Understanding Deep Learning

Simon J.D. Prince

May 1, 2024

If you enjoy this book, here are four ways you can help me:

1. Spread the word via social media. Posts in languages other than English particularly welcome. Tag me on LinkedIn or X and I'll probably say hi.
2. Write me an Amazon review. Preferably positive, but all publicity is good publicity...
3. Send me comments (see bottom of this page). I reply to everything eventually.
4. Buy a copy. I took 18 months completely off work to write this book and ideally I'd like to make minimum wage (or better) for this time. Also, I'd like to write a second edition, but I need to sell enough copies to do this. Thanks!

The most recent version of this document can be found at http://udlbook.com.

I would really appreciate help improving this document. No detail too small! Please contact me with suggestions, factual inaccuracies, ambiguities, questions, and errata via github or by e-mail at udlbookmail@gmail.com.

This book is dedicated to Blair, Calvert, Coppola, Ellison, Faulkner, Kerpatenko, Morris, Robinson, Sträussler, Wallace, Waymon, Wojnarowicz, and all the others whose work is even more important and interesting than deep learning.

Contents

Preface **ix**

Acknowledgements **xi**

1 Introduction **1**
1.1 Supervised learning . . . 1
1.2 Unsupervised learning . . . 7
1.3 Reinforcement learning . . . 11
1.4 Ethics . . . 12
1.5 Structure of book . . . 15
1.6 Other books . . . 15
1.7 How to read this book . . . 16

2 Supervised learning **17**
2.1 Supervised learning overview . . . 17
2.2 Linear regression example . . . 18
2.3 Summary . . . 22

3 Shallow neural networks **25**
3.1 Neural network example . . . 25
3.2 Universal approximation theorem . . . 29
3.3 Multivariate inputs and outputs . . . 30
3.4 Shallow neural networks: general case . . . 33
3.5 Terminology . . . 35
3.6 Summary . . . 36

4 Deep neural networks **41**
4.1 Composing neural networks . . . 41
4.2 From composing networks to deep networks . . . 43
4.3 Deep neural networks . . . 45
4.4 Matrix notation . . . 48
4.5 Shallow vs. deep neural networks . . . 49
4.6 Summary . . . 52

5 Loss functions **56**
5.1 Maximum likelihood . . . 56
5.2 Recipe for constructing loss functions . . . 60
5.3 Example 1: univariate regression . . . 61
5.4 Example 2: binary classification . . . 64
5.5 Example 3: multiclass classification . . . 67
5.6 Multiple outputs . . . 69
5.7 Cross-entropy loss . . . 71
5.8 Summary . . . 72

6 Fitting models **77**
6.1 Gradient descent . . . 77
6.2 Stochastic gradient descent . . . 83
6.3 Momentum . . . 86
6.4 Adam . . . 88
6.5 Training algorithm hyperparameters . . . 91
6.6 Summary . . . 91

7 Gradients and initialization **96**
7.1 Problem definitions . . . 96
7.2 Computing derivatives . . . 97
7.3 Toy example . . . 100
7.4 Backpropagation algorithm . . . 103
7.5 Parameter initialization . . . 107
7.6 Example training code . . . 111
7.7 Summary . . . 111

8 Measuring performance **118**
8.1 Training a simple model . . . 118
8.2 Sources of error . . . 120
8.3 Reducing error . . . 124
8.4 Double descent . . . 127
8.5 Choosing hyperparameters . . . 132
8.6 Summary . . . 133

9 Regularization **138**
9.1 Explicit regularization . . . 138
9.2 Implicit regularization . . . 141
9.3 Heuristics to improve performance . . . 144
9.4 Summary . . . 154

10 Convolutional networks **161**
10.1 Invariance and equivariance . . . 161
10.2 Convolutional networks for 1D inputs . . . 163
10.3 Convolutional networks for 2D inputs . . . 170

10.4 Downsampling and upsampling . . . 171
10.5 Applications . . . 174
10.6 Summary . . . 179

11 Residual networks **186**
11.1 Sequential processing . . . 186
11.2 Residual connections and residual blocks . . . 189
11.3 Exploding gradients in residual networks . . . 192
11.4 Batch normalization . . . 192
11.5 Common residual architectures . . . 195
11.6 Why do nets with residual connections perform so well? . . . 199
11.7 Summary . . . 199

12 Transformers **207**
12.1 Processing text data . . . 207
12.2 Dot-product self-attention . . . 208
12.3 Extensions to dot-product self-attention . . . 213
12.4 Transformer layers . . . 215
12.5 Transformers for natural language processing . . . 216
12.6 Encoder model example: BERT . . . 219
12.7 Decoder model example: GPT3 . . . 222
12.8 Encoder-decoder model example: machine translation . . . 226
12.9 Transformers for long sequences . . . 227
12.10 Transformers for images . . . 228
12.11 Summary . . . 232

13 Graph neural networks **240**
13.1 What is a graph? . . . 240
13.2 Graph representation . . . 243
13.3 Graph neural networks, tasks, and loss functions . . . 245
13.4 Graph convolutional networks . . . 248
13.5 Example: graph classification . . . 251
13.6 Inductive vs. transductive models . . . 252
13.7 Example: node classification . . . 253
13.8 Layers for graph convolutional networks . . . 256
13.9 Edge graphs . . . 260
13.10 Summary . . . 261

14 Unsupervised learning **268**
14.1 Taxonomy of unsupervised learning models . . . 268
14.2 What makes a good generative model? . . . 269
14.3 Quantifying performance . . . 271
14.4 Summary . . . 273

15 Generative Adversarial Networks **275**

15.1 Discrimination as a signal . . . 275
15.2 Improving stability . . . 280
15.3 Progressive growing, minibatch discrimination, and truncation . . . 286
15.4 Conditional generation . . . 288
15.5 Image translation . . . 290
15.6 StyleGAN . . . 295
15.7 Summary . . . 297

16 Normalizing flows **303**
16.1 1D example . . . 303
16.2 General case . . . 306
16.3 Invertible network layers . . . 308
16.4 Multi-scale flows . . . 316
16.5 Applications . . . 317
16.6 Summary . . . 320

17 Variational autoencoders **326**
17.1 Latent variable models . . . 326
17.2 Nonlinear latent variable model . . . 327
17.3 Training . . . 330
17.4 ELBO properties . . . 333
17.5 Variational approximation . . . 335
17.6 The variational autoencoder . . . 335
17.7 The reparameterization trick . . . 338
17.8 Applications . . . 339
17.9 Summary . . . 342

18 Diffusion models **348**
18.1 Overview . . . 348
18.2 Encoder (forward process) . . . 349
18.3 Decoder model (reverse process) . . . 355
18.4 Training . . . 356
18.5 Reparameterization of loss function . . . 360
18.6 Implementation . . . 362
18.7 Summary . . . 367

19 Reinforcement learning **373**
19.1 Markov decision processes, returns, and policies . . . 373
19.2 Expected return . . . 377
19.3 Tabular reinforcement learning . . . 381
19.4 Fitted Q-learning . . . 385
19.5 Policy gradient methods . . . 388
19.6 Actor-critic methods . . . 393
19.7 Offline reinforcement learning . . . 394
19.8 Summary . . . 395

20 Why does deep learning work? 401
20.1 The case against deep learning . . . 401
20.2 Factors that influence fitting performance . . . 402
20.3 Properties of loss functions . . . 406
20.4 Factors that determine generalization . . . 410
20.5 Do we need so many parameters? . . . 414
20.6 Do networks have to be deep? . . . 417
20.7 Summary . . . 418

21 Deep learning and ethics 420
21.1 Value alignment . . . 420
21.2 Intentional misuse . . . 426
21.3 Other social, ethical, and professional issues . . . 428
21.4 Case study . . . 430
21.5 The value-free ideal of science . . . 431
21.6 Responsible AI research as a collective action problem . . . 432
21.7 Ways forward . . . 433
21.8 Summary . . . 434

A Notation 436

B Mathematics 439
B.1 Functions . . . 439
B.2 Binomial coefficients . . . 441
B.3 Vector, matrices, and tensors . . . 442
B.4 Special types of matrix . . . 445
B.5 Matrix calculus . . . 447

C Probability 448
C.1 Random variables and probability distributions . . . 448
C.2 Expectation . . . 452
C.3 Normal probability distribution . . . 456
C.4 Sampling . . . 459
C.5 Distances between probability distributions . . . 459

Bibliography 462

Index 513

Preface

The history of deep learning is unusual in science. The perseverance of a small cabal of scientists, working over twenty-five years in a seemingly unpromising area, has revolutionized a field and dramatically impacted society. Usually, when researchers investigate an esoteric and apparently impractical corner of science or engineering, it remains just that — esoteric and impractical. However, this was a notable exception. Despite widespread skepticism, the systematic efforts of Yoshua Bengio, Geoffrey Hinton, Yann LeCun, and others eventually paid off.

The title of this book is "Understanding Deep Learning" to distinguish it from volumes that cover coding and other practical aspects. This text is primarily about the *ideas* that underlie deep learning. The first part of the book introduces deep learning models and discusses how to train them, measure their performance, and improve this performance. The next part considers architectures that are specialized to images, text, and graph data. These chapters require only introductory linear algebra, calculus, and probability and should be accessible to any second-year undergraduate in a quantitative discipline. Subsequent parts of the book tackle generative models and reinforcement learning. These chapters require more knowledge of probability and calculus and target more advanced students.

The title is also partly a joke — *no-one* really understands deep learning at the time of writing. Modern deep networks learn piecewise linear functions with more regions than there are atoms in the universe and can be trained with fewer data examples than model parameters. It is neither obvious that we should be able to fit these functions reliably nor that they should generalize well to new data. The penultimate chapter addresses these and other aspects that are not yet fully understood. Regardless, deep learning will change the world for better or worse. The final chapter discusses AI ethics and concludes with an appeal for practitioners to consider the moral implications of their work.

Your time is precious, and I have striven to curate and present the material so you can understand it as efficiently as possible. The main body of each chapter comprises a succinct description of only the most essential ideas, together with accompanying illustrations. The appendices review all mathematical prerequisites, and there should be no need to refer to external material. For readers wishing to delve deeper, each chapter has associated problems, Python notebooks, and extensive background notes.

Writing a book is a lonely, grinding, multiple-year process and is only worthwhile if the volume is widely adopted. If you enjoy reading this or have suggestions for improving it, please contact me via the accompanying website. I would love to hear your thoughts, which will inform and motivate subsequent editions.

Acknowledgments

Writing this book would not have been possible without the generous help and advice of these individuals: Kathryn Hume, Kevin Murphy, Christopher Bishop, Peng Xu, Yann Dubois, Justin Domke, Chris Fletcher, Yanshuai Cao, Wendy Tay, Corey Toler-Franklin, Dmytro Mishkin, Guy McCusker, Daniel Worrall, Paul McIlroy, Roy Amoyal, Austin Anderson, Romero Barata de Morais, Gabriel Harrison, Peter Ball, Alf Muir, David Bryson, Vedika Parulkar, Patryk Lietzau, Jessica Nicholson, Alexa Huxley, Oisin Mac Aodha, Giuseppe Castiglione, Josh Akylbekov, Alex Gougoulaki, Joshua Omilabu, Alister Guenther, Joe Goodier, Logan Wade, Joshua Guenther, Kylan Tobin, Benedict Ellett, Jad Araj, Andrew Glennerster, Giorgos Sfikas, Diya Vibhakar, Sam Mansat-Bhattacharyya, Ben Ross, Ivor Simpson, Gaurang Aggarwal, Shakeel Sheikh, Jacob Horton, Felix Rammell, Sasha Luccioni, Akshil Patel, Alessandro Gentilini, Kevin Mercier, Krzysztof Lichocki, Chuck Krapf, Brian Ha, Chris Kang, Leonardo Viotti, Kai Li, Himan Abdollahpouri, Ari Pakman, Giuseppe Antonio Di Luna, Dan Oneaţă, Conrad Whiteley, Joseph Santarcangelo, Brad Shook, Gabriel Brostow, Lei He, Ali Satvaty, Romain Sabathé, Qiang Zhou, Prasanna Vigneswaran, Siqi Zheng, Stephan Grein, Jonas Klesen, Giovanni Stilo, Huang Bokai, Kevin McGuinness, Qiang Sun, Zakaria Lotfi, Yifei Lin, Sylvain Bouix, Alex Pitt, Stephane Chretien, Robin Liu, Bian Li, Adam Jones, Marcin Świerkot, Tommy Löfstedt, Eugen Hotaj, Fernando Flores-Mangas, Tony Polichroniadis, Pietro Monticone, Rohan Deepak Ajwani, Menashe Yarden Einy, Robert Gevorgyan, Thilo Stadelmann, Gui JieMiao, Botao Zhu, Mohamed Elabbas, Satya Krishna Gorti, James Elder, Helio Perroni Filho, Xiaochao Qu, Jaekang Shin, Joshua Evans, Robert Dobson, Shibo Wang, Edoardo Zorzi, Stanisław Jastrzębski, Pieris Kalligeros, Matt Hewitt, Zvika Haramaty, Ted Mavroidis, Nikolaj Kuntner, Amir Yorav, Masoud Mokhtari, Xavier Gabaix, Marco Garosi, Vincent Schönbach, Avishek Mondal, Victor S.C. Lui, Sumit Bhatia, Julian Asilis, Hengchao Chen, Siavash Khallaghi, Csaba Szepesvári, Mike Singer, Mykhailo Shvets, Abdalla Ibrahim, Stefan Hell, Ron Raphaeli, Diogo Tavares, Aristotelis Siozopoulos, Jianrui Wu, Jannik Münz, Penn Mackintosh, Shawn Hoareau, Qianang Zhou, Emma Li, Charlie Groves, Xiang Lingxiao, Trivikram Muralidharan, Rajat Binaykiya, Germán del Cacho Salvador, Alexey Bloudov, Paul Colognese, Bo Yang, Jani Monoses, Adenilson Arcanjo, Matan Golani, Emmanuel Onzon, Shenghui Yan, Kamesh Kompella, Julius Aka, Johannes Brunnemann, Varniethan Ketheeswaran, Alex Ostrovsky, Daniel Burbank, Gavrie Philipson, Roozbeh Ehsani, Len Spek, Christoph Brune, Mohammad Nosrati, Bian Li, Runqi Chen, Qifu Hu, Rasmi Elasmar, Ronaldo Butrus, Carles Mesado, Jeffrey Wolberg, Olivier Koch, Edoardo Lanari, Fanmin Shi, Neel Maniar, Maksym Taran, Falk Langhammer, Reinaldo Lepsch, Max Talberg, Vishal Jain, Christian Arnold, Charles Hill, Nikita Panin, Steven Dillmann, Suhas Mathur, Harris Abdul Majid, Guolong Lin, Charles Elkan, Benedict Kuester, Vladimir Ivanov, Mohammad-Hadi Sotoudeh, Daniel Enériz Orta, Ian Jeffrey, Kwok Chun, Yu Liu, Tom Vettenburg, Aravinda Perera, Daniel Gigliotti, Iftikhar Ramnandan, Adnan Siddiquei, Will Knottenbelt, Valerio Di Stefano, and Srikant Jayaraman.

I'm particularly grateful to Daniyar Turmukhambetov, Amedeo Buonanno, Andrea Panizza,

Mark Hudson, and Bernhard Pfahringer, who provided detailed comments on multiple chapters of the book. I'd like to especially thank Andrew Fitzgibbon, Konstantinos Derpanis, and Tyler Mills, who read the whole book and whose enthusiasm helped me complete this project. I'd also like to thank Neill Campbell and Özgür Şimşek, who hosted me at the University of Bath, where I taught a course based on this material for the first time. Finally, I'm extremely grateful to my editor Elizabeth Swayze for her frank advice throughout this process.

Chapter 12 (transformers) and chapter 17 (variational autoencoders) were first published as blogs for Borealis AI, and adapted versions are reproduced with permission of Royal Bank of Canada along with Borealis AI. I am grateful for their support in this endeavor. Chapter 16 (normalizing flows) is loosely based on the review article by Kobyzev et al. (2020), on which I was a co-author. I was very fortunate to be able to collaborate on Chapter 21 with Travis LaCroix from Dalhousie University, who was both easy and fun to work with, and who did the lion's share of the work.

Attribution

- Chessboard image in figure 1.13 adapted from http://tinyurl.com/yc2d54d4.
- Cogs image in figures 1.2, 1.4, 1.10 adapted from http://tinyurl.com/2c7tttr8.
- Penguin image in figures 19.1–19.5 and 19.6–19.9 adapted from http://tinyurl.com/ycx9je56.
- Fish image in figures 19.2–19.5, 19.7, 19.10–19.12 adapted from http://tinyurl.com/4ueyhtsu.

Chapter 1

Introduction

Artificial intelligence, or *AI*, is concerned with building systems that simulate intelligent behavior. It encompasses a wide range of approaches, including those based on logic, search, and probabilistic reasoning. *Machine learning* is a subset of AI that learns to make decisions by fitting mathematical models to observed data. This area has seen explosive growth and is now (incorrectly) almost synonymous with the term AI.

A *deep neural network* is a type of machine learning model, and when it is fitted to data, this is referred to as *deep learning*. At the time of writing, deep networks are the most powerful and practical machine learning models and are often encountered in day-to-day life. It is commonplace to translate text from another language using a *natural language processing* algorithm, to search the internet for images of a particular object using a *computer vision* system, or to converse with a digital assistant via a *speech recognition* interface. All of these applications are powered by deep learning.

As the title suggests, this book aims to help a reader new to this field understand the principles behind deep learning. The book is neither terribly theoretical (there are no proofs) nor extremely practical (there is almost no code). The goal is to explain the underlying *ideas*; after consuming this volume, the reader will be able to apply deep learning to novel situations where there is no existing recipe for success.

Machine learning methods can coarsely be divided into three areas: supervised, unsupervised, and reinforcement learning. At the time of writing, the cutting-edge methods in all three areas rely on deep learning (figure 1.1). This introductory chapter describes these three areas at a high level, and this taxonomy is also loosely reflected in the book's organization. Whether we like it or not, deep learning is poised to change our world, and this change will not all be positive. Hence, this chapter also contains brief primer on AI ethics. We conclude with advice on how to make the most of this book.

1.1 Supervised learning

Supervised learning models define a mapping from input data to an output prediction. In the following sections, we discuss the inputs, the outputs, the model itself, and what is meant by "training" a model.

