}$$

One important aspect of PCA is that it finds a representation (through linear transformation) $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$ where $\text{Var}[\boldsymbol{z}]$ is diagonal. To do this, we will make use of the singular value decomposition (SVD) of $\boldsymbol{X}$: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top$, where $\boldsymbol{\Sigma}$ is an $n \times m$-dimensional rectangular diagonal matrix with the singular values of $\boldsymbol{X}$ on the main diagonal, $\boldsymbol{U}$ is an $n \times n$ matrix whose columns are orthonormal (i.e. unit length and orthogonal) and $\boldsymbol{W}$ is an $m \times m$ matrix also composed of orthonormal column vectors.

Using the SVD of $\boldsymbol{X}$, we can re-express the variance of $\boldsymbol{X}$ as:

$$\begin{aligned}
\text{Var}[\boldsymbol{x}] &= \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X} && (5.44)\\
&= \frac{1}{n-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.45)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.46)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top, && (5.47)
\end{aligned}$$

where we use the orthonormality of $\boldsymbol{U}$ ($\boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{I}$) and define $\boldsymbol{\Sigma}^2$ as an $m \times m$-dimensional diagonal matrix with the squares of the singular values of $\boldsymbol{X}$ on the diagonal, i.e. the ith diagonal elements is given by $\Sigma^2_{i,i}$. This shows that if we take $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, we can ensure that the covariance of $\boldsymbol{z}$ is diagonal as required.

$$\begin{aligned}
\text{Var}[\boldsymbol{z}] &= \frac{1}{n-1}\boldsymbol{Z}^\top \boldsymbol{Z} && (5.48)\\
&= \frac{1}{n-1}\boldsymbol{W}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{W} && (5.49)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{W}^\top \boldsymbol{\Sigma}^2 \boldsymbol{W}\boldsymbol{W}^\top && (5.50)\\
&= \frac{1}{n-1}\boldsymbol{\Sigma}^2 && (5.51)
\end{aligned}$$

Similar to our analysis of the variance of $\boldsymbol{X}$ above, we exploit the orthonormality of $\boldsymbol{W}$ (i.e., $\boldsymbol{W}^\top\boldsymbol{W} = \boldsymbol{I}$). Our use of SVD to solve for the PCA components of $\boldsymbol{X}$ (i.e. elements of $\boldsymbol{z}$) reveals an interesting connection to the eigen-decomposition

of a matrix related to $\boldsymbol{X}$. Specifically, the columns of $\boldsymbol{W}$ are the eigenvectors of the $n \times n$-dimensional matrix $\boldsymbol{X}^\top \boldsymbol{X}$.

The above analysis shows that when we project the data $\boldsymbol{x}$ to $\boldsymbol{z}$, via the linear transformation $\boldsymbol{W}$, the resulting representation has a diagonal covariance matrix (as given by $\boldsymbol{\Sigma}^2$) which immediately implies that the individual elements of $\boldsymbol{z}$ are mutually uncorrelated.

This ability of PCA to transform data into a representation where the elements are mutually uncorrelated is a very important property of PCA. It is a simple example of a representation that attempt to *disentangle the unknown factors of variation* underlying the data. In the case of PCA, this *disentangling* takes the form of finding a rotation of the input space (mediated via the transformation $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{z}$, as illustrated in Fig. 5.7. While correlation is an important category of dependency between elements of the data, we are also interested in learning representations that *disentangle* more complicated forms of feature dependencies. For this, we will need more than what can be done with a simple linear transformation. These issues are discussed below in Sec. 5.13 and later in detail in Chapter 16.

5.12 Weakly Supervised Learning

Weakly supervised learning is another class of learning methods that stands between supervised and unsupervised learning. It refers to a setting where the datasets consists of $(\boldsymbol{x}, y)$ pairs, as in supervised learning, but where the labels y are either unreliably present (i.e. with missing values) or noisy (i.e. where the label given is not the true label).

Methods for working with weakly labeled data have recently grown in importance due to the—largely untapped—potential for using large quantities of readily available weakly labeled data in a transfer learning paradigm to help solve problems where large, clean datasets are hard to come-by. The Internet has become a major source of this kind of noisy data.

For example, although we would like to train a computer vision system with labels indicating the presence and location of every object (and which pixels correspond to which object) in every image, such labeling is very human-labor intensive. Instead, we want to take advantage of images for which only the main object is identified, like the ImageNet dataset (Deng *et al.*, 2009), or worse, of video for which some general and high-level semantic spoken caption is approximately temporally aligned with the corresponding frames of the video, like the DVS data (Descriptive Video service) which has recently been released (Torabi *et al.*, 2015).

5.13 The Curse of Dimensionality and Statistical Limitations of Local Generalization

The number of variable configurations grows exponentially with the number of variables, i.e., with dimension, which brings up a statistical form of the *curse of dimensionality*, introduced in the next section. Many non-parametric learning algorithms, such as kernel machines with a Gaussian kernel, rely on a simple preference over functions which corresponds to an assumption of smoothness or local constancy. As argued in Section 5.13.2 that follows, this allows these algorithms to generalize near the training examples, but does not allow them to generalize in a non-trivial way far from them: the number of ups and downs that can be captured is limited by the number of training examples. This is particularly problematic with high-dimensional data, because of the curse of dimensionality. In order to reduce that difficulty, researchers have introduced the idea of dimensionality reduction and manifold learning, introduced in Section 5.13.3. This motivates the introduction of additional a priori about the task to be learned, as well as the idea of learning to better represent the data, the topic which constitutes the bulk of the rest of this book.

5.13.1 The Curse of Dimensionality

Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high. This phenomenon is known as the *curse of dimensionality*. Of particular concern is that the number of possible distinct configurations of the variables of interest increases exponentially as the dimensionality increases.