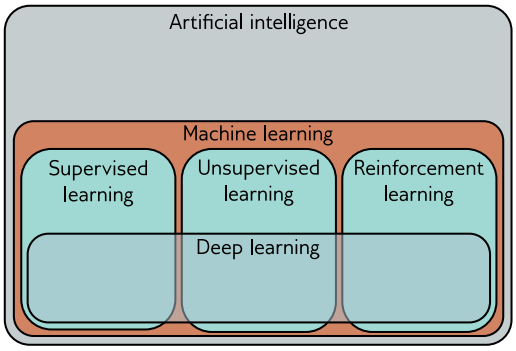


Figure 1.1 Machine learning is an area of artificial intelligence that fits mathematical models to observed data. It can coarsely be divided into supervised learning, unsupervised learning, and reinforcement learning. Deep neural networks contribute to each of these areas.

1.1.1 Regression and classification problems

Figure 1.2 depicts several regression and classification problems. In each case, there is a meaningful real-world input (a sentence, a sound file, an image, etc.), and this is encoded as a vector of numbers. This vector forms the model input. The model maps the input to an output vector which is then "translated" back to a meaningful real-world prediction. For now, we focus on the inputs and outputs and treat the model as a black box that ingests a vector of numbers and returns another vector of numbers.

The model in figure 1.2a predicts the price of a house based on input characteristics such as the square footage and the number of bedrooms. This is a *regression* problem because the model returns a continuous number (rather than a category assignment). In contrast, the model in 1.2b takes the chemical structure of a molecule as an input and predicts both the melting and boiling points. This is a *multivariate regression* problem since it predicts more than one number.

The model in figure 1.2c receives a text string containing a restaurant review as input and predicts whether the review is positive or negative. This is a *binary classification* problem because the model attempts to assign the input to one of two categories. The output vector contains the probabilities that the input belongs to each category. Figures 1.2d and 1.2e depict *multiclass classification* problems. Here, the model assigns the input to one of $N > 2$ categories. In the first case, the input is an audio file, and the model predicts which genre of music it contains. In the second case, the input is an image, and the model predicts which object it contains. In each case, the model returns a vector of size N that contains the probabilities of the N categories.

1.1.2 Inputs

The input data in figure 1.2 varies widely. In the house pricing example, the input is a fixed-length vector containing values that characterize the property. This is an example of *tabular data* because it has no internal structure; if we change the order of the inputs and build a new model, then we expect the model prediction to remain the same.

Conversely, the input in the restaurant review example is a body of text. This may be of variable length depending on the number of words in the review, and here input

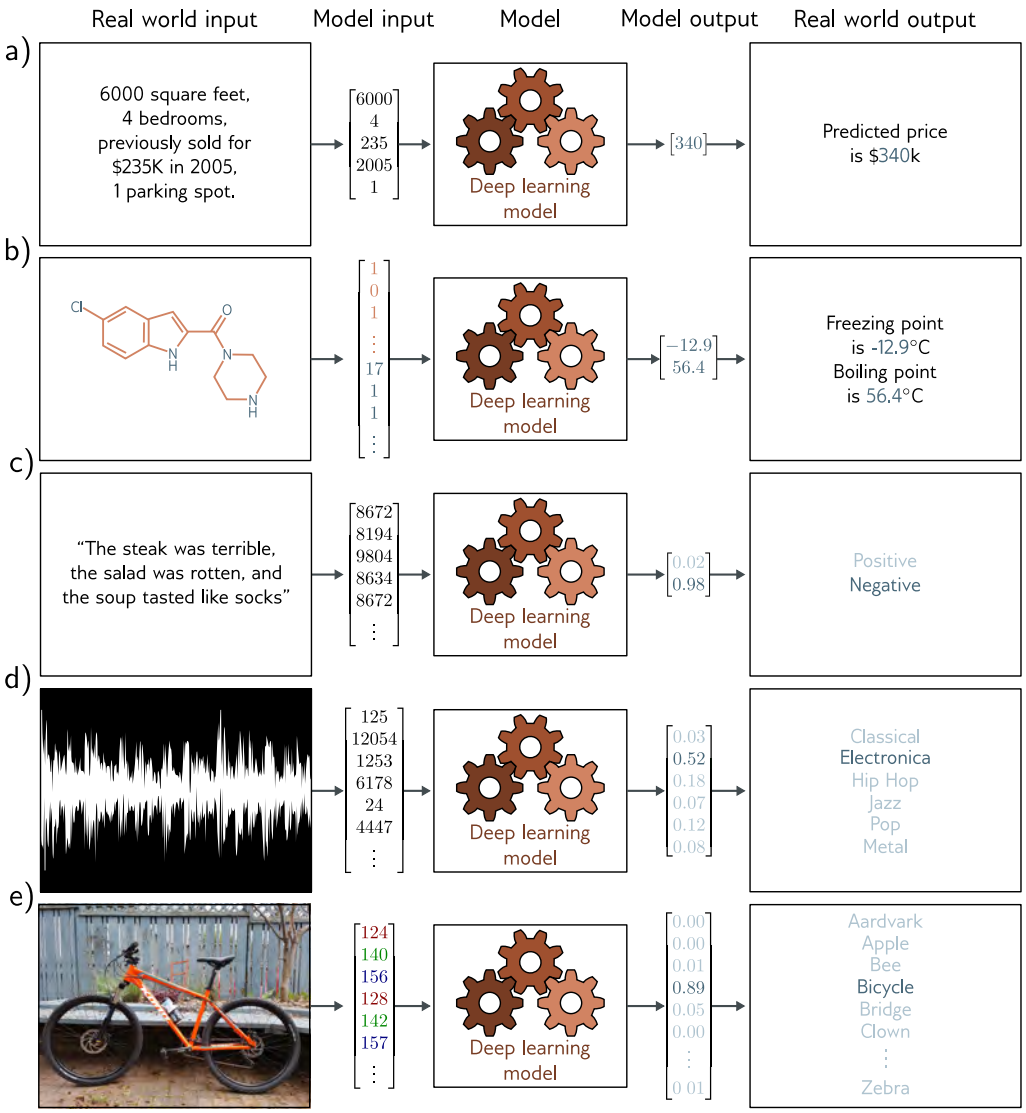


Figure 1.2 Regression and classification problems. a) This *regression* model takes a vector of numbers that characterize a property and predicts its price. b) This *multivariate regression* model takes the structure of a chemical molecule and predicts its melting and boiling points. c) This *binary classification* model takes a restaurant review and classifies it as either positive or negative. d) This *multiclass classification* problem assigns a snippet of audio to one of N genres. e) A second multiclass classification problem in which the model classifies an image according to which of N possible objects it might contain.

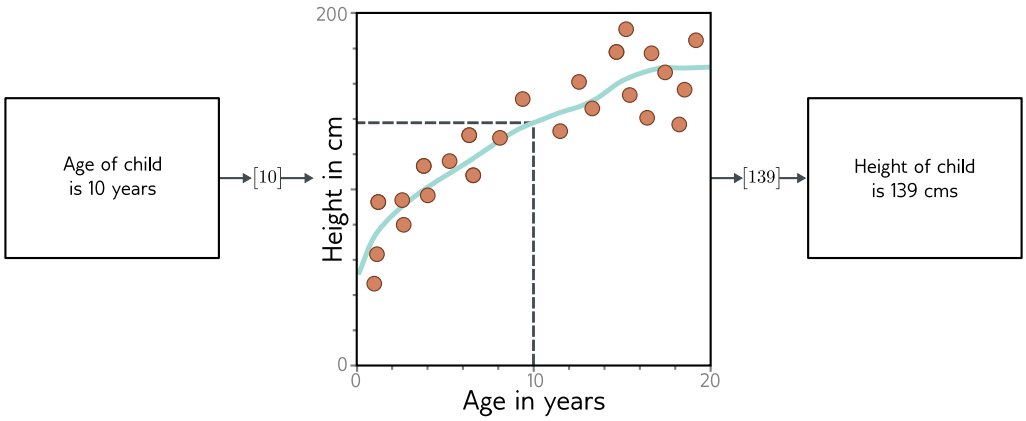


Figure 1.3 Machine learning model. The model represents a family of relationships that relate the input (age of child) to the output (height of child). The particular relationship is chosen using training data, which consists of input/output pairs (orange points). When we train the model, we search through the possible relationships for one that describes the data well. Here, the trained model is the cyan curve and can be used to compute the height for any age.

order is important; *my wife ate the chicken* is not the same as *the chicken ate my wife*. The text must be encoded into numerical form before passing it to the model. Here, we use a fixed vocabulary of size 10,000 and simply concatenate the word indices.

For the music classification example, the input vector might be of fixed size (perhaps a 10-second clip) but is very high-dimensional. Digital audio is usually sampled at 44.1 kHz and represented by 16-bit integers, so a ten-second clip consists of $441,000$ integers. Clearly, supervised learning models will have to be able to process sizeable inputs. The input in the image classification example (which consists of the concatenated RGB values at every pixel) is also enormous. Moreover, its structure is naturally two-dimensional; two pixels above and below one another are closely related, even if they are not adjacent in the input vector.

Finally, consider the input for the model that predicts the melting and boiling points of the molecule. A molecule may contain varying numbers of atoms that can be connected in different ways. In this case, the model must ingest both the geometric structure of the molecule and the constituent atoms to the model.

1.1.3 Machine learning models

Until now, we have treated the machine learning model as a black box that takes an input vector and returns an output vector. But what exactly is in this black box? Consider a model to predict the height of a child from their age (figure 1.3). The machine learning

model is a mathematical equation that describes how the average height varies as a function of age (cyan curve in figure 1.3). When we run the age through this equation, it returns the height. For example, if the age is 10 years, then we predict that the height will be 139 cm.

More precisely, the model represents a family of equations mapping the input to the output (i.e., a family of different cyan curves). The particular equation (curve) is chosen using *training data* (examples of input/output pairs). In figure 1.3, these pairs are represented by the orange points, and we can see that the model (cyan line) describes these data reasonably. When we talk about *training* or *fitting* a model, we mean that we search through the family of possible equations (possible cyan curves) relating input to output to find the one that describes the training data most accurately.

It follows that the models in figure 1.2 require labeled input/output pairs for training. For example, the music classification model would require a large number of audio clips where a human expert had identified the genre of each. These input/output pairs take the role of a teacher or supervisor for the training process, and this gives rise to the term *supervised learning*.

1.1.4 Deep neural networks

This book concerns deep neural networks, which are a particularly useful type of machine learning model. They are equations that can represent an extremely broad family of relationships between input and output, and where it is particularly easy to search through this family to find the relationship that describes the training data.

Deep neural networks can process inputs that are very large, of variable length, and contain various kinds of internal structures. They can output single real numbers (regression), multiple numbers (multivariate regression), or probabilities over two or more classes (binary and multiclass classification, respectively). As we shall see in the next section, their outputs may also be very large, of variable length, and contain internal structure. It is probably hard to imagine equations with these properties, and the reader should endeavor to suspend disbelief for now.

1.1.5 Structured outputs

Figure 1.4a depicts a multivariate binary classification model for semantic segmentation. Here, every pixel of an input image is assigned a binary label that indicates whether it belongs to a cow or the background. Figure 1.4b shows a multivariate regression model where the input is an image of a street scene and the output is the depth at each pixel. In both cases, the output is high-dimensional and structured. However, this structure is closely tied to the input, and this can be exploited; if a pixel is labeled as "cow," then a neighbor with a similar RGB value probably has the same label.

Figures 1.4c–e depict three models where the output has a complex structure that is not so closely tied to the input. Figure 1.4c shows a model where the input is an audio file and the output is the transcribed words from that file. Figure 1.4d is a translation

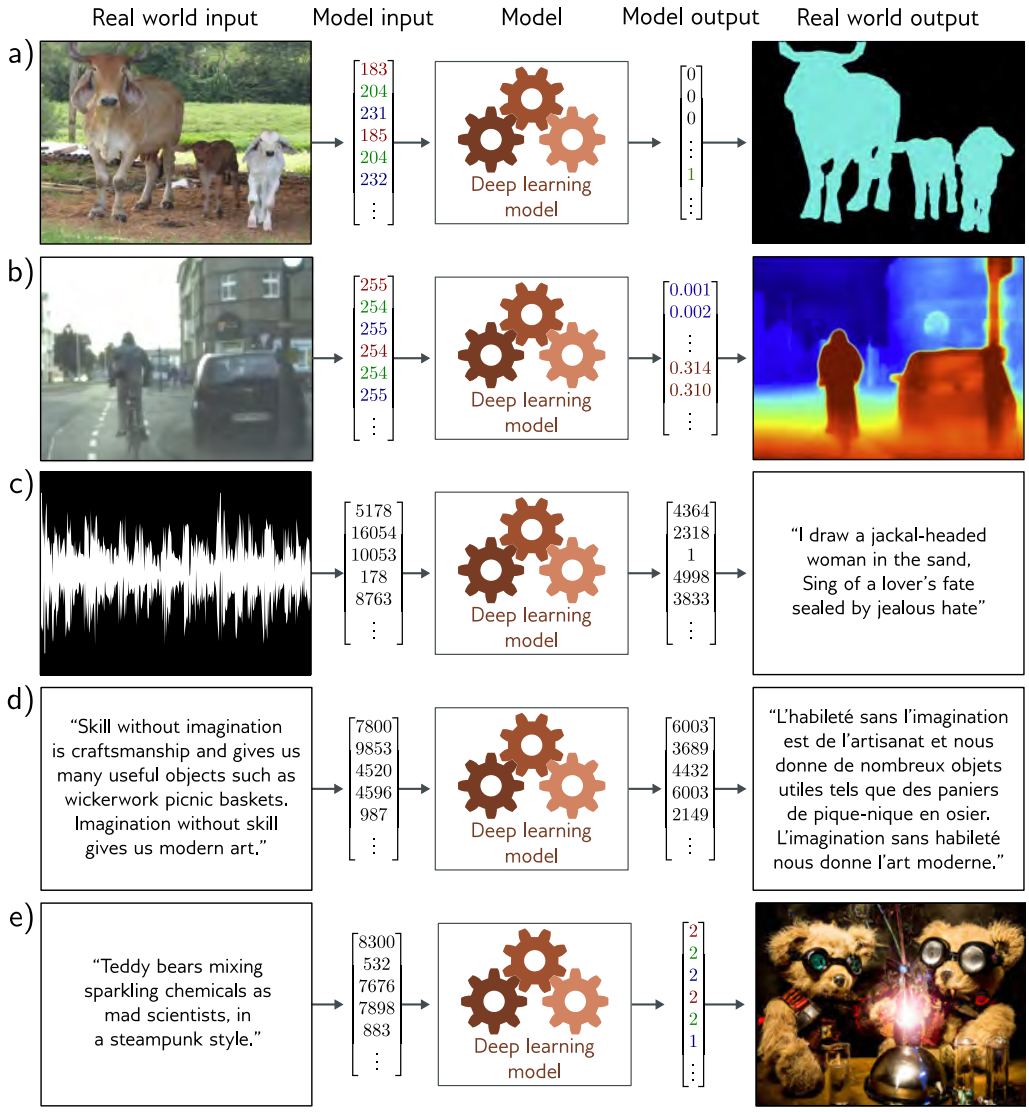


Figure 1.4 Supervised learning tasks with structured outputs. a) This semantic segmentation model maps an RGB image to a binary image indicating whether each pixel belongs to the background or a cow (adapted from Noh et al., 2015). b) This monocular depth estimation model maps an RGB image to an output image where each pixel represents the depth (adapted from Cordts et al., 2016). c) This audio transcription model maps an audio sample to a transcription of the spoken words in the audio. d) This translation model maps an English text string to its French translation. e) This image synthesis model maps a caption to an image (example from https://openai.com/dall-e-2/). In each case, the output has a complex internal structure or grammar. In some cases, many outputs are compatible with the input.

model in which the input is a body of text in English, and the output contains the French translation. Figure 1.4e depicts a very challenging task in which the input is descriptive text, and the model must produce an image that matches this description.

In principle, the latter three tasks can be tackled in the standard supervised learning framework, but they are more difficult for two reasons. First, the output may genuinely be ambiguous; there are multiple valid translations from an English sentence to a French one and multiple images that are compatible with any caption. Second, the output contains considerable structure; not all strings of words make valid English and French sentences, and not all collections of RGB values make plausible images. In addition to learning the mapping, we also have to respect the "grammar" of the output.

Fortunately, this "grammar" can be learned without the need for output labels. For example, we can learn how to form valid English sentences by learning the statistics of a large corpus of text data. This provides a connection with the next section of the book, which considers *unsupervised learning models*.

1.2 Unsupervised learning

Constructing a model from input data without corresponding output labels is termed *unsupervised learning*; the absence of output labels means there can be no "supervision." Rather than learning a mapping from input to output, the goal is to describe or understand the structure of the data. As was the case for supervised learning, the data may have very different characteristics; it may be discrete or continuous, low-dimensional or high-dimensional, and of constant or variable length.

1.2.1 Generative models

This book focuses on *generative unsupervised models*, which learn to synthesize new data examples that are statistically indistinguishable from the training data. Some generative models explicitly describe the probability distribution over the input data and here new examples are generated by sampling from this distribution. Others merely learn a mechanism to generate new examples without explicitly describing their distribution.

State-of-the-art generative models can synthesize examples that are extremely plausible but distinct from the training examples. They have been particularly successful at generating images (figure 1.5) and text (figure 1.6). They can also synthesize data under the constraint that some outputs are predetermined (termed *conditional generation*). Examples include image inpainting (figure 1.7) and text completion (figure 1.8). Indeed, modern generative models for text are so powerful that they can appear intelligent. Given a body of text followed by a question, the model can often "fill in" the missing answer by generating the most likely completion of the document. However, in reality, the model only knows about the statistics of language and does not understand the significance of its answers.

Figure 1.5 Generative models for images. Left: two images were generated from a model trained on pictures of cats. These are not real cats, but samples from a probability model. Right: two images generated from a model trained on images of buildings. Adapted from Karras et al. (2020b).

The moon had risen by the time I reached the edge of the forest, and the light that filtered through the trees was silver and cold. I shivered, though I was not cold, and quickened my pace. I had never been so far from the village before, and I was not sure what to expect. I had been walking for hours, and I was tired and hungry. I had left in such a hurry that I had not thought to pack any food, and I had not thought to bring a weapon. I was unarmed and alone in a strange place, and I did not know what I was doing.

I had been walking for so long that I had lost all sense of time, and I had no idea how far I had come. I only knew that I had to keep going. I had to find her. I was getting close. I could feel it. She was nearby, and she was in trouble. I had to find her and help her, before it was too late.