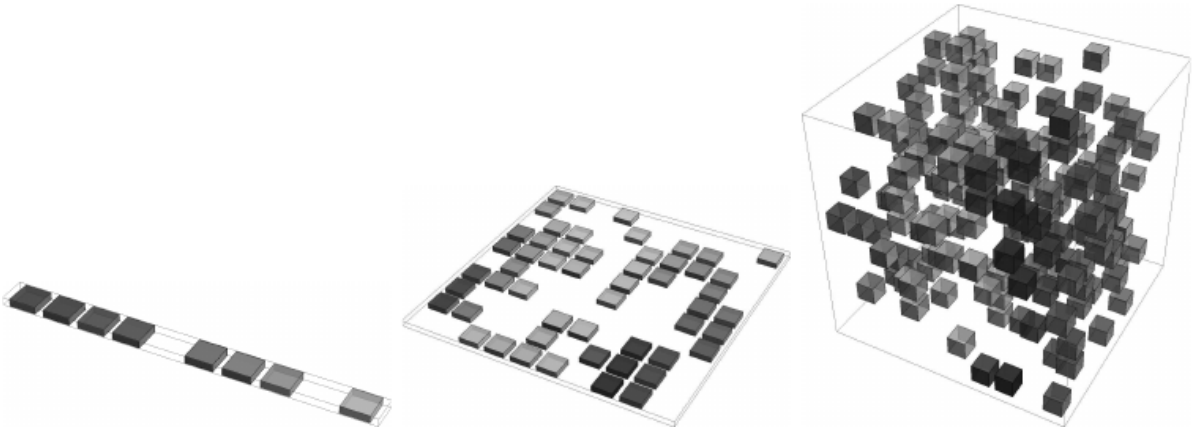


Figure 5.8: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. In the figure we first consider one-dimensional data (left), i.e., one variable for which we only care to distinguish 10 regions of interest. With enough examples falling within each of these regions (cells, in the figure), learning algorithms can easily generalize correctly, i.e., estimate the value of the target function within each region (and possibly interpolate between neighboring regions). With 2 dimensions (center), but still caring to distinguish 10 different values of each variable, we need to keep track of up to 10×10=100 regions, and we need at least that many examples to cover all those regions. With 3 dimensions (right) this grows to $10^3 = 1000$ regions and at least that many examples. For d dimensions and V values to be distinguished along each axis, it looks like we need $O(V^d)$ regions and examples. This is an instance of the curse of dimensionality. However, note that if the data distribution is concentrated on a smaller set of regions, we may actually not need to cover all the possible regions, only those where probability is non-negligible. *Figure graciously provided by, and with authorization from, Nicolas Chapados.*

The curse of dimensionality rears its ugly head in many places in computer science, and especially so in machine learning.

One challenge posed by the curse of dimensionality is a statistical challenge. As illustrated in Figure 5.8, a statistical challenge arises because the number of possible configurations of the variables of interest is much larger than the number of training examples. To understand the issue, let us consider that the input space is organized into a grid, like in the figure. In low dimensions we can describe this space with a low number of grid cells that are mostly occupied by the data. The least we can assume about the data generating distribution is that our learner should provide the same answer to two examples falling in the same grid cell. It is a form of local constancy assumption, a notion that we develop further in the next section. When generalizing to a new data point, we can usually tell what to do simply by inspecting the training examples that lie in the same cell as the new input. For example, if estimating the probability density at some point $\boldsymbol{x}$, we can just return the number of training examples in the same unit volume cell as $\boldsymbol{x}$, divided by the total number of training examples. If we wish to

classify an example, we can return the most common class of training examples in the same cell. If we are doing regression we can average the target values observed over the examples in that cell. But what about the cells for which we have seen no example? Because in high-dimensional spaces the number of configurations is going to be huge, much larger than our number of examples, most configurations will have no training example associated with it. How could we possibly say something meaningful about these new configurations? A simple answer is to extend the local constancy assumption into a smoothness assumption, as explained next.

5.13.2 Smoothness and Local Constancy A Priori Preference

As argued previously, and especially in high-dimensional spaces (because of the curse of dimensionality introduced above), machine learning algorithms need priors, i.e., a preference over the space of solutions, in order to generalize to new configurations not seen in the training set. The specification of these preferences includes the choice of model family, as well as any regularizer or other aspects of the algorithm that influence the final outcome of training. We consider here a particular family of preferences which underlie many classical machine learning algorithms, and which we call the *smoothness prior* or the *local constancy prior*. We find that when the function to be learned has many ups and downs, and this is typically the case in high-dimensional spaces because of the curse of dimensionality (see above), then the smoothness prior is insufficient to achieve good generalization. We argue that more assumptions are needed in order to generalize better, in this setting. Deep learning algorithms typically introduce such additional assumptions. This starts with the classical multi-layer neural networks studied in the next chapter (Chapter 6), and in Chapter 16 we return to the advantages that representation learning, distributed representations and depth can bring towards generalization, even in high-dimensional spaces.

Different smoothness or local constancy priors can be expressed, but what they basically say is that the target function or distribution of interest f^* is such that

$$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon) \tag{5.52}$$

for most configurations $\boldsymbol{x}$ and small change ϵ. In other words, if we know a good answer (e.g., for an example $\boldsymbol{x}$) then that answer is probably good in the neighborhood of $\boldsymbol{x}$, and if we have several good answers in some neighborhood we would combine them (e.g., by some form of averaging or interpolation) to produce an answer that agrees with them as much as possible.

An example of locally constant family of functions is the *histogram*, which breaks the input space into a number of *distinguishable regions* or "bins" in which any $\boldsymbol{x}$ can fall, and produces a constant output within each region. Another

example of piecewise constant learned function is what we obtain with *k-nearest neighbors* predictors, where $f(\boldsymbol{x})$ is constant in some region R containing all the points $\boldsymbol{x}$ that have the same set of k nearest neighbors from the training set. If we are doing classification and k=1, $f(\boldsymbol{x})$ is just the output class associated with the nearest neighbor of $\boldsymbol{x}$ in the training set. If we are doing regression, $f(\boldsymbol{x})$ is the average of the outputs associated with the k nearest neighbors of $\boldsymbol{x}$. Note that in both cases, for $k = 1$, the number of distinguishable regions of cannot be more than the number of training examples. See Murphy (2012); Bishop (2006) or other machine learning textbooks for more material on histograms and nearest-neighbor classifiers.

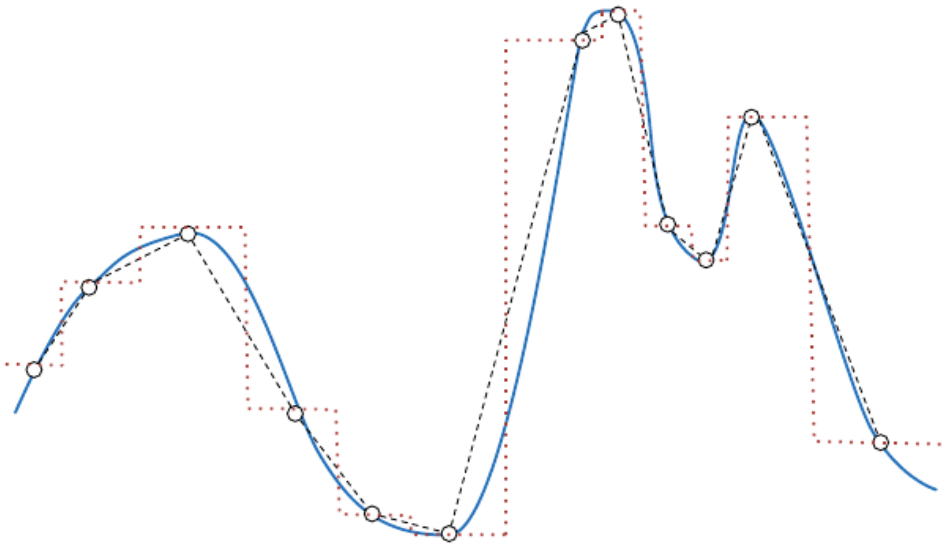


Figure 5.9: Illustration of interpolation and kernel-based methods, which construct a smooth function by interpolating in various ways between the training examples (circles), which act like knot points controlling the shape of the implicit regions that separate them as well as the values to output within each region. Depending on the type of kernel, one obtains a piecewise constant (histogram-like, in dotted red), a piecewise linear (dashed black) or a smoother kernel (bold blue). The underlying assumption is that the target function is as smooth or locally as constant as possible. This assumption allows to *generalize locally*, i.e., to extend the answer known at some point $\boldsymbol{x}$ to nearby points, and this works very well so long as, like in the figure, there are enough examples to cover most of the ups and downs of the target function.

To obtain even more smoothness, we can *interpolate* between neighboring training examples, as illustrated in Figure 5.9. For example, *non-parametric kernel density estimation methods* and *kernel regression* methods construct a learned function f of the form of Eq. 5.41 for classification or regression, or alternatively,

e.g., in the Parzen regression estimator, of the form

$$f(\boldsymbol{x}) = b + \sum_{i=1}^{n} \alpha_i \frac{k(\boldsymbol{x}, \boldsymbol{x}^{(i)})}{\sum_{j=1}^{n} k(\boldsymbol{x}, x^{(j)})}.$$

If the kernel function k is discrete (e.g. 0 or 1), then this can include the above cases where f is piecewise constant and a discrete set of regions (no more than one per training example) can be distinguished. However, better results can often be obtained if k is smooth, e.g., the Gaussian kernel from Eq. 5.42. With k a *local kernel* (Bengio *et al.*, 2006a; Bengio and LeCun, 2007b; Bengio, 2009)[6] , we can think of each $\boldsymbol{x}^{(i)}$ as a *template* and the kernel function as a *similarity function* that *matches a template and a test example*.

With the Gaussian kernel, we do not have a piecewise constant function but instead a continuous and smooth function. In fact, the choice of k can be shown to correspond to a particular form of smoothness. Equivalently, we can think of many of these estimators as the result of smoothing the *empirical distribution* by convolving it with a function associated with the kernel, e.g., the Gaussian kernel density estimator is the empirical distribution convolved with the Gaussian density.

Although in classical non-parametric estimators the α_i of Eq. 5.41 are fixed (e.g. to $1/n$ for density estimation and to $y^{(i)}$ for supervised learning from examples $(\boldsymbol{x}^{(i)}, y^{(i)})$), they can be optimized, and this is the basis of more modern non-parametric kernel methods (Schölkopf and Smola, 2002) such as the Support Vector Machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995) (see also Section 5.10.2).

However, as illustrated in Figure 5.9, even though these smooth kernel methods generalize better, the main thing that has changed is that one can basically interpolate between the neighboring examples, in some space associated with the kernel. One can then think of the training examples as control knots which locally specify the shape of each region and the associated output.

[6] i.e., with $k(\boldsymbol{u}, \boldsymbol{v})$ large when $\boldsymbol{u} = \boldsymbol{v}$ and decreasing as they get farther apart