Figure 1.6 Short story synthesized from a generative model of text data. The model describes a probability distribution that assigns a probability to every output string. Sampling from the model creates strings that follow the statistics of the training data (here, short stories) but have never been seen before.

Figure 1.7 Inpainting. In the original image (left), the boy is obscured by metal cables. These undesirable regions (center) are removed and the generative model synthesizes a new image (right) under the constraint that the remaining pixels must stay the same. Adapted from Saharia et al. (2022a).

I was a little nervous before my first lecture at the University of Bath. It seemed like there were hundreds of students and they looked intimidating. I stepped up to the lectern and was about to speak when something bizarre happened.

Suddenly, the room was filled with a deafening noise, like a giant roar. It was so loud that I couldn't hear anything else and I had to cover my ears. I could see the students looking around, confused and frightened. Then, as quickly as it had started, the noise stopped and the room was silent again.

I stood there for a few moments, trying to make sense of what had just happened. Then I realized that the students were all staring at me, waiting for me to say something. I tried to think of something witty or clever to say, but my mind was blank. So I just said, "Well, that was strange,' and then I started my lecture.

Figure 1.8 Conditional text synthesis. Given an initial body of text (in black), generative models of text can continue the string plausibly by synthesizing the "missing" remaining part of the string. Generated by GPT3 (Brown et al., 2020).

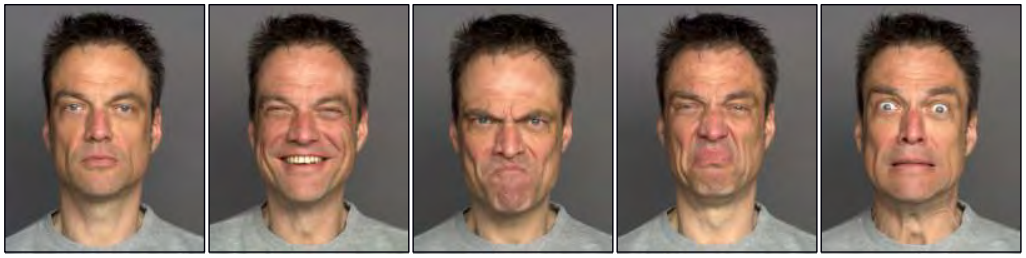


Figure 1.9 Variation of the human face. The human face contains roughly 42 muscles, so it's possible to describe most of the variation in images of the same person in the same lighting with just 42 numbers. In general, datasets of images, music, and text can be described by a relatively small number of underlying variables although it is typically more difficult to tie these to particular physical mechanisms. Images from Dynamic FACES database (Holland et al., 2019).

1.2.2 Latent variables

Some (but not all) generative models exploit the observation that data can be lower dimensional than the raw number of observed variables suggests. For example, the number of valid and meaningful English sentences is considerably smaller than the number of strings created by drawing words at random. Similarly, real-world images are a tiny subset of the images that can be created by drawing random RGB values for every pixel. This is because images are generated by physical processes (see figure 1.9).

This leads to the idea that we can describe each data example using a smaller number of underlying *latent variables*. Here, the role of deep learning is to describe the mapping between these latent variables and the data. The latent variables typically have a simple

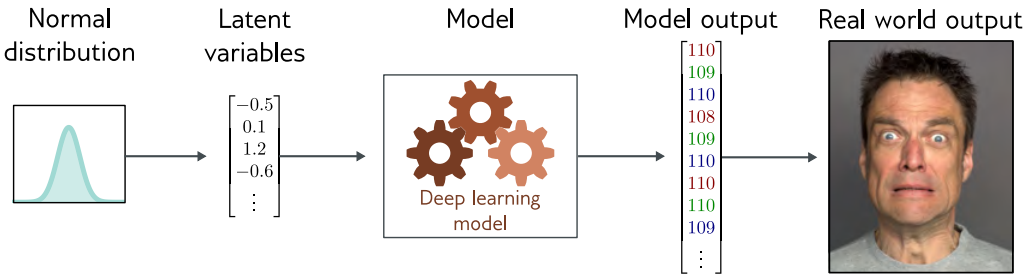


Figure 1.10 Latent variables. Many generative models use a deep learning model to describe the relationship between a low-dimensional "latent" variable and the observed high-dimensional data. The latent variables have a simple probability distribution by design. Hence, new examples can be generated by sampling from the simple distribution over the latent variables and then using the deep learning model to map the sample to the observed data space.

Figure 1.11 Image interpolation. In each row the left and right images are real and the three images in between represent a sequence of interpolations created by a generative model. The generative models that underpin these interpolations have learned that all images can be created by a set of underlying latent variables. By finding these variables for the two real images, interpolating their values, and then using these intermediate variables to create new images, we can generate intermediate results that are both visually plausible and mix the characteristics of the two original images. Top row adapted from Sauer et al. (2022). Bottom row adapted from Ramesh et al. (2022).

Figure 1.12 Multiple images generated from the caption "A teddy bear on a skateboard in Times Square." Generated by DALL·E-2 (Ramesh et al., 2022).

probability distribution by design. By sampling from this distribution and passing the result through the deep learning model, we can create new samples (figure 1.10).

These models lead to new methods for manipulating real data. For example, consider finding the latent variables that underpin two real examples. We can interpolate between these examples by interpolating between their latent representations and mapping the intermediate positions back into the data space (figure 1.11).

1.2.3 Connecting supervised and unsupervised learning

Generative models with latent variables can also benefit supervised learning models where the outputs have structure (figure 1.4). For example, consider learning to predict the images corresponding to a caption. Rather than directly map the text input to an image, we can learn a relation between latent variables that explain the text and the latent variables that explain the image.

This has three advantages. First, we may need fewer text/image pairs to learn this mapping now that the inputs and outputs are lower dimensional. Second, we are more likely to generate a plausible-looking image; any sensible values of the latent variables should produce something that looks like a plausible example. Third, if we introduce randomness to either the mapping between the two sets of latent variables or the mapping from the latent variables to the image, then we can generate multiple images that are all described well by the caption (figure 1.12).

1.3 Reinforcement learning

The final area of machine learning is reinforcement learning. This paradigm introduces the idea of an agent which lives in a world and can perform certain actions at each time step. The actions change the state of the system but not necessarily in a deterministic way. Taking an action can also produce rewards, and the goal of reinforcement learning

is for the agent to learn to choose actions that lead to high rewards on average.

One complication is that the reward may occur some time after the action is taken, so associating a reward with an action is not straightforward. This is known as the *temporal credit assignment problem*. As the agent learns, it must trade off *exploration* and *exploitation* of what it already knows; perhaps the agent has already learned how to receive modest rewards; should it follow this strategy (exploit what it knows), or should it try different actions to see if it can improve (explore other opportunities)?

1.3.1 Two examples

Consider teaching a humanoid robot to locomote. The robot can perform a limited number of actions at a given time (moving various joints), and these change the state of the world (its pose). We might reward the robot for reaching checkpoints in an obstacle course. To reach each checkpoint, it must perform many actions, and it's unclear which ones contributed to the reward when it is received and which were irrelevant. This is an example of the temporal credit assignment problem.

A second example is learning to play chess. Again, the agent has a set of valid actions (chess moves) at any given time. However, these actions change the state of the system in a non-deterministic way; for any choice of action, the opposing player might respond with many different moves. Here, we might set up a reward structure based on capturing pieces or just have a single reward at the end of the game for winning. In the latter case, the temporal credit assignment problem is extreme; the system must learn which of the many moves it made were instrumental to success or failure.

The exploration-exploitation trade-off is also apparent in these two examples. The robot may have discovered that it can make progress by lying on its side and pushing with one leg. This strategy will move the robot and yields rewards, but much more slowly than the optimal solution: to balance on its legs and walk. So, it faces a choice between exploiting what it already knows (how to slide along the floor awkwardly) and exploring the space of actions (which might result in much faster locomotion). Similarly, in the chess example, the agent may learn a reasonable sequence of opening moves. Should it exploit this knowledge or explore different opening sequences?

It is perhaps not obvious how deep learning fits into the reinforcement learning framework. There are several possible approaches, but one technique is to use deep networks to build a mapping from the observed world state to an action. This is known as a *policy network*. In the robot example, the policy network would learn a mapping from its sensor measurements to joint movements. In the chess example, the network would learn a mapping from the current state of the board to the choice of move (figure 1.13).

1.4 Ethics

It would be irresponsible to write this book without discussing the ethical implications of artificial intelligence. This potent technology will change the world to at least the

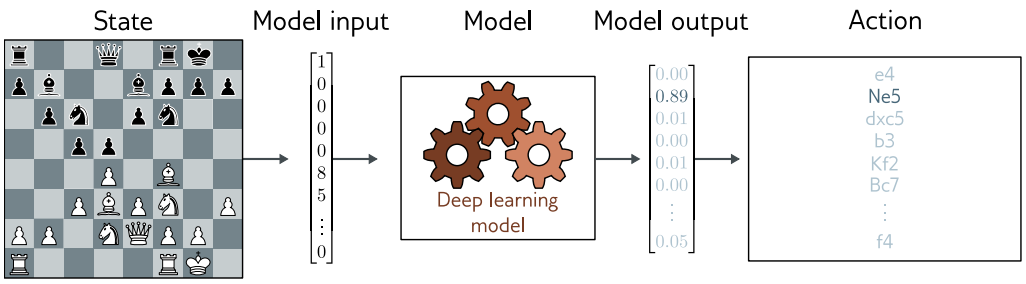


Figure 1.13 Policy networks for reinforcement learning. One way to incorporate deep neural networks into reinforcement learning is to use them to define a mapping from the state (here position on chessboard) to the actions (possible moves). This mapping is known as a *policy*.

same extent as electricity, the internal combustion engine, the transistor, or the internet. The potential benefits in healthcare, design, entertainment, transport, education, and almost every area of commerce are enormous. However, scientists and engineers are often unrealistically optimistic about the outcomes of their work, and the potential for harm is just as great. The following paragraphs highlight five concerns.

Bias and fairness: If we train a system to predict salary levels for individuals based on historical data, then this system will reproduce historical biases; for example, it will probably predict that women should be paid less than men. Several such cases have already become international news stories: an AI system for super-resolving face images made non-white people look more white; a system for generating images produced only pictures of men when asked to synthesize pictures of lawyers. Careless application of algorithmic decision-making using AI has the potential to entrench or aggravate existing biases. See Binns (2018) for further discussion.

Explainability: Deep learning systems make decisions, but we do not usually know exactly how or based on what information. They may contain billions of parameters, and there is no way we can understand how they work based on examination. This has led to the sub-field of explainable AI. One moderately successful area is producing local explanations; we cannot explain the entire system, but we can produce an interpretable description of why a particular decision was made. However, it remains unknown whether it is possible to build complex decision-making systems that are fully transparent to their users or even their creators. See Grennan et al. (2022) for further information.

Weaponizing AI: All significant technologies have been applied directly or indirectly toward war. Sadly, violent conflict seems to be an inevitable feature of human behavior. AI is arguably the most powerful technology ever built and will doubtless be deployed extensively in a military context. Indeed, this is already happening (Heikkilä, 2022).

Concentrating power: It is not from a benevolent interest in improving the lot of the human race that the world's most powerful companies are investing heavily in artificial intelligence. They know that these technologies will allow them to reap enormous profits. Like any advanced technology, deep learning is likely to concentrate power in the hands of the few organizations that control it. Automating jobs that are currently done by humans will change the economic environment and disproportionately affect the livelihoods of lower-paid workers with fewer skills. Optimists argue similar disruptions happened during the industrial revolution and resulted in shorter working hours. The truth is that we simply do not know what effects the large-scale adoption of AI will have on society (see David, 2015).

Existential risk: The major existential risks to the human race all result from technology. Climate change has been driven by industrialization. Nuclear weapons derive from the study of physics. Pandemics are more probable and spread faster because innovations in transport, agriculture, and construction have allowed a larger, denser, and more interconnected population. Artificial intelligence brings new existential risks. We should be very cautious about building systems that are more capable and extensible than human beings. In the most optimistic case, it will put vast power in the hands of the owners. In the most pessimistic case, we will be unable to control it or even understand its motives (see Tegmark, 2018).

This list is far from exhaustive. AI could also enable surveillance, disinformation, violations of privacy, fraud, and manipulation of financial markets, and the energy required to train AI systems contributes to climate change. Moreover, these concerns are not speculative; there are already many examples of ethically dubious applications of AI (consult Dao, 2021, for a partial list). In addition, the recent history of the internet has shown how new technology can cause harm in unexpected ways. The online community of the eighties and early nineties could hardly have predicted the proliferation of fake news, spam, online harassment, fraud, cyberbullying, incel culture, political manipulation, doxxing, online radicalization, and revenge porn.

Everyone studying or researching (or writing books about) AI should contemplate to what degree scientists are accountable for the uses of their technology. We should consider that capitalism primarily drives the development of AI and that legal advances and deployment for social good are likely to lag significantly behind. We should reflect on whether it's possible, as scientists and engineers, to control progress in this field and to reduce the potential for harm. We should consider what kind of organizations we are prepared to work for. How serious are they in their commitment to reducing the potential harms of AI? Are they simply "ethics-washing" to reduce reputational risk, or do they actually implement mechanisms to halt ethically suspect projects?

All readers are encouraged to investigate these issues further. The online course at https://ethics-of-ai.mooc.fi/ is a useful introductory resource. If you are a professor teaching from this book, you are encouraged to raise these issues with your students. If you are a student taking a course where this is not done, then lobby your professor to make this happen. If you are deploying or researching AI in a corporate environment, you are encouraged to scrutinize your employer's values and to help change them (or leave) if they are wanting.

1.5 Structure of book

The structure of the book follows the structure of this introduction. Chapters 2–9 walk through the supervised learning pipeline. We describe shallow and deep neural networks and discuss how to train them and measure and improve their performance. Chapters 10–13 describe common architectural variations of deep neural networks, including convolutional networks, residual connections, and transformers. These architectures are used across supervised, unsupervised, and reinforcement learning.

Chapters 14–18 tackle unsupervised learning using deep neural networks. We devote a chapter each to four modern deep generative models: generative adversarial networks, variational autoencoders, normalizing flows, and diffusion models. Chapter 19 is a brief introduction to deep reinforcement learning. This is a topic that easily justifies its own book, so the treatment is necessarily superficial. However, this treatment is intended to be a good starting point for readers unfamiliar with this area.

Despite the title of this book, some aspects of deep learning remain poorly understood. Chapter 20 poses some fundamental questions. Why are deep networks so easy to train? Why do they generalize so well? Why do they need so many parameters? Do they need to be deep? Along the way, we explore unexpected phenomena such as the structure of the loss function, double descent, grokking, and lottery tickets. The book concludes with chapter 21, which discusses ethics and deep learning.

1.6 Other books

This book is self-contained but is limited to coverage of deep learning. It is intended to be the spiritual successor to *Deep Learning* (Goodfellow et al., 2016) which is a fantastic resource but does not cover recent advances. For a broader look at machine learning, the most up-to-date and encyclopedic resource is *Probabilistic Machine Learning* (Murphy, 2022, 2023). However, *Pattern Recognition and Machine Learning* (Bishop, 2006) is still an excellent and relevant book.

If you enjoy this book, then my previous volume, *Computer Vision: Models, Learning, and Inference* (Prince, 2012), is still worth reading. Some parts have dated badly, but it contains a thorough introduction to probability, including Bayesian methods, and good introductory coverage of latent variable models, geometry for computer vision, Gaussian processes, and graphical models. It uses identical notation to this book and can be found online. A detailed treatment of graphical models can be found in *Probabilistic Graphical Models: Principles and Techniques* (Koller & Friedman, 2009), and Gaussian processes are covered by *Gaussian Processes for Machine Learning* (Williams & Rasmussen, 2006).

For background mathematics, consult *Mathematics for Machine Learning* (Deisenroth et al., 2020). For a more coding-oriented approach, consult *Dive into Deep Learning* (Zhang et al., 2023). The best overview for computer vision is Szeliski (2022), and there is also the impending book *Foundations of Computer Vision* (Torralba et al., 2024). A good starting point to learn about graph neural networks is *Graph Representation Learning* (Hamilton, 2020). The definitive work on reinforcement learning is *Reinforce-*

ment Learning: An Introduction (Sutton & Barto, 2018). A good initial resource is *Foundations of Deep Reinforcement Learning* (Graesser & Keng, 2019).

1.7 How to read this book

Most remaining chapters in this book contain a main body of text, a notes section, and a set of problems. The main body of the text is intended to be self-contained and can be read without recourse to the other parts of the chapter. As much as possible, background mathematics is incorporated into the main body of the text. However, for larger topics that would be a distraction to the main thread of the argument, the background material is appendicized, and a reference is provided in the margin. Most notation in this book is standard. However, some conventions are less widely used, and the reader is encouraged to consult appendix A before proceeding.

Appendix A Notation

The main body of text includes many novel illustrations and visualizations of deep learning models and results. I've worked hard to provide new explanations of existing ideas rather than merely curate the work of others. Deep learning is a new field, and sometimes phenomena are poorly understood. I try to make it clear where this is the case and when my explanations should be treated with caution.

References are included in the main body of the chapter only where results are depicted. Instead, they can be found in the notes section at the end of the chapter. I do not generally respect historical precedent in the main text; if an ancestor of a current technique is no longer useful, then I will not mention it. However, the historical development of the field is described in the notes section, and hopefully, credit is fairly assigned. The notes are organized into paragraphs and provide pointers for further reading. They should help the reader orient themselves within the sub-area and understand how it relates to other parts of machine learning. The notes are less self-contained than the main text. Depending on your level of background knowledge and interest, you may find these sections more or less useful.

Each chapter has a number of associated problems. They are referenced in the margin of the main text at the point that they should be attempted. As George Pólya noted, "Mathematics, you see, is not a spectator sport." He was correct, and I highly recommend that you attempt the problems as you go. In some cases, they provide insights that will help you understand the main text. Problems for which the answers are provided on the associated website are indicated with an asterisk. Additionally, Python notebooks that will help you understand the ideas in this book are also available via the website, and these are also referenced in the margins of the text. Indeed, if you are feeling rusty, it might be worth working through the notebook on background mathematics right now.

Notebook 1.1 Background mathematics

Unfortunately, the pace of research in AI makes it inevitable that this book will be a constant work in progress. If there are parts you find hard to understand, notable omissions, or sections that seem extraneous, please get in touch via the associated website. Together, we can make the next edition better.

Chapter 2

Supervised learning

A *supervised learning model* defines a mapping from one or more inputs to one or more outputs. For example, the input might be the age and mileage of a secondhand Toyota Prius, and the output might be the estimated value of the car in dollars.

The model is just a mathematical equation; when the inputs are passed through this equation, it computes the output, and this is termed *inference*. The model equation also contains *parameters*. Different parameter values change the outcome of the computation; the model equation describes a family of possible relationships between inputs and outputs, and the parameters specify the particular relationship.

When we *train* or *learn* a model, we find parameters that describe the true relationship between inputs and outputs. A learning algorithm takes a training set of input/output pairs and manipulates the parameters until the inputs predict their corresponding outputs as closely as possible. If the model works well for these training pairs, then we hope it will make good predictions for new inputs where the true output is unknown.

The goal of this chapter is to expand on these ideas. First, we describe this framework more formally and introduce some notation. Then we work through a simple example in which we use a straight line to describe the relationship between input and output. This linear model is both familiar and easy to visualize, but nevertheless illustrates all the main ideas of supervised learning.

2.1 Supervised learning overview

In supervised learning, we aim to build a model that takes an input $\mathbf{x}$ and outputs a prediction $\mathbf{y}$. For simplicity, we assume that both the input $\mathbf{x}$ and output $\mathbf{y}$ are vectors of a predetermined and fixed size and that the elements of each vector are always ordered in the same way; in the Prius example above, the input $\mathbf{x}$ would always contain the age of the car and then the mileage, in that order. This is termed *structured* or *tabular* data.

To make the prediction, we need a model $\mathbf{f}[\bullet]$ that takes input $\mathbf{x}$ and returns $\mathbf{y}$, so:

$$\mathbf{y} = \mathbf{f}[\mathbf{x}]. \tag{2.1}$$

When we compute the prediction $\mathbf{y}$ from the input $\mathbf{x}$, we call this *inference*.

The model is just a mathematical equation with a fixed form. It represents a family of different relations between the input and the output. The model also contains *parameters* $\boldsymbol{\phi}$. The choice of parameters determines the particular relation between input and output, so we should really write:

$$\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]. \tag{2.2}$$

When we talk about *learning* or *training* a model, we mean that we attempt to find parameters $\boldsymbol{\phi}$ that make sensible output predictions from the input. We learn these parameters using a *training dataset* of I pairs of input and output examples $\{\mathbf{x}_i, \mathbf{y}_i\}$. We aim to select parameters that map each training input to its associated output as closely as possible. We quantify the degree of mismatch in this mapping with the *loss* L. This is a scalar value that summarizes how poorly the model predicts the training outputs from their corresponding inputs for parameters $\boldsymbol{\phi}$.

We can treat the loss as a function $L[\boldsymbol{\phi}]$ of these parameters. When we train the model, we are seeking parameters $\hat{\boldsymbol{\phi}}$ that minimize this *loss function*:[1]

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L\left[\boldsymbol{\phi}\right]\Big]. \tag{2.3}$$

If the loss is small after this minimization, we have found model parameters that accurately predict the training outputs $\mathbf{y}_i$ from the training inputs $\mathbf{x}_i$.

After training a model, we must now assess its performance; we run the model on separate *test data* to see how well it *generalizes* to examples that it didn't observe during training. If the performance is adequate, then we are ready to deploy the model.

2.2 Linear regression example

Let's now make these ideas concrete with a simple example. We consider a model $y = \mathrm{f}[x, \boldsymbol{\phi}]$ that predicts a single output y from a single input x. Then we develop a loss function, and finally, we discuss model training.

2.2.1 1D linear regression model

A *1D linear regression model* describes the relationship between input x and output y as a straight line:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{2.4}$$

[1] More properly, the loss function also depends on the training data $\{\mathbf{x}_i, \mathbf{y}_i\}$, so we should write $L\left[\{\mathbf{x}_i, \mathbf{y}_i\}, \boldsymbol{\phi}\right]$, but this is rather cumbersome.

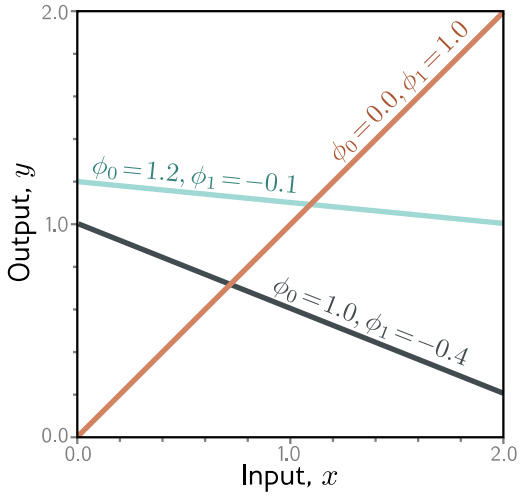


Figure 2.1 Linear regression model. For a given choice of parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model makes a prediction for the output (y-axis) based on the input (x-axis). Different choices for the y-intercept ϕ_0 and the slope ϕ_1 change these predictions (cyan, orange, and gray lines). The linear regression model (equation 2.4) defines a family of input/output relations (lines) and the parameters determine the member of the family (the particular line).

This model has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where ϕ_0 is the y-intercept of the line and ϕ_1 is the slope. Different choices for the y-intercept and slope result in different relations between input and output (figure 2.1). Hence, equation 2.4 defines a family of possible input-output relations (all possible lines), and the choice of parameters determines the member of this family (the particular line).

2.2.2 Loss

For this model, the training dataset (figure 2.2a) consists of I input/output pairs $\{x_i, y_i\}$. Figures 2.2b–d show three lines defined by three sets of parameters. The green line in figure 2.2d describes the data more accurately than the other two since it is much closer to the data points. However, we need a principled approach for deciding which parameters $\boldsymbol{\phi}$ are better than others. To this end, we assign a numerical value to each choice of parameters that quantifies the degree of mismatch between the model and the data. We term this value the *loss*; a lower loss means a better fit.

The mismatch is captured by the deviation between the model predictions $\mathrm{f}[x_i, \boldsymbol{\phi}]$ (height of the line at x_i) and the ground truth outputs y_i. These deviations are depicted as orange dashed lines in figures 2.2b–d. We quantify the total mismatch, *training error*, or *loss* as the sum of the squares of these deviations for all I training pairs:

$$\begin{aligned} L[\phi] &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2 . \end{aligned} \tag{2.5}$$

Since the best parameters minimize this expression, we call this a *least-squares* loss. The squaring operation means that the direction of the deviation (i.e., whether the line is

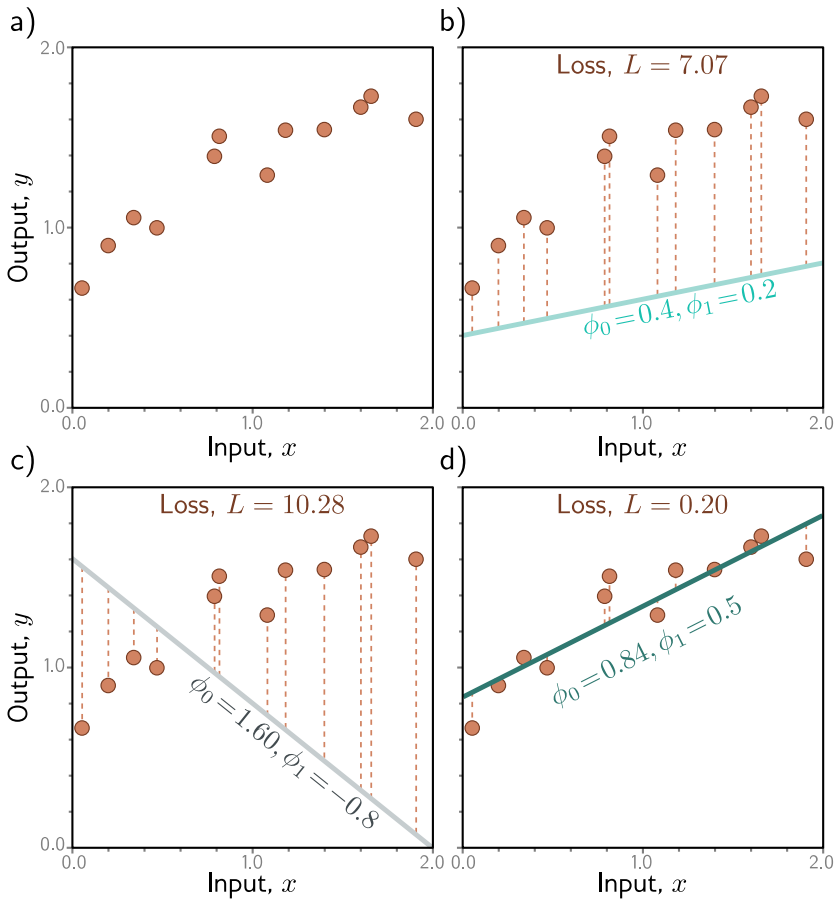


Figure 2.2 Linear regression training data, model, and loss. a) The training data (orange points) consist of $I = 12$ input/output pairs $\{x_i, y_i\}$. b–d) Each panel shows the linear regression model with different parameters. Depending on the choice of y-intercept and slope parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model errors (orange dashed lines) may be larger or smaller. The loss L is the sum of the squares of these errors. The parameters that define the lines in panels (b) and (c) have large losses $L = 7.07$ and $L = 10.28$, respectively because the models fit badly. The loss $L = 0.20$ in panel (d) is smaller because the model fits well; in fact, this has the smallest loss of all possible lines, so these are the optimal parameters.

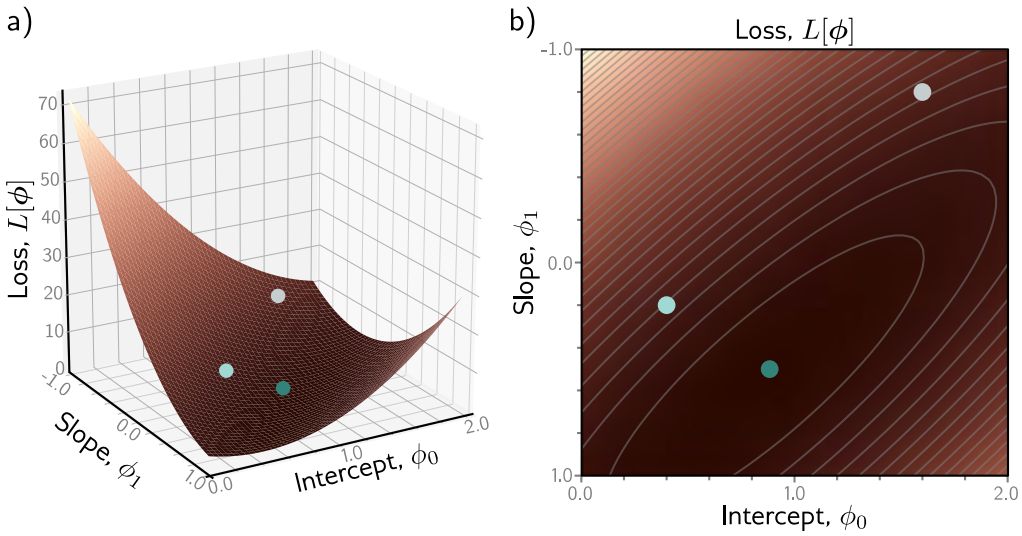


Figure 2.3 Loss function for linear regression model with the dataset in figure 2.2a. a) Each combination of parameters $\boldsymbol{\phi} = [\phi_0,\phi_1]^T$ has an associated loss. The resulting loss function $L[\boldsymbol{\phi}]$ can be visualized as a surface. The three circles represent the lines from figure 2.2b–d. b) The loss can also be visualized as a heatmap, where brighter regions represent larger losses; here we are looking straight down at the surface in (a) from above and gray ellipses represent isocontours. The best fitting line (figure 2.2d) has the parameters with the smallest loss (green circle).

above or below the data) is unimportant. There are also theoretical reasons for this choice which we return to in chapter 5.

The loss L is a function of the parameters $\boldsymbol{\phi}$; it will be larger when the model fit is poor (figure 2.2b,c) and smaller when it is good (figure 2.2d). Considered in this light, we term $L[\boldsymbol{\phi}]$ the *loss function* or *cost function*. The goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this quantity:

Notebook 2.1 Supervised learning

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L[\boldsymbol{\phi}]\Big] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\left(\mathrm{f}[x_i,\boldsymbol{\phi}]-y_i\right)^2\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\left(\phi_0+\phi_1 x_i-y_i\right)^2\right]. \qquad (2.6)\end{aligned}$$

There are only two parameters (the y-intercept ϕ_0 and slope ϕ_1), so we can calculate the loss for every combination of values and visualize the loss function as a surface (figure 2.3). The "best" parameters are at the minimum of this surface.

Problems 2.1–2.2

2.2.3 Training

The process of finding parameters that minimize the loss is termed *model fitting*, *training*, or *learning*. The basic method is to choose the initial parameters randomly and then improve them by "walking down" the loss function until we reach the bottom (figure 2.4). One way to do this is to measure the gradient of the surface at the current position and take a step in the direction that is most steeply downhill. Then we repeat this process until the gradient is flat and we can improve no further.[2]

2.2.4 Testing

Having trained the model, we want to know how it will perform in the real world. We do this by computing the loss on a separate set of *test data*. The degree to which the prediction accuracy *generalizes* to the test data depends in part on how representative and complete the training data is. However, it also depends on how expressive the model is. A simple model like a line might not be able to capture the true relationship between input and output. This is known as *underfitting*. Conversely, a very expressive model may describe statistical peculiarities of the training data that are atypical and lead to unusual predictions. This is known as *overfitting*.

2.3 Summary

A supervised learning model is a function $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that relates inputs $\mathbf{x}$ to outputs $\mathbf{y}$. The particular relationship is determined by parameters $\boldsymbol{\phi}$. To train the model, we define a loss function $L[\boldsymbol{\phi}]$ over a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$. This quantifies the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and observed outputs $\mathbf{y}_i$ as a function of the parameters $\boldsymbol{\phi}$. Then we search for the parameters that minimize the loss. We evaluate the model on a different set of test data to see how well it generalizes to new inputs.

Chapters 3–9 expand on these ideas. First, we tackle the model itself; 1D linear regression has the obvious drawback that it can only describe the relationship between the input and output as a straight line. Shallow neural networks (chapter 3) are only slightly more complex than linear regression but describe a much larger family of input/output relationships. Deep neural networks (chapter 4) are just as expressive but can describe complex functions with fewer parameters and work better in practice.

Chapter 5 investigates loss functions for different tasks and reveals the theoretical underpinnings of the least-squares loss. Chapters 6 and 7 discuss the training process. Chapter 8 discusses how to measure model performance. Chapter 9 considers *regularization* techniques, which aim to improve that performance.

[2] This iterative approach is not actually necessary for the linear regression model. Here, it's possible to find closed-form expressions for the parameters. However, this *gradient descent* approach works for more complex models where there is no closed-form solution and where there are too many parameters to evaluate the loss for every combination of values.

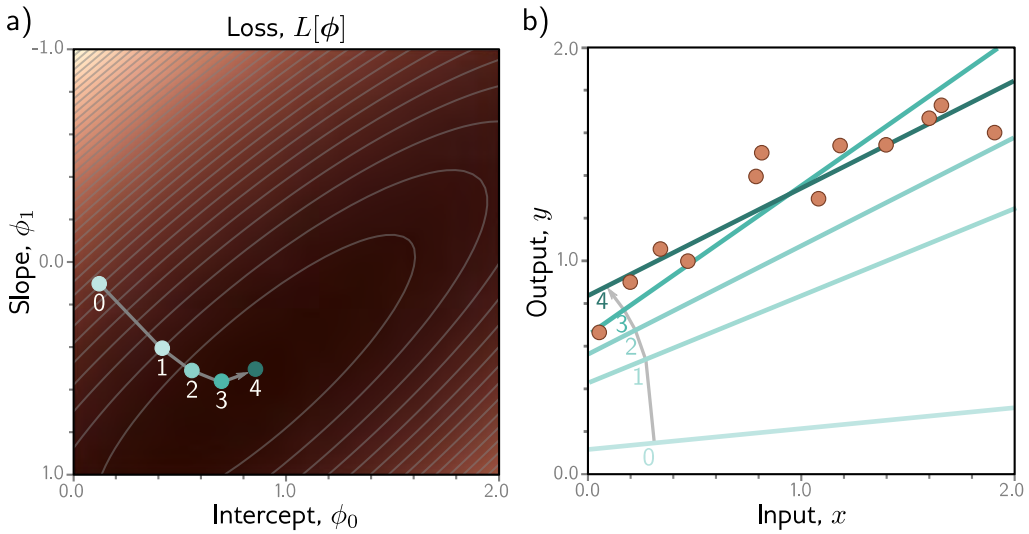


Figure 2.4 Linear regression training. The goal is to find the y-intercept and slope parameters that correspond to the smallest loss. a) Iterative training algorithms initialize the parameters randomly and then improve them by "walking downhill" until no further improvement can be made. Here, we start at position 0 and move a certain distance downhill (perpendicular to the contours) to position 1. Then we re-calculate the downhill direction and move to position 2. Eventually, we reach the minimum of the function (position 4). b) Each position 0–4 from panel (a) corresponds to a different y-intercept and slope and so represents a different line. As the loss decreases, the lines fit the data more closely.

Notes

Loss functions vs. cost functions: In much of machine learning and in this book, the terms loss function and cost function are used interchangeably. However, more properly, a loss function is the individual term associated with a data point (i.e., each of the squared terms on the right-hand side of equation 2.5), and the cost function is the overall quantity that is minimized (i.e., the entire right-hand side of equation 2.5). A cost function can contain additional terms that are not associated with individual data points (see section 9.1). More generally, an *objective function* is any function that is to be maximized or minimized.

Generative vs. discriminative models: The models $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ in this chapter are *discriminative models*. These make an output prediction $\mathbf{y}$ from real-world measurements $\mathbf{x}$. Another approach is to build a *generative model* $\mathbf{x} = \mathbf{g}[\mathbf{y}, \boldsymbol{\phi}]$, in which the real-world measurements $\mathbf{x}$ are computed as a function of the output $\mathbf{y}$. Problem 2.3

The generative approach has the disadvantage that it doesn't directly predict $\mathbf{y}$. To perform inference, we must invert the generative equation as $\mathbf{y} = \mathbf{g}^{-1}[\mathbf{x}, \boldsymbol{\phi}]$, and this may be difficult. However, generative models have the advantage that we can build in prior knowledge about how the data were created. For example, if we wanted to predict the 3D position and orientation $\mathbf{y}$

of a car in an image $\mathbf{x}$, then we could build knowledge about car shape, 3D geometry, and light transport into the function $\mathbf{x} = \mathbf{g}[\mathbf{y}, \boldsymbol{\phi}]$.