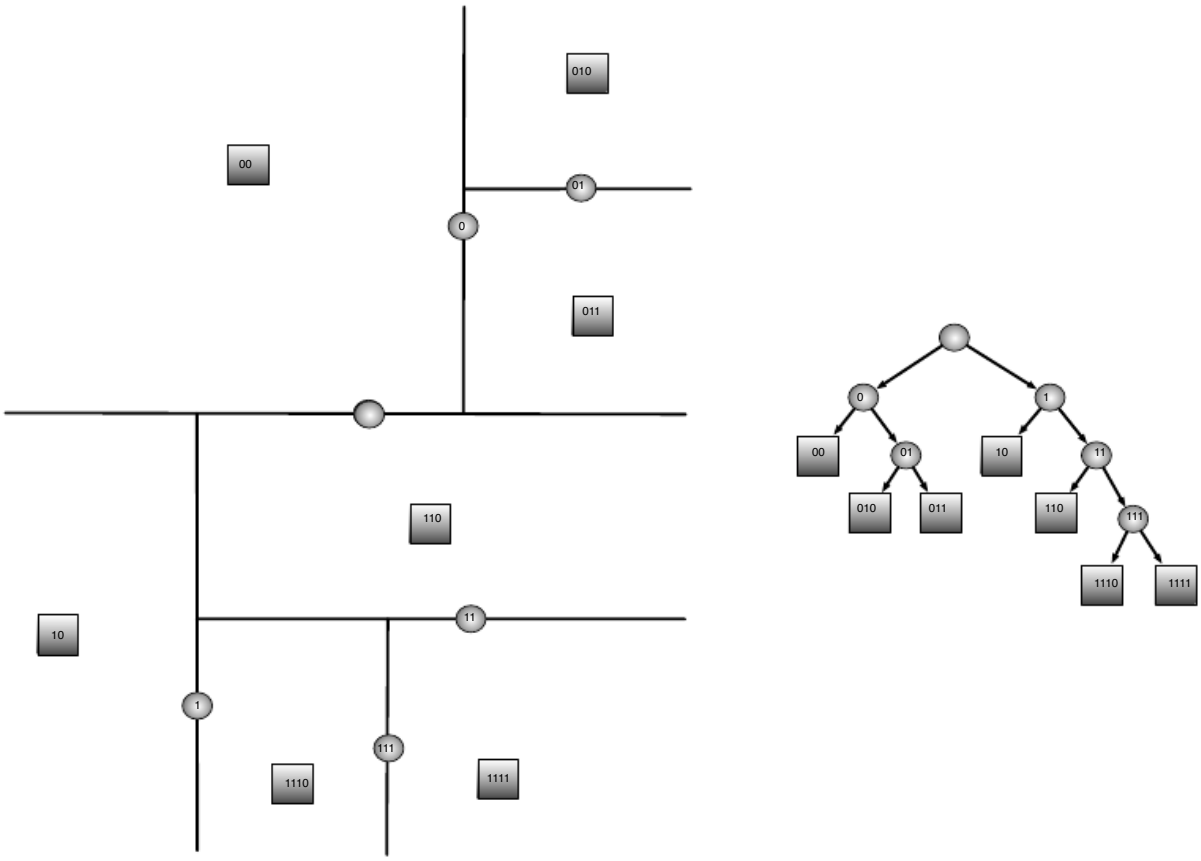


Figure 5.10: Decision tree (right) and how it cuts the input space into regions, with a constant output in each region (left). Each node of the tree (circle or square) is associated with a region (the entire space for the root node, with the empty string identifier). Internal nodes (circles) split their region in two, in this case (the most common) via an axis aligned cut. Leaf nodes (squares) are associated with an "answer", such as the average target output for the training examples that fall in the corresponding region. Each node is displayed with a binary string identifier corresponding to its position in the tree, obtained by adding a bit to its parent (0=choose left or top, 1=choose right or bottom). Note that the result is a piecewise-constant function, and note how the number of regions (pieces) cannot be greater than the number of examples, hence it is not possible to learn a function that has more ups and downs than the number of training examples.

Another type of non-parametric learning algorithm that also breaks the input space into regions and has separate parameters for each region is the *decision tree* (Breiman *et al.*, 1984) and its many variants. We give a brief account here and illustrate decision trees in Figure 5.10, but please refer as needed to Murphy (2012); Bishop (2006) or other machine learning textbooks for more material on decision trees. Each node of the decision tree is associated with a region in the input space, and internal nodes breaks that region into one sub-region for each child of the node (typically using an axis-aligned cut). Typically, a constant output $f(n(\boldsymbol{x}))$ is returned by the decision tree predictor for any $\boldsymbol{x}$ falling in the

region associated with a particular leaf node $n(\boldsymbol{x})$. Because each example only informs the region in which it falls about which output to produce, *one cannot have more regions than training examples.* If the target function can be well approximated by cutting the input space into N regions (with a different answer in each region), then at least N examples are needed (and a multiple of N is needed to achieve some level of statistical confidence in the predicted output). All this is also true if the tree is used for density estimation (the output is simply an estimate of the density within the region, which can be obtained by the ratio of the number of training examples in the region by the region volume) or whether a non-constant (e.g. linear) predictor is associated with each leaf (then more examples are needed within each leaf node, but the relationship between number of regions and number of examples remains linear). We examine below how this may hurt the generalization ability of decision trees and other learning algorithms that are based only on the smoothness or local constancy priors, when the input is high-dimensional, i.e., because of the curse of dimensionality.

In all cases, the smoothness assumption (Eq. 5.52) allows the learner to *generalize locally.* Since we assume that the target function obeys $f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon)$ most of the time for small ϵ, we can generalize the empirical distribution (or the $(\boldsymbol{x}, y)$ training pairs) to the neighborhood of the training examples. If $(\boldsymbol{x}^{(i)}, y^{(i)})$ is a supervised (input,target) training example, then we expect $f^*(\boldsymbol{x}^{(i)}) \approx y^{(i)}$, and therefore if $\boldsymbol{x}$ is a near neighbor of $\boldsymbol{x}^{(i)}$, we expect that $f^*(\boldsymbol{x}) \approx y^{(i)}$. By considering more neighbors, we can obtain better generalization, by better executing the smoothness assumption.

In general, to distinguish $O(N)$ regions in input space, all of these methods require $O(N)$ examples (and typically there are $O(N)$ parameters associated with the $O(N)$ regions). This is illustrated in Figure 5.11 in the case of a nearest-neighbor or clustering scenario, where each training example can be used to define one region. Is there a way to represent a complex function that has many more regions to be distinguished than the number of training examples? Clearly, assuming only smoothness of the underlying function will not allow a learner to do that. For example, imagine that the target function is a kind of checkerboard, i.e., with a lot of variations, but a simple structure to them, and imagine that the number of training examples is substantially less than the number of black and white regions. Based on local generalization and the smoothness or local constancy prior, we could get the correct answer within a constant-colour region, but we could not correctly predict the checkerboard pattern. The only thing that an example tells us, with this prior, is that nearby points should have the same colour, and the only way to get the checkerboard right is to cover all of its cells with at least one example.