This seems like a good idea, but in fact, discriminative models dominate modern machine learning; the advantage gained from exploiting prior knowledge in generative models is usually trumped by learning very flexible discriminative models with large amounts of training data.

Problems

Problem 2.1 To walk "downhill" on the loss function (equation 2.5), we measure its gradient with respect to the parameters ϕ_0 and ϕ_1. Calculate expressions for the slopes $\partial L/\partial\phi_0$ and $\partial L/\partial\phi_1$.

Problem 2.2 Show that we can find the minimum of the loss function in closed form by setting the expression for the derivatives from problem 2.1 to zero and solving for ϕ_0 and ϕ_1. Note that this works for linear regression but not for more complex models; this is why we use iterative model fitting methods like gradient descent (figure 2.4).

Problem 2.3* Consider reformulating linear regression as a generative model, so we have $x = \mathrm{g}[y, \boldsymbol{\phi}] = \phi_0 + \phi_1 y$. What is the new loss function? Find an expression for the inverse function $y = \mathrm{g}^{-1}[x, \boldsymbol{\phi}]$ that we would use to perform inference. Will this model make the same predictions as the discriminative version for a given training dataset $\{x_i, y_i\}$? One way to establish this is to write code that fits a line to three data points using both methods and see if the result is the same.

Chapter 3

Shallow neural networks

Chapter 2 introduced supervised learning using 1D linear regression. However, this model can only describe the input/output relationship as a line. This chapter introduces shallow neural networks. These describe piecewise linear functions and are expressive enough to approximate arbitrarily complex relationships between multi-dimensional inputs and outputs.

3.1 Neural network example

Shallow neural networks are functions $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that map multivariate inputs $\mathbf{x}$ to multivariate outputs $\mathbf{y}$. We defer a full definition until section 3.4 and introduce the main ideas using an example network $\mathrm{f}[x, \boldsymbol{\phi}]$ that maps a scalar input x to a scalar output y and has ten parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x]. \end{aligned} \tag{3.1}$$

We can break down this calculation into three parts: first, we compute three linear functions of the input data ($\theta_{10} + \theta_{11}x$, $\theta_{20} + \theta_{21}x$, and $\theta_{30} + \theta_{31}x$). Second, we pass the three results through an *activation function* $\mathrm{a}[\bullet]$. Finally, we weight the three resulting activations with ϕ_1, ϕ_2, and ϕ_3, sum them, and add an offset ϕ_0.

To complete the description, we must define the activation function $\mathrm{a}[\bullet]$. There are many possibilities, but the most common choice is the *rectified linear unit* or *ReLU*:

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}. \tag{3.2}$$

This returns the input when it is positive and zero otherwise (figure 3.1).

It is probably not obvious which family of input/output relations is represented by equation 3.1. Nonetheless, the ideas from the previous chapter are all applicable. Equation 3.1 represents a family of functions where the particular member of the family

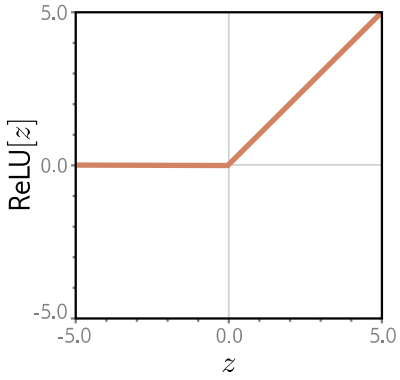


Figure 3.1 Rectified linear unit (ReLU). This activation function returns zero if the input is less than zero and returns the input unchanged otherwise. In other words, it clips negative values to zero. Note that there are many other possible choices for the activation function (see figure 3.13), but the ReLU is the most commonly used and the easiest to understand.

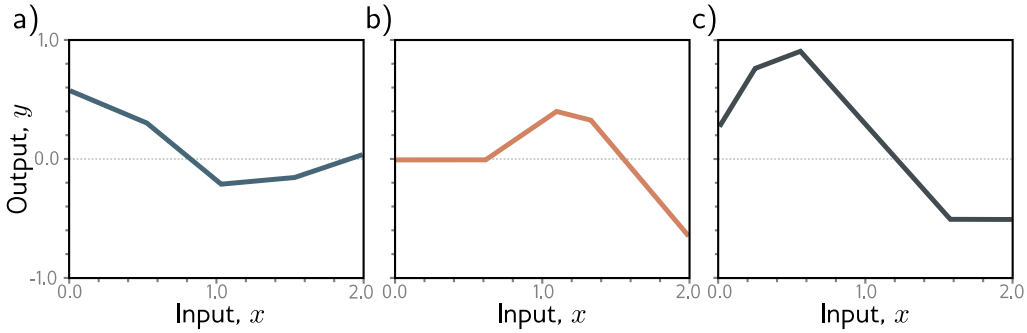


Figure 3.2 Family of functions defined by equation 3.1. a–c) Functions for three different choices of the ten parameters $\boldsymbol{\phi}$. In each case, the input/output relation is piecewise linear. However, the positions of the joints, the slopes of the linear regions between them, and the overall height vary.

depends on the ten parameters in $\boldsymbol{\phi}$. If we know these parameters, we can perform inference (predict y) by evaluating the equation for a given input x. Given a training dataset $\{x_i, y_i\}_{i=1}^{I}$, we can define a least squares loss function $L[\boldsymbol{\phi}]$ and use this to measure how effectively the model describes this dataset for any given parameter values $\boldsymbol{\phi}$. To train the model, we search for the values $\hat{\boldsymbol{\phi}}$ that minimize this loss.

3.1.1 Neural network intuition

In fact, equation 3.1 represents a family of continuous piecewise linear functions (figure 3.2) with up to four linear regions. We now break down equation 3.1 and show *why* it describes this family. To make this easier to understand, we split the function into two parts. First, we introduce the intermediate quantities:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{3.3}$$

where we refer to h_1, h_2, and h_3 as *hidden units*. Second, we compute the output by combining these hidden units with a linear function:[1]

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{3.4}$$

Figure 3.3 shows the flow of computation that creates the function in figure 3.2a. Each hidden unit contains a linear function $\theta_{\bullet 0} + \theta_{\bullet 1}x$ of the input, and that line is clipped by the ReLU function $a[\bullet]$ below zero. The positions where the three lines cross zero become the three "joints" in the final output. The three clipped lines are then weighted by ϕ_1, ϕ_2, and ϕ_3, respectively. Finally, the offset ϕ_0 is added, which controls the overall height of the final function.

Problems 3.1–3.8

Each linear region in figure 3.3j corresponds to a different *activation pattern* in the hidden units. When a unit is clipped, we refer to it as *inactive*, and when it is not clipped, we refer to it as *active*. For example, the shaded region receives contributions from h_1 and h_3 (which are active) but not from h_2 (which is inactive). The slope of each linear region is determined by (i) the original slopes $\theta_{\bullet 1}$ of the active inputs for this region and (ii) the weights $\phi_\bullet$ that were subsequently applied. For example, the slope in the shaded region (see problem 3.3) is $\theta_{11}\phi_1 + \theta_{31}\phi_3$, where the first term is the slope in panel (g) and the second term is the slope in panel (i).

Each hidden unit contributes one "joint" to the function, so with three hidden units, there can be four linear regions. However, only three of the slopes of these regions are independent; the fourth is either zero (if all the hidden units are inactive in this region) or is a sum of slopes from the other regions.

Notebook 3.1
Shallow networks I

Problem 3.9

3.1.2 Depicting neural networks

We have been discussing a neural network with one input, one output, and three hidden units. We visualize this network in figure 3.4a. The input is on the left, the hidden units are in the middle, and the output is on the right. Each connection represents one of the ten parameters. To simplify this representation, we do not typically draw the intercept parameters, so this network is usually depicted as in figure 3.4b.

[1] For the purposes of this book, a linear function has the form $z' = \phi_0 + \sum_i \phi_i z_i$. Any other type of function is nonlinear. For instance, the ReLU function (equation 3.2) and the example neural network that contains it (equation 3.1) are both nonlinear. See notes at end of chapter for further clarification.

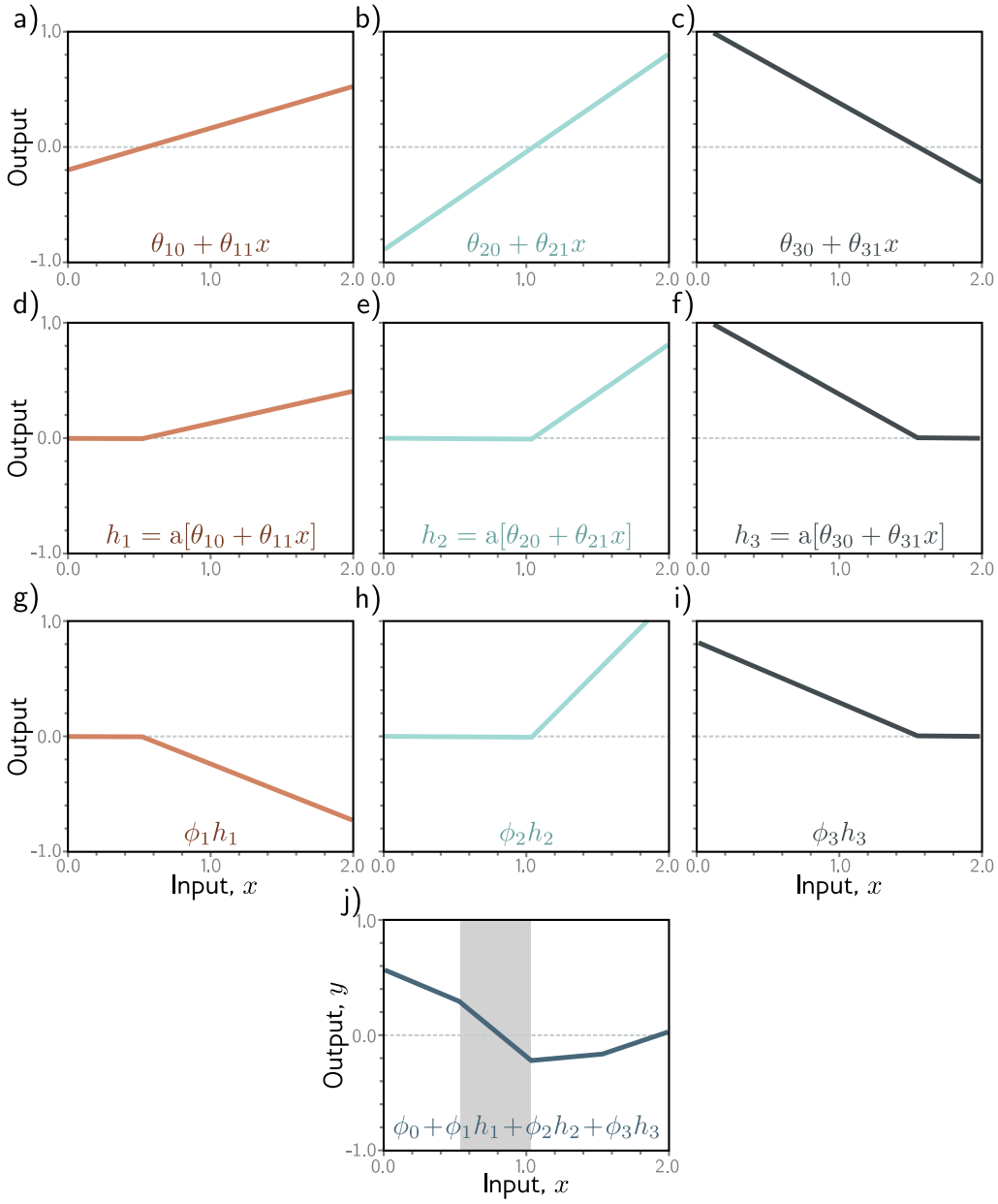


Figure 3.3 Computation for function in figure 3.2a. a–c) The input x is passed through three linear functions, each with a different y-intercept $\theta_{\bullet 0}$ and slope $\theta_{\bullet 1}$. d–f) Each line is passed through the ReLU activation function, which clips negative values to zero. g–i) The three clipped lines are then weighted (scaled) by ϕ_1, ϕ_2, and ϕ_3, respectively. j) Finally, the clipped and weighted functions are summed, and an offset ϕ_0 that controls the height is added. Each of the four linear regions corresponds to a different activation pattern in the hidden units. In the shaded region, h_2 is inactive (clipped), but h_1 and h_3 are both active.

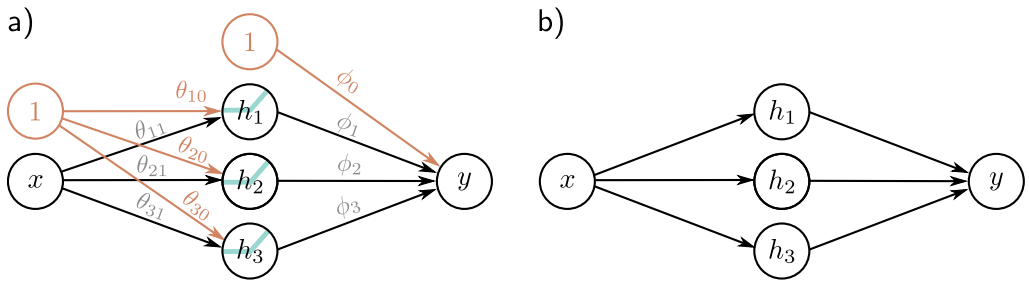


Figure 3.4 Depicting neural networks. a) The input x is on the left, the hidden units h_1, h_2, and h_3 in the center, and the output y on the right. Computation flows from left to right. The input is used to compute the hidden units, which are combined to create the output. Each of the ten arrows represents a parameter (intercepts in orange and slopes in black). Each parameter multiplies its source and adds the result to its target. For example, we multiply the parameter ϕ_1 by source h_1 and add it to y. We introduce additional nodes containing ones (orange circles) to incorporate the offsets into this scheme, so we multiply ϕ_0 by one (with no effect) and add it to y. ReLU functions are applied at the hidden units. b) More typically, the intercepts, ReLU functions, and parameter names are omitted; this simpler depiction represents the same network.

3.2 Universal approximation theorem

In the previous section, we introduced an example neural network with one input, one output, ReLU activation functions, and three hidden units. Let's now generalize this slightly and consider the case with D hidden units where the d^{th} hidden unit is:

$$h_d = \mathrm{a}[\theta_{d0} + \theta_{d1}x], \tag{3.5}$$

and these are combined linearly to create the output:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d. \tag{3.6}$$

The number of hidden units in a shallow network is a measure of the *network capacity*. With ReLU activation functions, the output of a network with D hidden units has at most D joints and so is a piecewise linear function with at most $D+1$ linear regions. As we add more hidden units, the model can approximate more complex functions.

Problem 3.10

Indeed, with enough capacity (hidden units), a shallow network can describe any continuous 1D function defined on a compact subset of the real line to arbitrary precision. To see this, consider that every time we add a hidden unit, we add another linear region to the function. As these regions become more numerous, they represent smaller sections of the function, which are increasingly well approximated by a line (figure 3.5). The *universal approximation theorem* proves that for any continuous function, there exists a shallow network that can approximate this function to any specified precision.

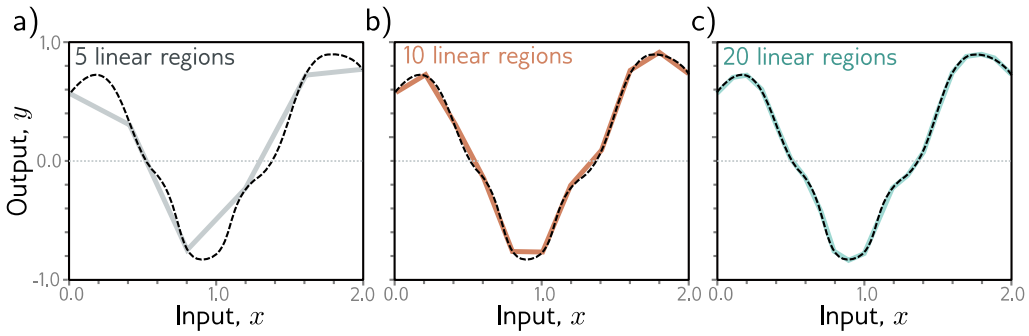


Figure 3.5 Approximation of a 1D function (dashed line) by a piecewise linear model. a–c) As the number of regions increases, the model becomes closer and closer to the continuous function. A neural network with a scalar input creates one extra linear region per hidden unit. The universal approximation theorem proves that, with enough hidden units, there exists a shallow neural network that can describe any given continuous function defined on a compact subset of $\mathbb{R}^{D_i}$ to arbitrary precision.

3.3 Multivariate inputs and outputs

In the above example, the network has a single scalar input x and a single scalar output y. However, the universal approximation theorem also holds for the more general case where the network maps multivariate inputs $\mathbf{x} = [x_1, x_2, \ldots, x_{D_i}]^T$ to multivariate output predictions $\mathbf{y} = [y_1, y_2, \ldots, y_{D_o}]^T$. We first explore how to extend the model to predict multivariate outputs. Then we consider multivariate inputs. Finally, in section 3.4, we present a general definition of a shallow neural network.

3.3.1 Visualizing multivariate outputs

To extend the network to multivariate outputs $\mathbf{y}$, we simply use a different linear function of the hidden units for each output. So, a network with a scalar input x, four hidden units h_1, h_2, h_3, and h_4, and a 2D multivariate output $\mathbf{y} = [y_1, y_2]^T$ would be defined as:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \\ h_4 &= \mathrm{a}[\theta_{40} + \theta_{41}x], \end{aligned} \tag{3.7}$$

and

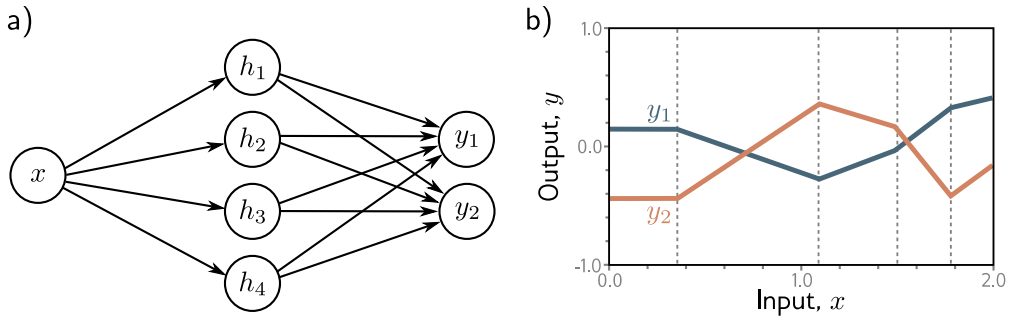


Figure 3.6 Network with one input, four hidden units, and two outputs. a) Visualization of network structure. b) This network produces two piecewise linear functions, $y_1[x]$ and $y_2[x]$. The four "joints" of these functions (at vertical dotted lines) are constrained to be in the same places since they share the same hidden units, but the slopes and overall height may differ.

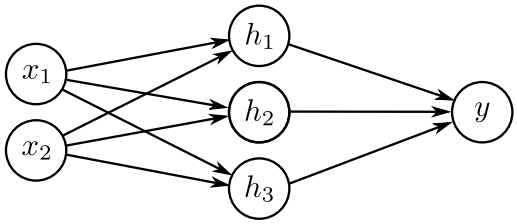


Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

$$\begin{aligned} y_1 &= \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4 \\ y_2 &= \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4. \end{aligned} \tag{3.8}$$

The two outputs are two different linear functions of the hidden units.

As we saw in figure 3.3, the "joints" in the piecewise functions depend on where the initial linear functions $\theta_{\bullet 0} + \theta_{\bullet 1}x$ are clipped by the ReLU functions $a[\bullet]$ at the hidden units. Since both outputs y_1 and y_2 are different linear functions of the same four hidden units, the four "joints" in each must be in the same places. However, the slopes of the linear regions and the overall vertical offset can differ (figure 3.6). Problem 3.11

3.3.2 Visualizing multivariate inputs

To cope with multivariate inputs $\mathbf{x}$, we extend the linear relations between the input and the hidden units. So a network with two inputs $\mathbf{x} = [x_1, x_2]^T$ and a scalar output y (figure 3.7) might have three hidden units defined by:

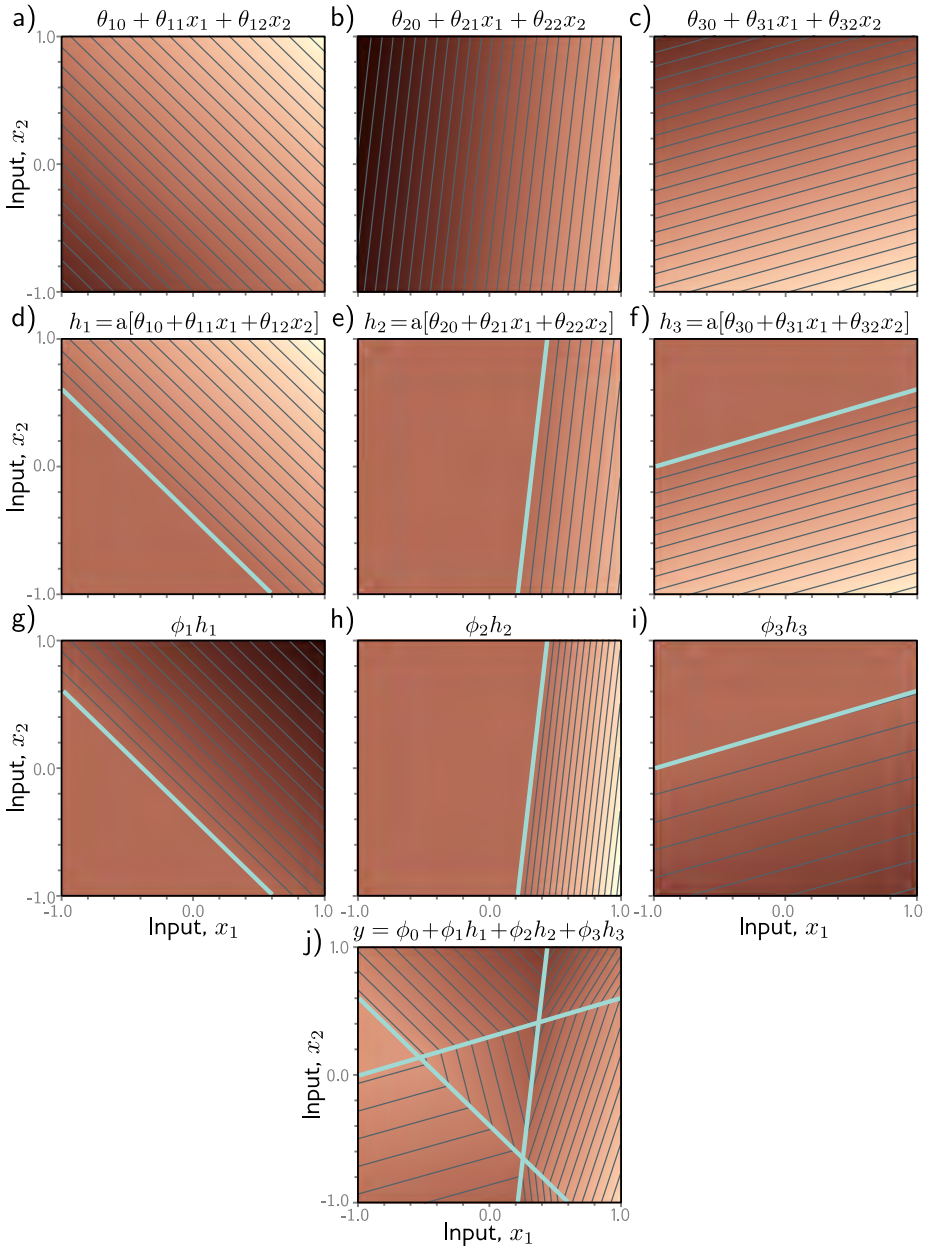


Figure 3.8 Processing in network with two inputs $\mathbf{x} = [x_1, x_2]^T$, three hidden units h_1, h_2, h_3, and one output y. a–c) The input to each hidden unit is a linear function of the two inputs, which corresponds to an oriented plane. Brightness indicates function output. For example, in panel (a), the brightness represents $\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2$. Thin lines are contours. d–f) Each plane is clipped by the ReLU activation function (cyan lines are equivalent to "joints" in figures 3.3d–f). g-i) The clipped planes are then weighted, and j) summed together with an offset that determines the overall height of the surface. The result is a continuous surface made up of convex piecewise linear polygonal regions.

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2], \end{aligned} \tag{3.9}$$

where there is now one slope parameter for each input. The hidden units are combined to form the output in the usual way:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{3.10}$$

Figure 3.8 illustrates the processing of this network. Each hidden unit receives a linear combination of the two inputs, which forms an oriented plane in the 3D input/output space. The activation function clips the negative values of these planes to zero. The clipped planes are then recombined in a second linear function (equation 3.10) to create a continuous piecewise linear surface consisting of convex polygonal regions (figure 3.8j). Each region corresponds to a different activation pattern. For example, in the central triangular region, the first and third hidden units are active, and the second is inactive.

Problems 3.12–3.13

Notebook 3.2 Shallow networks II

Appendix B.1.2 Convex region

When there are more than two inputs to the model, it becomes difficult to visualize. However, the interpretation is similar. The output will be a continuous piecewise linear function of the input, where the linear regions are now convex polytopes in the multi-dimensional input space.

Note that as the input dimensions grow, the number of linear regions increases rapidly (figure 3.9). To get a feeling for how rapidly, consider that each hidden unit defines a hyperplane that delineates the part of space where this unit is active from the part where it is not (cyan lines in 3.8d–f). If we had the same number of hidden units as input dimensions D_i, we could align each hyperplane with one of the coordinate axes (figure 3.10). For two input dimensions, this would divide the space into four quadrants. For three dimensions, this would create eight octants, and for D_i dimensions, this would create 2^{D_i} orthants. Shallow neural networks usually have more hidden units than input dimensions, so they typically create more than 2^{D_i} linear regions.

Notebook 3.3 Shallow network regions

3.4 Shallow neural networks: general case

We have described several example shallow networks to help develop intuition about how they work. We now define a general equation for a shallow neural network $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that maps a multi-dimensional input $\mathbf{x} \in \mathbb{R}^{D_i}$ to a multi-dimensional output $\mathbf{y} \in \mathbb{R}^{D_o}$ using $\mathbf{h} \in \mathbb{R}^{D}$ hidden units. Each hidden unit is computed as:

$$h_d = \mathrm{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right], \tag{3.11}$$

and these are combined linearly to create the output:

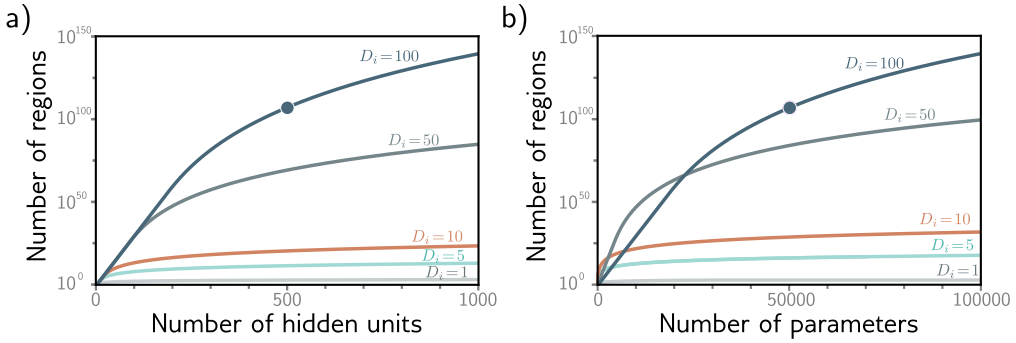


Figure 3.9 Linear regions vs. hidden units. a) Maximum possible regions as a function of the number of hidden units for five different input dimensions $D_i = \{1, 5, 10, 50, 100\}$. The number of regions increases rapidly in high dimensions; with $D = 500$ units and input size $D_i = 100$, there can be greater than 10^{107} regions (solid circle). b) The same data are plotted as a function of the number of parameters. The solid circle represents the same model as in panel (a) with $D = 500$ hidden units. This network has $51,001$ parameters and would be considered very small by modern standards.

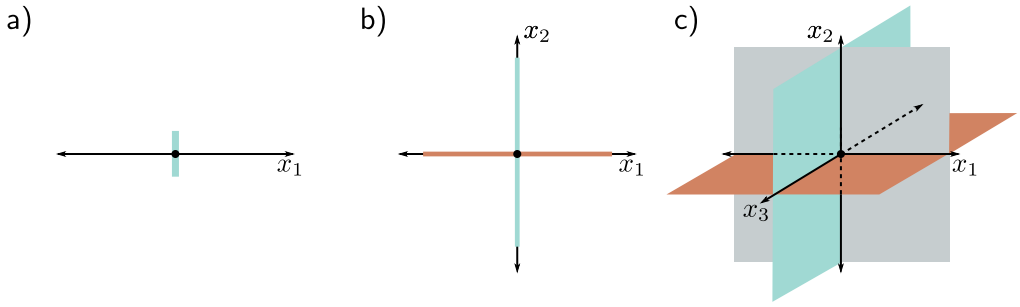


Figure 3.10 Number of linear regions vs. input dimensions. a) With a single input dimension, a model with one hidden unit creates one joint, which divides the axis into two linear regions. b) With two input dimensions, a model with two hidden units can divide the input space using two lines (here aligned with axes) to create four regions. c) With three input dimensions, a model with three hidden units can divide the input space using three planes (again aligned with axes) to create eight regions. Continuing this argument, it follows that a model with D_i input dimensions and D_i hidden units can divide the input space with D_i hyperplanes to create 2^{D_i} linear regions.

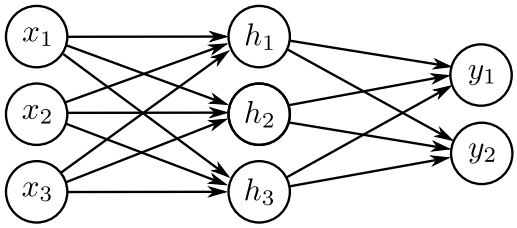


Figure 3.11 Visualization of neural network with three inputs and two outputs. This network has twenty parameters. There are fifteen slopes (indicated by arrows) and five offsets (not shown).

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d, \tag{3.12}$$

where $a[\bullet]$ is a nonlinear activation function. The model has parameters $\boldsymbol{\phi} = \{\theta_{\bullet\bullet}, \phi_{\bullet\bullet}\}$. Figure 3.11 shows an example with three inputs, three hidden units, and two outputs.

Problems 3.14–3.17

The activation function permits the model to describe nonlinear relations between input and the output, and as such, it must be nonlinear itself; with no activation function, or a linear activation function, the overall mapping from input to output would be restricted to be linear. Many different activation functions have been tried (see figure 3.13), but the most common choice is the ReLU (figure 3.1), which has the merit of being easily interpretable. With ReLU activations, the network divides the input space into convex polytopes defined by the intersections of hyperplanes computed by the "joints" in the ReLU functions. Each convex polytope contains a different linear function. The polytopes are the same for each output, but the linear functions they contain can differ.

Notebook 3.4 Activation functions

3.5 Terminology

We conclude this chapter by introducing some terminology. Regrettably, neural networks have a lot of associated jargon. They are often referred to in terms of *layers*. The left of figure 3.12 is the *input layer*, the center is the *hidden layer*, and to the right is the *output layer*. We would say that the network in figure 3.12 has one hidden layer containing four hidden units. The hidden units themselves are sometimes referred to as *neurons*. When we pass data through the network, the values of the inputs to the hidden layer (i.e., before the ReLU functions are applied) are termed *pre-activations*. The values at the hidden layer (i.e., after the ReLU functions) are termed *activations*.

For historical reasons, any neural network with at least one hidden layer is also called a *multi-layer perceptron*, or *MLP* for short. Networks with one hidden layer (as described in this chapter) are sometimes referred to as *shallow neural networks*. Networks with multiple hidden layers (as described in the next chapter) are referred to as *deep neural networks*. Neural networks in which the connections form an acyclic graph (i.e., a graph with no loops, as in all the examples in this chapter) are referred to as *feed-forward networks*. If every element in one layer connects to every element in the next (as in all the examples in this chapter), the network is *fully connected*. These connections

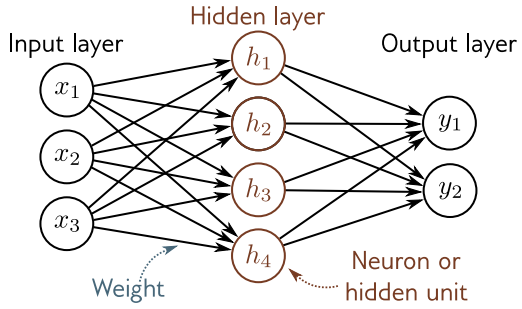


Figure 3.12 Terminology. A shallow network consists of an input layer, a hidden layer, and an output layer. Each layer is connected to the next by forward connections (arrows). For this reason, these models are referred to as feed-forward networks. When every variable in one layer connects to every variable in the next, we call this a fully connected network. Each connection represents a slope parameter in the underlying equation, and these parameters are termed weights. The variables in the hidden layer are termed neurons or hidden units. The values feeding into the hidden units are termed pre-activations, and the values at the hidden units (i.e., after the ReLU function is applied) are termed activations.

represent slope parameters in the underlying equations and are referred to as *network weights*. The offset parameters (not shown in figure 3.12) are called *biases*.

3.6 Summary

Shallow neural networks have one hidden layer. They (i) compute several linear functions of the input, (ii) pass each result through an activation function, and then (iii) take a linear combination of these activations to form the outputs. Shallow neural networks make predictions $\mathbf{y}$ based on inputs $\mathbf{x}$ by dividing the input space into a continuous surface of piecewise linear regions. With enough hidden units (neurons), shallow neural networks can approximate any continuous function to arbitrary precision.

Chapter 4 discusses deep neural networks, which extend the models from this chapter by adding more hidden layers. Chapters 5–7 describe how to train these models.

Notes

"Neural" networks: If the models in this chapter are just functions, why are they called "neural networks"? The connection is, unfortunately, tenuous. Visualizations like figure 3.12 consist of nodes (inputs, hidden units, and outputs) that are densely connected to one another. This bears a superficial similarity to neurons in the mammalian brain, which also have dense connections. However, there is scant evidence that brain computation works in the same way as neural networks, and it is unhelpful to think about biology going forward.

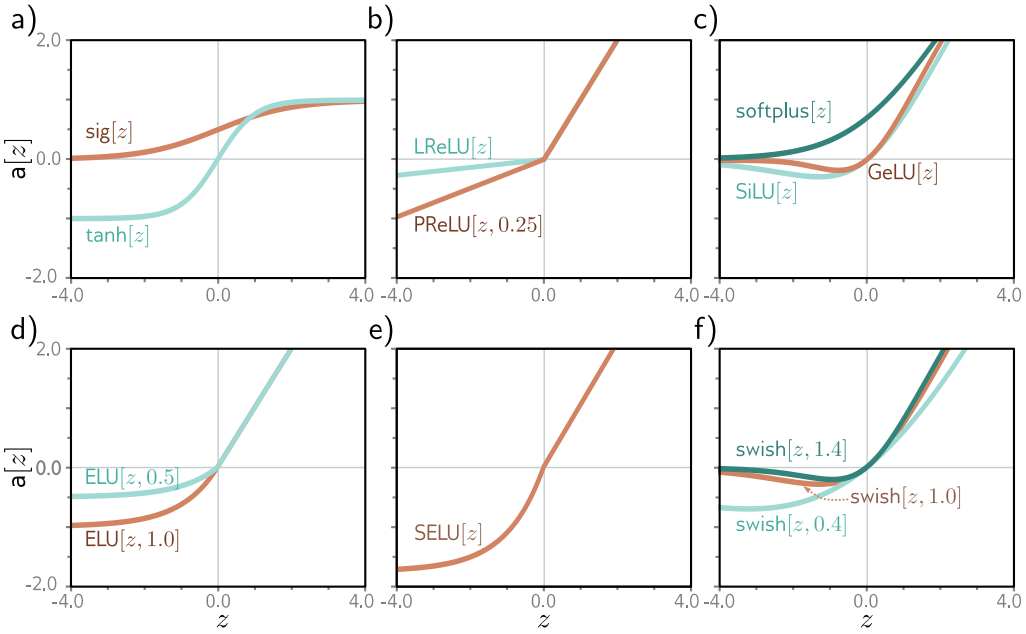


Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

History of neural networks: McCulloch & Pitts (1943) first came up with the notion of an artificial neuron that combined inputs to produce an output, but this model did not have a practical learning algorithm. Rosenblatt (1958) developed the *perceptron*, which linearly combined inputs and then thresholded them to make a yes/no decision. He also provided an algorithm to learn the weights from data. Minsky & Papert (1969) argued that the linear function was inadequate for general classification problems but that adding hidden layers with nonlinear activation functions (hence the term multi-layer perceptron) could allow the learning of more general input/output relations. However, they concluded that Rosenblatt's algorithm could not learn the parameters of such models. It was not until the 1980s that a practical algorithm (backpropagation, see chapter 7) was developed, and significant work on neural networks resumed. The history of neural networks is chronicled by Kurenkov (2020), Sejnowski (2018), and Schmidhuber (2022).