The smoothness assumption and the associated non-parametric learning algo-

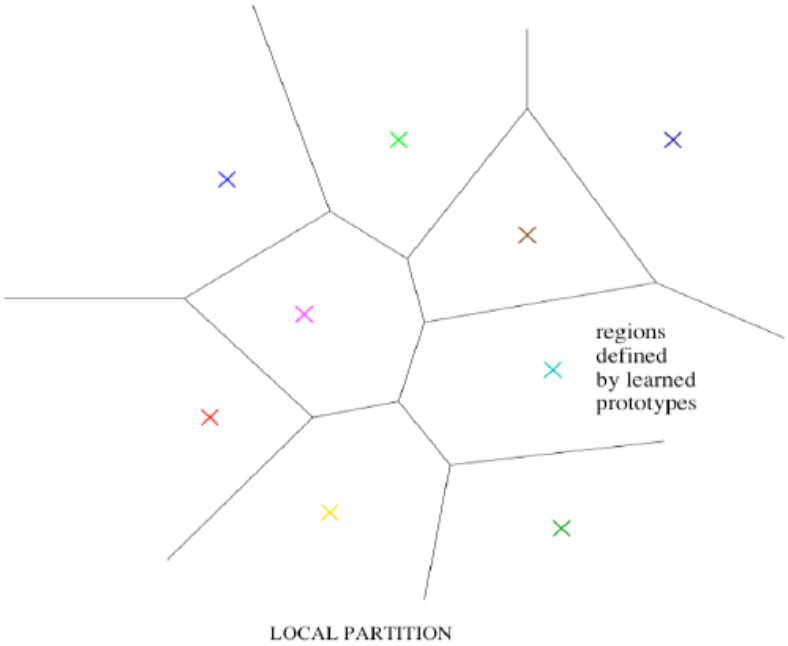


Figure 5.11: Illustration of how non-parametric learning algorithms that exploit only the smoothness or local constancy priors typically break up the input space into regions, with examples in those regions being used both to define the region boundaries and what the output should be within each region. The figure shows the case of clustering or 1-nearest-neighbor classifiers, for which each training example (cross of a different color) defines a region or a template (here, the different regions form a Voronoi tessellation). The number of these contiguous regions cannot grow faster than the number of training examples. In the case of a decision tree, the regions are recursively obtained by axis-aligned cuts within existing regions, but for these and for kernel machines with a local kernel (such as the Gaussian kernel), the same property holds, and generalization can only be *local*: each training example only informs the learner about how to generalize in some neighborhood around it.

rithms work extremely well *so long as there are enough examples to cover most of the ups and downs of the target function.* This is generally true when the function to be learned is smooth enough, which is typically the case for low-dimensional data. And if it is not very smooth (we want to distinguish a huge number of regions compared to the number of examples), is there any hope to generalize well?

Both of these questions are answered positively in Chapter 16. The key insight is that a very large number of regions, e.g., $O(2^N)$, can be defined with $O(N)$ examples, so long as we introduce some dependencies between the regions via additional priors about the underlying data generating distribution. In this way, we can actually generalize non-locally (Bengio and Monperrus, 2005; Bengio *et al.*, 2006b). A neural network can actually learn a checkerboard pattern. Similarly, some recurrent neural networks can learn the n-bit parity (at least for some not too large values of n). Of course we could also solve the checkerboard task by making a much stronger assumption, e.g., that the target function is periodic. However, neural networks can generalize to a much wider variety of structures, and indeed our AI tasks have structure that is much too complex to be limited to periodicity, so we want learning algorithms that embody more general-purpose assumptions. The core idea in deep learning is that we assume that the data was generated by the *composition of factors* or features, potentially at multiple levels in a hierarchy. These apparently mild assumptions allow an exponential gain in the relationship between the number of examples and the number of regions that can be distinguished, as discussed in Chapter 16. Priors that are based on compositionality, such as arising from learning distributed representations and from a deep composition of representations, can give an exponential advantage, which can hopefully counter the exponential curse of dimensionality. Chapter 16 discusses these questions from the angle of representation learning and the objective of *disentangling the underlying factors of variation.*

5.13.3 Manifold Learning and the Curse of Dimensionality

We consider here a particular type of machine learning task called *manifold learning.* Although they have been introduced to reduce the curse of dimensionality. We will argue that they allow one to visualize and highlight how the smoothness prior is not sufficient to generalize in high-dimensional spaces. Chapter 17 is devoted to the manifold perspective on representation learning and goes in much greater details in this topic as well as in actual manifold learning algorithms based on neural networks.

A *manifold* is a connected region, i.e., a set of points, associated with a neighborhood around each point, which makes it locally look like a Euclidean space. The notion of neighbor implies the existence of transformations that can be ap-

plied to *move on the manifold* from one position to a neighboring one. Although there is a formal mathematical meaning to this term, in machine learning it tends to be used more loosely to talk about a connected set of points that can be well approximated by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space. Each dimension corresponds to a local direction of variation, i.e., moving along the manifold in some direction. The manifolds we talk about in machine learning are subsets of points, also called a submanifold, of the embedding space (which is also a manifold).

Manifold learning algorithms assume that the data distribution is concentrated in a small number of dimensions, i.e., that the set of high-probability configurations can be approximated by a low-dimensional manifold. Figure 5.7 (left) illustrates a distribution that is concentrated near a linear manifold (the manifold is along a 1-dimensional straight line). Manifold learning was introduced in the case of continuous-valued data and the unsupervised learning setting, although this probability concentration idea can be generalized to both discrete data and the supervised learning setting: the key assumption remains that probability mass is highly concentrated.