Activation functions: The ReLU function has been used as far back as Fukushima (1969). However, in the early days of neural networks, it was more common to use the logistic sigmoid or tanh activation functions (figure 3.13a). The ReLU was re-popularized by Jarrett et al. (2009), Nair & Hinton (2010), and Glorot et al. (2011) and is an important part of the success story of modern neural networks. It has the nice property that the derivative of the output with respect to the input is always one for inputs greater than zero. This contributes to the stability and efficiency of training (see chapter 7) and contrasts with the derivatives of sigmoid activation

functions, which saturate (become close to zero) for large positive and large negative inputs.

However, the ReLU function has the disadvantage that its derivative is zero for negative inputs. If all the training examples produce negative inputs to a given ReLU function, then we cannot improve the parameters feeding into this ReLU during training. The gradient with respect to the incoming weights is locally flat, so we cannot "walk downhill." This is known as the *dying ReLU* problem. Many variations on the ReLU have been proposed to resolve this problem (figure 3.13b), including (i) the leaky ReLU (Maas et al., 2013), which also has a linear output for negative values with a smaller slope of 0.1, (ii) the parametric ReLU (He et al., 2015), which treats the slope of the negative portion as an unknown parameter, and (iii) the concatenated ReLU (Shang et al., 2016), which produces two outputs, one of which clips below zero (i.e., like a typical ReLU) and one of which clips above zero.

A variety of smooth functions have also been investigated (figure 3.13c–d), including the softplus function (Glorot et al., 2011), Gaussian error linear unit (Hendrycks & Gimpel, 2016), sigmoid linear unit (Hendrycks & Gimpel, 2016), and exponential linear unit (Clevert et al., 2015). Most of these are attempts to avoid the dying ReLU problem while limiting the gradient for negative values. Klambauer et al. (2017) introduced the scaled exponential linear unit (figure 3.13e), which is particularly interesting as it helps stabilize the variance of the activations when the input variance has a limited range (see section 7.5). Ramachandran et al. (2017) adopted an empirical approach to choosing an activation function. They searched the space of possible functions to find the one that performed best over a variety of supervised learning tasks. The optimal function was found to be $a[x] = x/(1+\exp[-\beta x])$, where β is a learned parameter (figure 3.13f). They termed this function *Swish*. Interestingly, this was a rediscovery of activation functions previously proposed by Hendrycks & Gimpel (2016) and Elfwing et al. (2018). Howard et al. (2019) approximated Swish by the HardSwish function, which has a very similar shape but is faster to compute:

$$\text{HardSwish}[z] = \begin{cases} 0 & z < -3 \\ z(z+3)/6 & -3 \leq z \leq 3 \\ z & z > 3 \end{cases}. \tag{3.13}$$

There is no definitive answer as to which of these activations functions is empirically superior. However, the leaky ReLU, parameterized ReLU, and many of the continuous functions can be shown to provide minor performance gains over the ReLU in particular situations. We restrict attention to neural networks with the basic ReLU function for the rest of this book because it's easy to characterize the functions they create in terms of the number of linear regions.

Universal approximation theorem: The *width version* of this theorem states that there exists a network with one hidden layer containing a finite number of hidden units that can approximate any specified continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy. This was proved by Cybenko (1989) for a class of sigmoid activations and was later shown to be true for a larger class of nonlinear activation functions (Hornik, 1991).

Number of linear regions: Consider a shallow network with $D_i \geq 2$-dimensional inputs and D hidden units. The number of linear regions is determined by the intersections of the D hyperplanes created by the "joints" in the ReLU functions (e.g., figure 3.8d–f). Each region is created by a different combination of the ReLU functions clipping or not clipping the input. The number of regions created by D hyperplanes in the $D_i \leq D$-dimensional input space was shown by Zaslavsky (1975) to be at most $\sum_{j=0}^{D_i} \binom{D}{j}$ (i.e., a sum of binomial coefficients). As a rule of thumb, shallow neural networks almost always have a larger number D of hidden units than input dimensions D_i and create between 2^{D_i} and 2^D linear regions.

Appendix B.2 Binomial coefficient

Problem 3.18

Linear, affine, and nonlinear functions: Technically, a linear transformation f[•] is any function that obeys the principle of superposition, so $f[a+b] = f[a]+f[b]$. This definition implies that $f[2a] = 2f[a]$.The weighted sum $f[h_1, h_2, h_3] = \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$ is linear, but once the offset (bias) is added so $f[h_1, h_2, h_3] = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$, this is no longer true. To see this, consider that the output is doubled when we double the arguments of the former function. This is not the case for the latter function, which is more properly termed an *affine* function. However, it is common in machine learning to conflate these terms. We follow this convention in this book and refer to both as linear. All other functions we will encounter are nonlinear.

Problems

Problem 3.1 What kind of mapping from input to output would be created if the activation function in equation 3.1 was linear so that $a[z] = \psi_0 + \psi_1 z$? What kind of mapping would be created if the activation function was removed, so $a[z] = z$?

Problem 3.2 For each of the four linear regions in figure 3.3j, indicate which hidden units are inactive and which are active (i.e., which do and do not clip their inputs).

Problem 3.3* Derive expressions for the positions of the "joints" in function in figure 3.3j in terms of the ten parameters $\boldsymbol{\phi}$ and the input x. Derive expressions for the slopes of the four linear regions.

Problem 3.4 Draw a version of figure 3.3 where the y-intercept and slope of the third hidden unit have changed as in figure 3.14c. Assume that the remaining parameters remain the same.

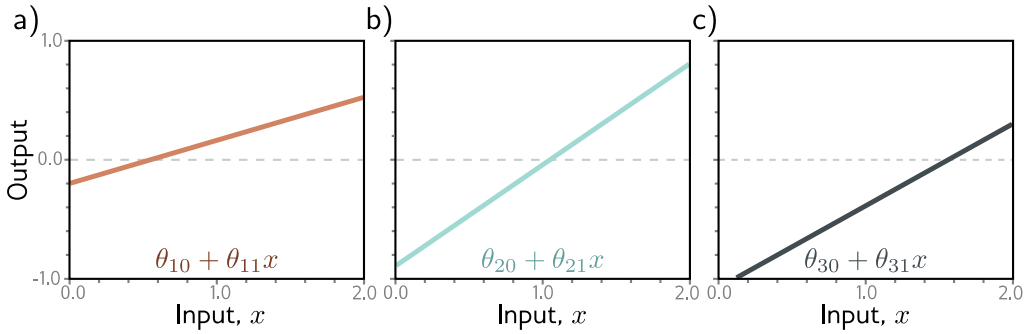


Figure 3.14 Processing in network with one input, three hidden units, and one output for problem 3.4. a–c) The input to each hidden unit is a linear function of the inputs. The first two are the same as in figure 3.3, but the last one differs.

Problem 3.5 Prove that the following property holds for $\alpha \in \mathbb{R}^+$:

$$\text{ReLU}[\alpha \cdot z] = \alpha \cdot \text{ReLU}[z]. \tag{3.14}$$

This is known as the *non-negative homogeneity* property of the ReLU function.

Problem 3.6 Following on from problem 3.5, what happens to the shallow network defined in equations 3.3 and 3.4 when we multiply the parameters θ_{10} and θ_{11} by a positive constant α and divide the slope ϕ_1 by the same parameter α? What happens if α is negative?

Problem 3.7 Consider fitting the model in equation 3.1 using a least squares loss function. Does this loss function have a unique minimum? i.e., is there a single "best" set of parameters?

Problem 3.8 Consider replacing the ReLU activation function with (i) the Heaviside step function heaviside$[z]$, (ii) the hyperbolic tangent function $\tanh[z]$, and (iii) the rectangular function rect$[z]$, where:

$$\text{heaviside}[z] = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases} \qquad \text{rect}[z] = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}. \tag{3.15}$$

Redraw a version of figure 3.3 for each of these functions. The original parameters were: $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\} = \{-0.23, -1.3, 1.3, 0.66, -0.2, 0.4, -0.9, 0.9, 1.1, -0.7\}$. Provide an informal description of the family of functions that can be created by neural networks with one input, three hidden units, and one output for each activation function.

Problem 3.9* Show that the third linear region in figure 3.3 has a slope that is the sum of the slopes of the first and fourth linear regions.

Problem 3.10 Consider a neural network with one input, one output, and three hidden units. The construction in figure 3.3 shows how this creates four linear regions. Under what circumstances could this network produce a function with fewer than four linear regions?

Problem 3.11* How many parameters does the model in figure 3.6 have?

Problem 3.12 How many parameters does the model in figure 3.7 have?

Problem 3.13 What is the activation pattern for each of the seven regions in figure 3.8? In other words, which hidden units are active (pass the input) and which are inactive (clip the input) for each region?

Problem 3.14 Write out the equations that define the network in figure 3.11. There should be three equations to compute the three hidden units from the inputs and two equations to compute the outputs from the hidden units.

Problem 3.15* What is the maximum possible number of 3D linear regions that can be created by the network in figure 3.11?

Problem 3.16 Write out the equations for a network with two inputs, four hidden units, and three outputs. Draw this model in the style of figure 3.11.

Problem 3.17* Equations 3.11 and 3.12 define a general neural network with D_i inputs, one hidden layer containing D hidden units, and D_o outputs. Find an expression for the number of parameters in the model in terms of D_i, D, and D_o.

Problem 3.18* Show that the maximum number of regions created by a shallow network with $D_i = 2$-dimensional input, $D_o = 1$-dimensional output, and $D = 3$ hidden units is seven, as in figure 3.8j. Use the result of Zaslavsky (1975) that the maximum number of regions created by partitioning a D_i-dimensional space with D hyperplanes is $\sum_{j=0}^{D_i} \binom{D}{j}$. What is the maximum number of regions if we add two more hidden units to this model, so $D = 5$?

Chapter 4

Deep neural networks

The last chapter described shallow neural networks, which have a single hidden layer. This chapter introduces deep neural networks, which have more than one hidden layer. With ReLU activation functions, both shallow and deep networks describe piecewise linear mappings from input to output.

As the number of hidden units increases, shallow neural networks improve their descriptive power. Indeed, with enough hidden units, shallow networks can describe arbitrarily complex functions in high dimensions. However, it turns out that for some functions, the required number of hidden units is impractically large. Deep networks can produce many more linear regions than shallow networks for a given number of parameters. Hence, from a practical standpoint, they can be used to describe a broader family of functions.

4.1 Composing neural networks

To gain insight into the behavior of deep neural networks, we first consider composing two shallow networks so the output of the first becomes the input of the second. Consider two shallow networks with three hidden units each (figure 4.1a). The first network takes an input x and returns output y and is defined by:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{4.1}$$

and

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{4.2}$$

The second network takes y as input and returns y' and is defined by:

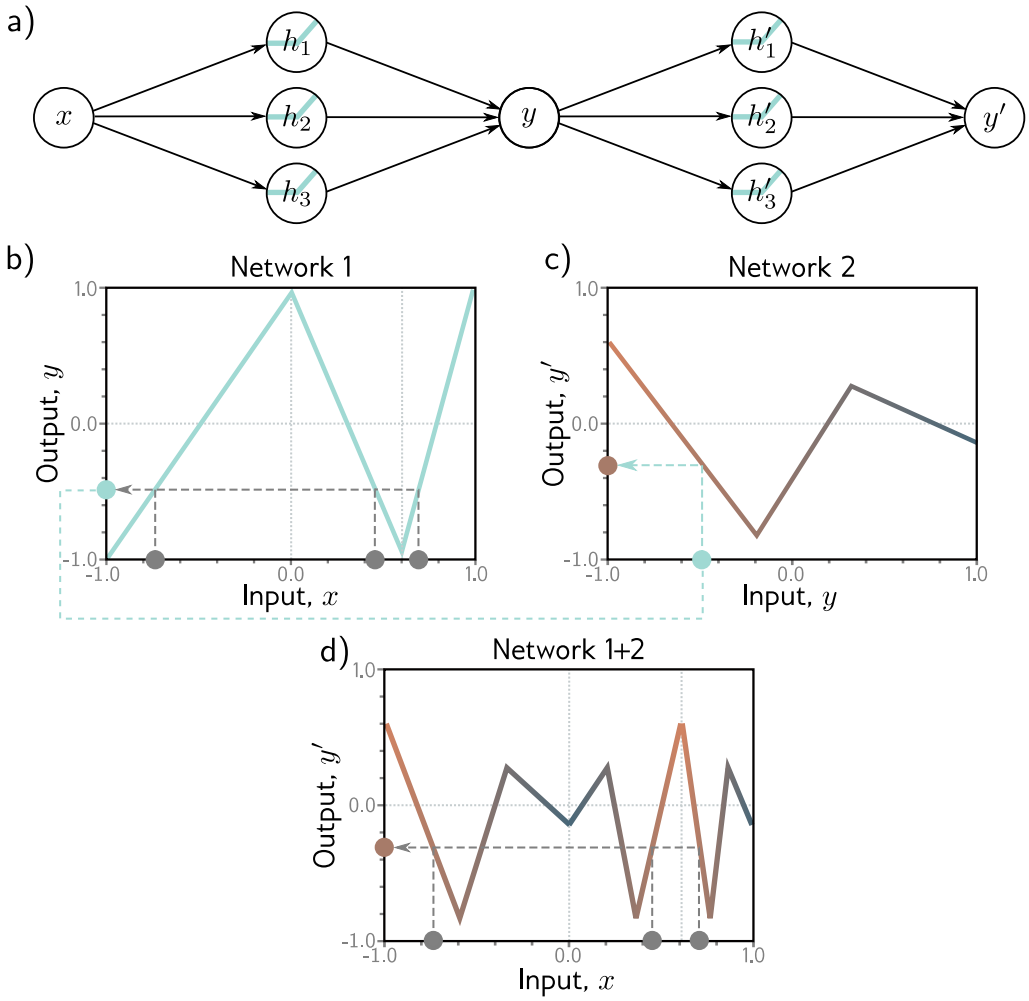


Figure 4.1 Composing two single-layer networks with three hidden units each. a) The output y of the first network constitutes the input to the second network. b) The first network maps inputs $x \in [-1, 1]$ to outputs $y \in [-1, 1]$ using a function comprising three linear regions that are chosen so that they alternate the sign of their slope (fourth linear region is outside range of graph). Multiple inputs x (gray circles) now map to the same output y (cyan circle). c) The second network defines a function comprising three linear regions that takes y and returns y' (i.e., the cyan circle is mapped to the brown circle). d) The combined effect of these two functions when composed is that (i) three different inputs x are mapped to any given value of y by the first network and (ii) are processed in the same way by the second network; the result is that the function defined by the second network in panel (c) is duplicated three times, variously flipped and rescaled according to the slope of the regions of panel (b).

$$\begin{aligned} h'_1 &= \mathrm{a}[\theta'_{10} + \theta'_{11}y] \\ h'_2 &= \mathrm{a}[\theta'_{20} + \theta'_{21}y] \\ h'_3 &= \mathrm{a}[\theta'_{30} + \theta'_{31}y], \end{aligned} \tag{4.3}$$

and

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3. \tag{4.4}$$

With ReLU activations, this model also describes a family of piecewise linear functions. However, the number of linear regions is potentially greater than for a shallow network with six hidden units. To see this, consider choosing the first network to produce three alternating regions of positive and negative slope (figure 4.1b). This means that three different ranges of x are mapped to the same output range $y \in [-1, 1]$, and the subsequent mapping from this range of y to y' is applied three times. The overall effect is that the function defined by the second network is duplicated three times to create nine linear regions. The same principle applies in higher dimensions (figure 4.2).

Problem 4.1

Notebook 4.1 Composing networks

A different way to think about composing networks is that the first network "folds" the input space x back onto itself so that multiple inputs generate the same output. Then the second network applies a function, which is replicated at all points that were folded on top of one another (figure 4.3).

4.2 From composing networks to deep networks

The previous section showed that we could create complex functions by passing the output of one shallow neural network into a second network. We now show that this is a special case of a deep network with two hidden layers.