Is this assumption reasonable? It seems to be true for almost all of the AI tasks such as those involving images, sounds, and text. To be convinced of this we will invoke (a) the observation that probability mass is concentrated and (b) the observed objects can generally be transformed into other plausible configurations via some small changes (which indicates a notion of direction of variation while staying on the "manifold"). For (a), consider that if the assumption of probability concentration was false, then sampling uniformly at random from in the set of all configurations (e.g., uniformly in $\mathbb{R}^n$) should produce probable (data-like) configurations reasonably often. But this is not what we observe in practice. For example, generate pixel configurations for an image by independently picking the grey level (or a binary 0 vs 1) for each pixel. What kind of images do you get? You get "white noise" images, that look like the old television sets when no signal is coming in, as illustrated in Figure 5.12 (left). What is the probability that you would obtain something that looks like a natural image, with this procedure? Almost zero, because the set of probable configurations (near the manifold of natural images) occupies a very small volume out of the total set of pixel configurations. Similarly, if you generate a document by picking letters randomly, what is the probability that you will get a meaningful English-language text? Almost zero, again, because most of the long sequences of letters do not correspond to a natural language sequence: the distribution of natural language sequences occupies a very small volume in the total space of sequences of letters.

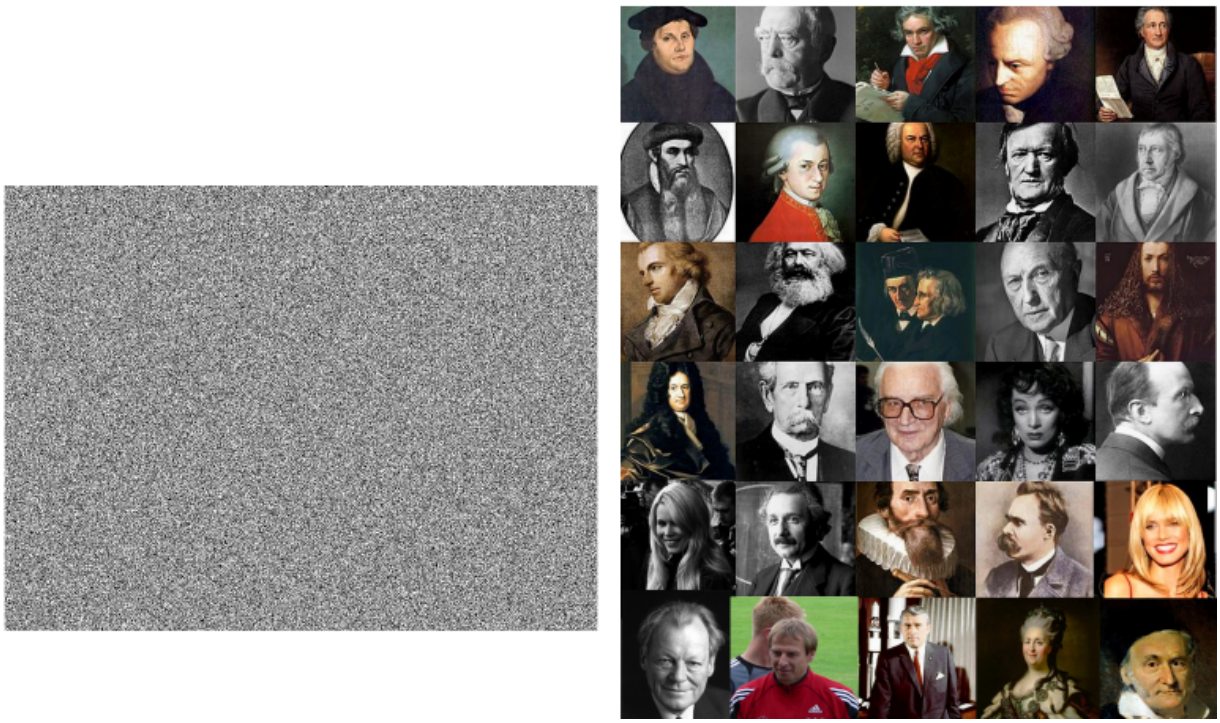


Figure 5.12: Sampling images uniformly at random, e.g., by randomly picking each pixel according to a uniform distribution, gives rise to white noise images such as illustrated on the left. Although there is a non-zero probability to generate something that looks like a natural image (like those on the right), that probability is exponentially tiny (exponential in the number of pixels!). This suggests that natural images are very "special", and that they occupy a tiny volume of the space of images.

The above thought experiments, which are in agreement with the many experimental results of the manifold learning literature, e.g. (Cayton, 2005; Narayanan and Mitter, 2010; Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004), clearly establish that for a large class of datasets of interest in AI, the *manifold hypothesis* is true: the data generating distribution concentrates in a small number of dimensions, as in the cartoon of Figure 17.4, from Chapter 17. That chapter explores the relationships between representation learning and manifold learning: if the data distribution concentrates on a smaller number of dimensions, then we can think of these dimensions as natural coordinates for the data, and we can think of representation learning algorithms as ways to map the input space to a new and often lower-dimensional space which captures the leading dimensions of variation present in the data as axes or dimensions of the representation.

An initial hope of early work on manifold learning (Roweis and Saul, 2000; Tenenbaum *et al.*, 2000) was to *reduce the effect of the curse of dimensionality*, by first reducing the data to a lower dimensional representation (e.g. mapping

(x_1, x_2) to z_1 in Figure 5.7 (right)), and then applying ordinary machine learning in that transformed space. This dimensionality reduction can be achieved by learning a transformation (generally non-linear, unlike with PCA introduced in Section 5.11.1) of the data that is *invertible for most training examples*, i.e., that keeps the information in the input example. It is only possible to reconstruct input examples from their low-dimensional representation because they lie on a lower-dimensional manifold, of course. This is basically how *auto-encoders* (Chapter 15) are trained.

The hope was that by non-linearly projecting the data in a new space of lower dimension, we would reduce the curse of dimensionality by only looking at relevant dimensions, i.e., a smaller set of regions of interest (cells, in Figure 5.8). This can indeed be the case, however, as discussed in Chapter 17, the manifolds can be highly curved and have a very large number of twists, requiring still a very large number of regions to be distinguished (every up and down of each corner of the manifold). And even if we were to reduce the dimensionality of an input from 10000 (e.g. 100×100 binary pixels) to 100, 2^{100} is still too large to hope covering with a training set. This still rules out the use of purely local generalization (i.e., the smoothness prior only) to model such manifolds, as discussed in Chapter 17 around Figure 17.4 and 17.5. It may also be that although the effective dimensionality of the data could be small, some examples could fall outside of the main manifold and that we do not want to systematically lose that information. A *sparse representation* then becomes a possible way to represent data that is mostly low-dimensional, although occasionally occupying more dimensions. This can be achieved with a high-dimensional representation whose elements are 0 most of the time. We can see that the effective dimension (the number of non-zeros) then can change depending on where we are in input space, which can be useful. Sparse representations are discussed in Section 15.8.

The next part of the book introduces specific deep learning algorithms that aim at discovering representations that are useful for some task, i.e., trying to extract the directions of variations that matter for the task of interest, often in a supervised setting. The last part of the book concentrates more on unsupervised representation learning algorithms, which attempt to capture all of the directions of variation that are salient in the data distribution.

Part II

Modern Practical Deep Networks

This part of the book summarizes the state of modern deep learning as it is used to solve practical applications.

Deep learning has a long history and many aspirations. Several approaches have been proposed that have yet to entirely bear fruit. Several ambitious goals have yet to be realized. These less-developed branches of deep learning appear in the final part of the book.

This part focuses only on those approaches that are essentially working technologies that are already used heavily in industry.

Modern deep learning provides a very powerful framework for supervised learning. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large model and dataset of labeled training examples. Other tasks, that can not be described as associating one vector to another, or that are difficult enough that to do them a person would require time to think and reflect, remain beyond the scope of deep learning for now.

This part of the book describes the core parametric function approximation technology that is behind nearly all modern practical applications of deep learning. Our description includes details such as, how to efficiently model specific kinds of inputs, how to process image inputs with convolutional networks as well as how to process sequence inputs with recurrent and recursive networks. Moreover, we provide guidance for how to preprocess the data for various tasks and how to choose the values of the various settings that govern the behavior of these algorithms.

These chapters are the most important for a practitioner – someone who wants to begin implementing and using deep learning algorithms to solve real-world problems today.

Chapter 6

Feedforward Deep Networks

6.1 From Fixed Features to Learned Features

In Chapter 5 we considered linear regression, linear classifiers and logistic regression, and introduced kernel machines, all of which involve a fixed set of features on which a linear predictor is trained. These models perform non-linear transformations of data, but the non-linear part is pre-defined. How can we learn non-linear transformations of the data that create a new feature space? How can we automate feature learning? This is what neural networks with hidden layers allow us to do.

Feedforward supervised neural networks were among the first and most successful learning algorithms (Rumelhart *et al.*, 1986e,c). They are also called deep networks, multi-layer perceptron (MLP), or simply neural networks and the vanilla architecture with a single hidden layer is illustrated in Figure 6.1. A deeper version is obtained by simply having more hidden layers. Each hidden layer corresponds to a new learned representation of the input vector, trained towards some objective, such as making it easier to produce desired answers in output.

MLPs can learn powerful non-linear transformations: in fact, with enough hidden units they can represent arbitrarily complex but smooth functions (see Section 6.5). This is achieved by composing simpler but still non-linear learned transformations. By transforming the data non-linearly into a new space, a classification problem that was not linearly separable (not solvable by a linear classifier) can become separable, as illustrated in Figure 6.2.

6.1.1 Estimating Conditional Statistics

To gently move from linear predictors to non-linear ones, let us consider the squared error loss function studied in the previous chapter, where the learning

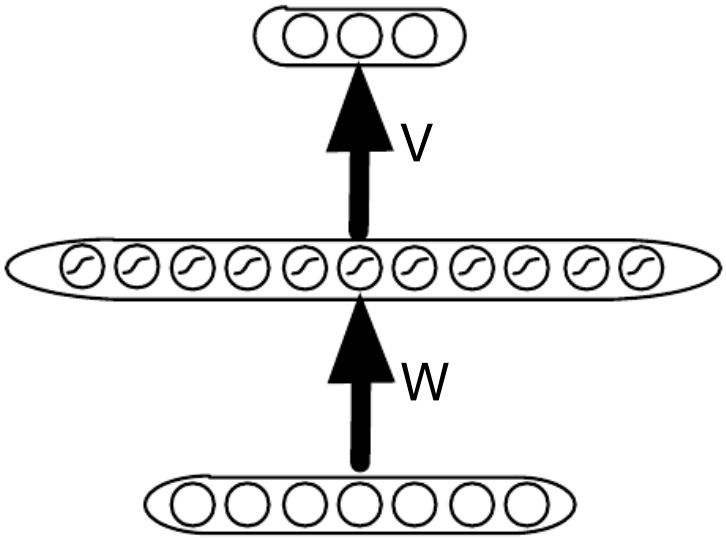


Figure 6.1: Vanilla (shallow) MLP, with one sigmoid hidden layer, computing vector-valued hidden unit vector $\boldsymbol{h} = \text{sigmoid}(\boldsymbol{c} + \boldsymbol{W}\boldsymbol{x})$ with weight matrix $\boldsymbol{W}$ and offset vector $\boldsymbol{c}$. The output vector is obtained via another learned affine transformation $\hat{\boldsymbol{y}} = \boldsymbol{b} + \boldsymbol{V}\boldsymbol{h}$, with weight matrix $\boldsymbol{V}$ and output offset vector $\boldsymbol{b}$. The vector of hidden unit values $\boldsymbol{h}$ provides a new set of features, i.e., a new representation, derived from the raw input $\boldsymbol{x}$.

task is the estimation of the expected value of $\mathbf{y}$ given $\mathbf{x}$. In the context of linear regression, the conditional expectation of y is used as the mean of a Gaussian distribution that we fit with maximum likelihood. We can generalize linear regression to regression via any function f by defining the mean squared error of f:

$$\mathbb{E}[||\boldsymbol{y} - f(\boldsymbol{x})||^2]$$

where the expectation is over the training set during training, and over the data generating distribution to obtain generalization error.

We can generalize its interpretation beyond the case where f is linear or affine, uncovering an interesting property: minimizing it yields an estimator of the conditional expectation of the output variable $\mathbf{y}$ given the input variable $\mathbf{x}$, i.e.,

$$\arg\min_{f \in \mathbb{H}} \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})}[||\boldsymbol{y} - f(\boldsymbol{x})||^2] = \mathbb{E}_{p(\mathbf{x},y)}[\mathbf{y}|\boldsymbol{x}]. \tag{6.1}$$

provided that our set of function $\mathbb{H}$ contains $\mathbb{E}_{p(\mathbf{x},\mathbf{y})}[\mathbf{y} \mid \boldsymbol{x}]$. (If you would like to work out the proof yourself, it is easy to do using calculus of variations, which we describe in Chapter 19.4.2).

Similarly, we can generalize conditional maximum likelihood (introduced in Section 5.8.1) to other distributions than the Gaussian, as discussed below when defining the objective function for MLPs.

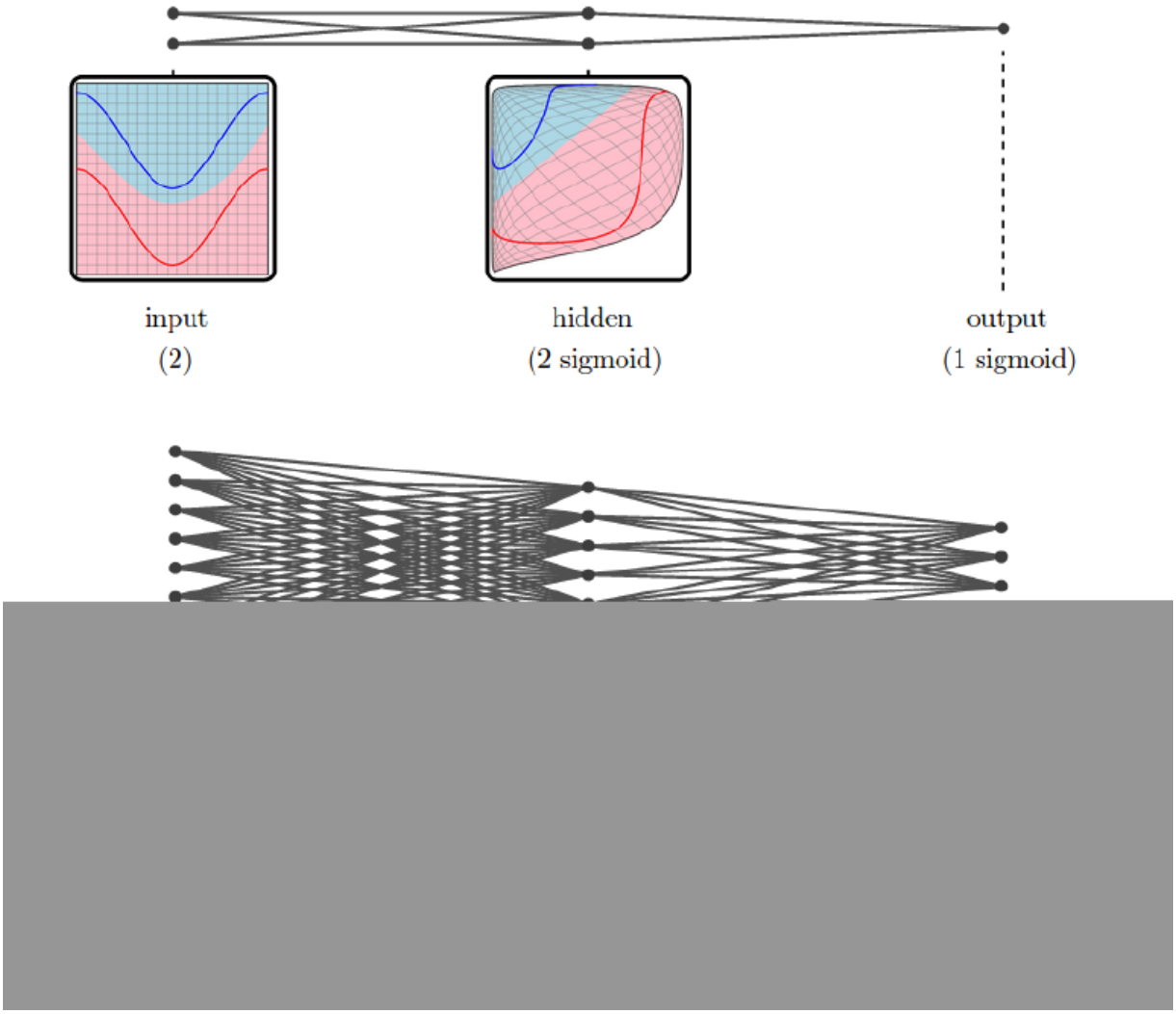


Figure 6.2: Each layer of a trained neural network non-linearly transforms its input, distorting the space so that the task becomes easier to perform, e.g., linear classification in the new feature space, in the above figures. Top: a vanilla neural network with 2 hidden units in its single hidden layer can transform the 2-D input space (shown in the blue and pink square figure) so that the examples from the two classes (shown as the points on the red and blue curves) become linearly separable (the red and blue curves correspond to the "manifold" near which examples concentrate, while the red and pink areas correspond to the regions where the neural network classifies an input as either blue or red). Bottom: with a larger hidden layer (100 here), the MNIST digit images (with $28 \times 28 = 784$ pixels) can be transformed so that the classes (each shown with a different color) can be much more easily classified by the output layer (over 10 digit categories). Both figures are reproduced with permission by Chris Olah from `http://colah.github.io/`, where many more insightful visualizations can be found.