The output of the first network ($y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$) is a linear combination of the activations at the hidden units. The first operations of the second network (equation 4.3 in which we calculate $\theta'_{10} + \theta'_{11}y$, $\theta'_{20} + \theta'_{21}y$, and $\theta'_{30} + \theta'_{31}y$) are linear in the output of the first network. Applying one linear function to another yields another linear function. Substituting the expression for y into equation 4.3 gives:

$$\begin{aligned} h'_1 &= \mathrm{a}[\theta'_{10} + \theta'_{11}y] &&= \mathrm{a}[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\ h'_2 &= \mathrm{a}[\theta'_{20} + \theta'_{21}y] &&= \mathrm{a}[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\ h'_3 &= \mathrm{a}[\theta'_{30} + \theta'_{31}y] &&= \mathrm{a}[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3], \end{aligned} \tag{4.5}$$

which we can rewrite as:

$$\begin{aligned} h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3], \end{aligned} \tag{4.6}$$

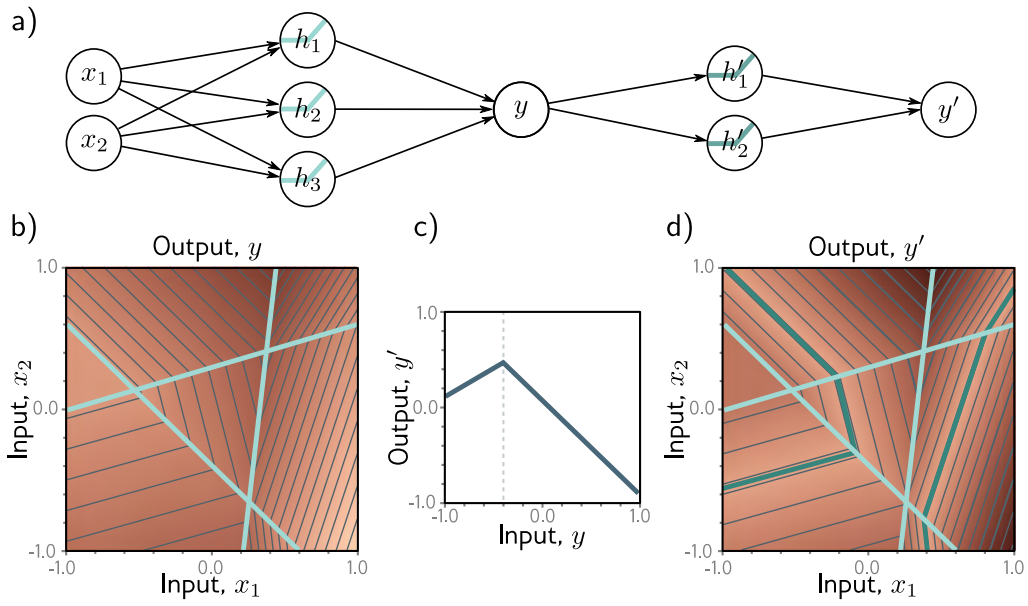


Figure 4.2 Composing neural networks with a 2D input. a) The first network (from figure 3.8) has three hidden units and takes two inputs x_1 and x_2 and returns a scalar output y. This is passed into a second network with two hidden units to produce y'. b) The first network produces a function consisting of seven linear regions, one of which is flat. c) The second network defines a function comprising two linear regions in $y \in [-1, 1]$. d) When these networks are composed, each of the six non-flat regions from the first network is divided into two new regions by the second network to create a total of 13 linear regions.

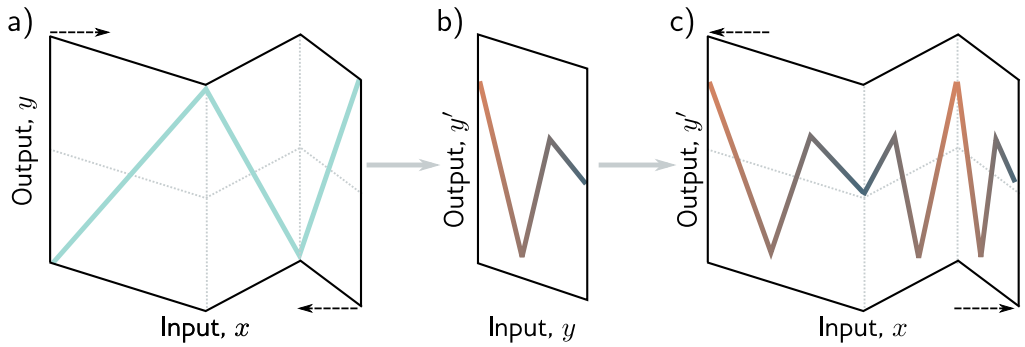


Figure 4.3 Deep networks as folding input space. a) One way to think about the first network from figure 4.1 is that it "folds" the input space back on top of itself. b) The second network applies its function to the folded space. c) The final output is revealed by "unfolding" again.

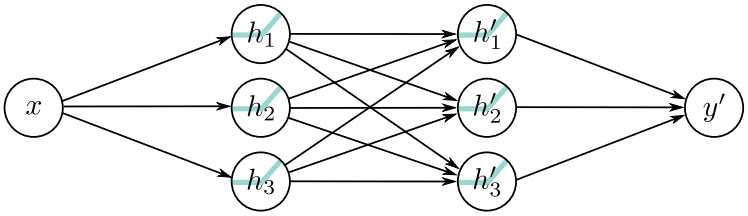


Figure 4.4 Neural network with one input, one output, and two hidden layers, each containing three hidden units.

where $\psi_{10} = \theta'_{10} + \theta'_{11}\phi_0, \psi_{11} = \theta'_{11}\phi_1, \psi_{12} = \theta'_{11}\phi_2$ and so on. The result is a network with two hidden layers (figure 4.4).

It follows that a network with two layers can represent the family of functions created by passing the output of one single-layer network into another. In fact, it represents a broader family because in equation 4.6, the nine slope parameters $\psi_{11}, \psi_{21}, \ldots, \psi_{33}$ can take arbitrary values, whereas, in equation 4.5, these parameters are constrained to be the outer product $[\theta'_{11}, \theta'_{21}, \theta'_{31}]^T[\phi_1, \phi_2, \phi_3]$.

4.3 Deep neural networks

In the previous section, we showed that composing two shallow networks yields a special case of a deep network with two hidden layers. Now we consider the general case of a deep network with two hidden layers, each containing three hidden units (figure 4.4). The first layer is defined by:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{4.7}$$

the second layer by:

$$\begin{aligned} h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3], \end{aligned} \tag{4.8}$$

and the output by:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3. \tag{4.9}$$

Notebook 4.2 Clipping functions

Considering these equations leads to another way to think about how the network constructs an increasingly complicated function (figure 4.5):

1. The three hidden units h_1, h_2, and h_3 in the first layer are computed as usual by forming linear functions of the input and passing these through ReLU activation functions (equation 4.7).
2. The pre-activations at the second layer are computed by taking three new linear functions of these hidden units (arguments of the activation functions in equation 4.8). At this point, we effectively have a shallow network with three outputs; we have computed three piecewise linear functions with the "joints" between linear regions in the same places (see figure 3.6).
3. At the second hidden layer, another ReLU function $\mathrm{a}[\bullet]$ is applied to each function (equation 4.8), which clips them and adds new "joints" to each.
4. The final output is a linear combination of these hidden units (equation 4.9).

In conclusion, we can either think of each layer as "folding" the input space or as creating new functions, which are clipped (creating new regions) and then recombined. The former view emphasizes the dependencies in the output function but not how clipping creates new joints, and the latter has the opposite emphasis. Ultimately, both descriptions provide only partial insight into how deep neural networks operate. Regardless, it's important not to lose sight of the fact that this is still merely an equation relating input x to output y'. Indeed, we can combine equations 4.7–4.9 to get one expression:

$$\begin{aligned} y' = \; & \phi_0' + \phi_1' \mathrm{a}\left[\psi_{10} + \psi_{11}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{12}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{13}\mathrm{a}[\theta_{30} + \theta_{31}x]\right] \\ & + \phi_2' \mathrm{a}[\psi_{20} + \psi_{21}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{23}\mathrm{a}[\theta_{30} + \theta_{31}x]] \\ & + \phi_3' \mathrm{a}[\psi_{30} + \psi_{31}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{32}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{33}\mathrm{a}[\theta_{30} + \theta_{31}x]], \end{aligned} \tag{4.10}$$

although this is admittedly rather difficult to understand.

4.3.1 Hyperparameters

We can extend the deep network construction to more than two hidden layers; modern networks might have more than a hundred layers with thousands of hidden units at each layer. The number of hidden units in each layer is referred to as the *width* of the network, and the number of hidden layers as the *depth*. The total number of hidden units is a measure of the network's *capacity*.

Problem 4.2

We denote the number of layers as K and the number of hidden units in each layer as $D_1, D_2, \ldots, D_K$. These are examples of *hyperparameters*. They are quantities chosen before we learn the model parameters (i.e., the slope and intercept terms). For fixed hyperparameters (e.g., $K = 2$ layers with $D_k = 3$ hidden units in each), the model describes a family of functions, and the parameters determine the particular function. Hence, when we also consider the hyperparameters, we can think of neural networks as representing a family of families of functions relating input to output.

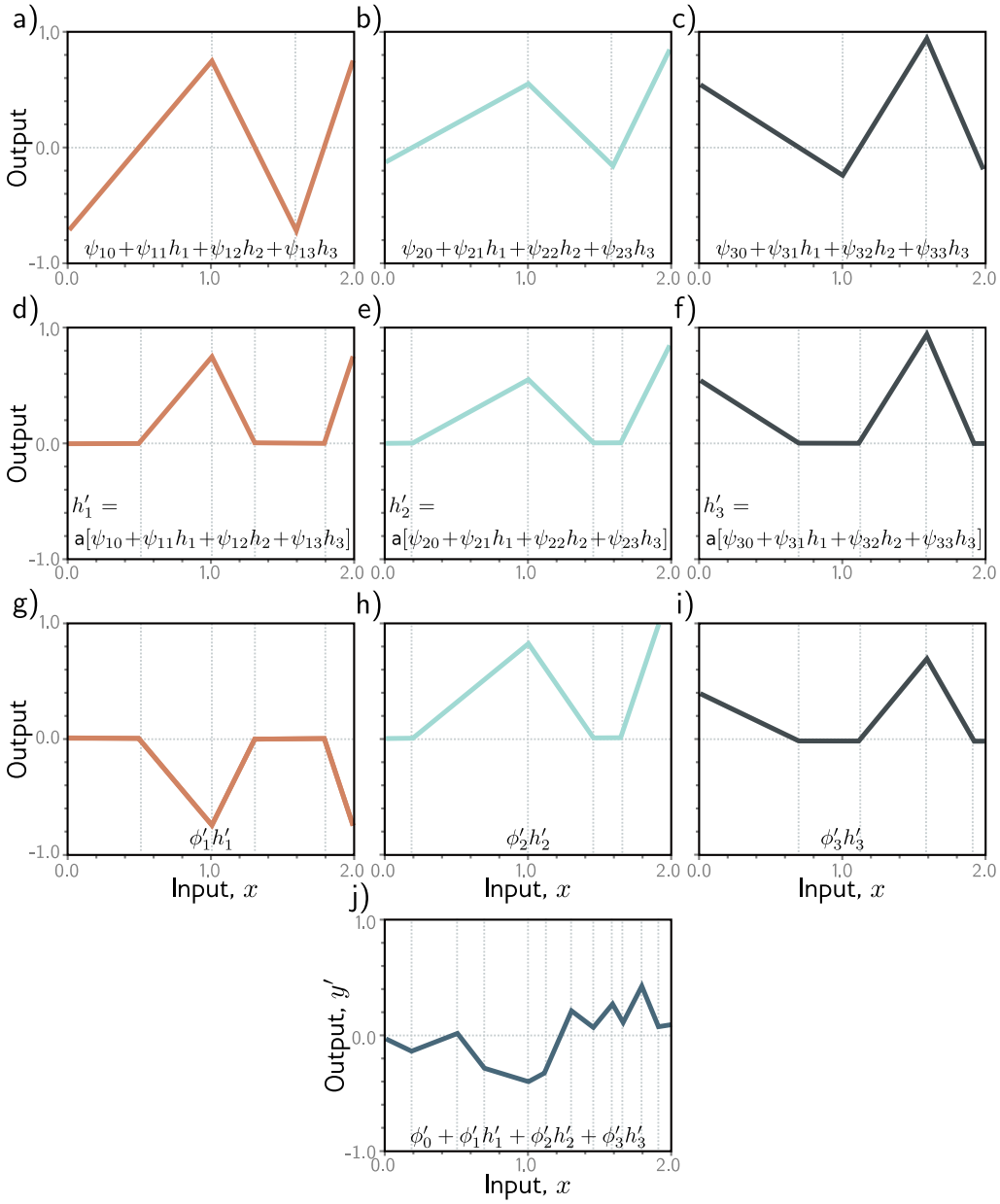


Figure 4.5 Computation for the deep network in figure 4.4. a–c) The inputs to the second hidden layer (i.e., the pre-activations) are three piecewise linear functions where the "joints" between the linear regions are at the same places (see figure 3.6). d–f) Each piecewise linear function is clipped to zero by the ReLU activation function. g–i) These clipped functions are then weighted with parameters ϕ'_1, ϕ'_2, and ϕ'_3, respectively. j) Finally, the clipped and weighted functions are summed and an offset ϕ'_0 that controls the overall height is added.

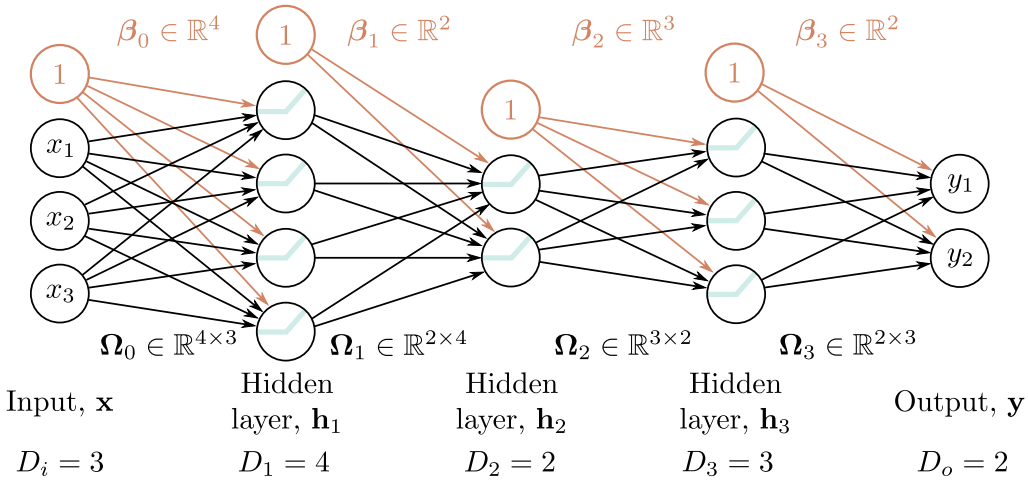


Figure 4.6 Matrix notation for network with $D_i = 3$-dimensional input $\mathbf{x}$, $D_o = 2$-dimensional output $\mathbf{y}$, and $K = 3$ hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$ of dimensions $D_1 = 4$, $D_2 = 2$, and $D_3 = 3$ respectively. The weights are stored in matrices $\boldsymbol{\Omega}_k$ that pre-multiply the activations from the preceding layer to create the pre-activations at the subsequent layer. For example, the weight matrix $\boldsymbol{\Omega}_1$ that computes the pre-activations at $\mathbf{h}_2$ from the activations at $\mathbf{h}_1$ has dimension 2×4. It is applied to the four hidden units in layer one and creates the inputs to the two hidden units at layer two. The biases are stored in vectors $\boldsymbol{\beta}_k$ and have the dimension of the layer into which they feed. For example, the bias vector $\boldsymbol{\beta}_2$ is length three because layer $\mathbf{h}_3$ contains three hidden units.

4.4 Matrix notation

Appendix B.3 Matrices

We have seen that a deep neural network consists of linear transformations alternating with activation functions. We could equivalently describe equations 4.7–4.9 in matrix notation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right], \tag{4.11}$$

$$\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right], \tag{4.12}$$

and

$$y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix}, \tag{4.13}$$

or even more compactly in matrix notation as:

$$\begin{aligned}\mathbf{h} &= \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\theta} x\right] \\ \mathbf{h}' &= \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right] \\ y' &= \phi_0' + \boldsymbol{\phi}'\mathbf{h}', \end{aligned} \tag{4.14}$$

where, in each case, the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of its vector input.

4.4.1 General formulation

This notation becomes cumbersome for networks with many layers. Hence, from now on, we will describe the vector of hidden units at layer k as $\mathbf{h}_k$, the vector of biases (intercepts) that contribute to hidden layer $k+1$ as $\boldsymbol{\beta}_k$, and the weights (slopes) that are applied to the k^{th} layer and contribute to the $(k+1)^{th}$ layer as $\boldsymbol{\Omega}_k$. A general deep network $\mathbf{y} = \mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with K layers can now be written as:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\ \mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K. \end{aligned} \tag{4.15}$$

The parameters $\boldsymbol{\phi}$ of this model comprise all of these weight matrices and bias vectors $\boldsymbol{\phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k=0}^{K}$.

If the k^{th} layer has D_k hidden units, then the bias vector $\boldsymbol{\beta}_{k-1}$ will be of size D_k. The last bias vector $\boldsymbol{\beta}_K$ has the size D_o of the output. The first weight matrix $\boldsymbol{\Omega}_0$ has size $D_1 \times D_i$ where D_i is the size of the input. The last weight matrix $\boldsymbol{\Omega}_K$ is $D_o \times D_K$, and the remaining matrices $\boldsymbol{\Omega}_k$ are $D_{k+1} \times D_k$ (figure 4.6).

Notebook 4.3
Deep networks

Problems 4.3–4.6

We can equivalently write the network as a single function:

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{a}\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{a}\left[\ldots\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{a}\left[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}\right]\right]\ldots\right]\right]. \tag{4.16}$$

4.5 Shallow vs. deep neural networks

Chapter 3 discussed shallow networks (with a single hidden layer), and here we have described deep networks (with multiple hidden layers). We now compare these models.

4.5.1 Ability to approximate different functions

In section 3.2, we argued that shallow neural networks with enough capacity (hidden units) could model any continuous function arbitrarily closely. In this chapter, we saw that a deep network with two hidden layers could represent the composition of two shallow networks. If the second of these networks computes the identity function, then this deep network replicates a single shallow network. Hence, it can also approximate any continuous function arbitrarily closely given sufficient capacity.

Problem 4.7

4.5.2 Number of linear regions per parameter

A shallow network with one input, one output, and $D > 2$ hidden units can create up to $D + 1$ linear regions and is defined by $3D + 1$ parameters. A deep network with one input, one output, and K layers of $D > 2$ hidden units can create a function with up to $(D + 1)^K$ linear regions using $3D + 1 + (K - 1)D(D + 1)$ parameters.

Problems 4.8–4.11

Figure 4.7a shows how the maximum number of linear regions increases as a function of the number of parameters for networks mapping scalar input x to scalar output y. Deep neural networks create much more complex functions for a fixed parameter budget. This effect is magnified as the number of input dimensions D_i increases (figure 4.7b), although computing the maximum number of regions is less straightforward.

This seems attractive, but the flexibility of the functions is still limited by the number of parameters. Deep networks can create extremely large numbers of linear regions, but these contain complex dependencies and symmetries. We saw some of these when we considered deep networks as "folding" the input space (figure 4.3). So, it's not clear that the greater number of regions is an advantage unless (i) there are similar symmetries in the real-world functions that we wish to approximate or (ii) we have reason to believe that the mapping from input to output really does involve a composition of simpler functions.

4.5.3 Depth efficiency

Both deep and shallow networks can model arbitrary functions, but some functions can be approximated much more efficiently with deep networks. Functions have been identified that require a shallow network with exponentially more hidden units to achieve an equivalent approximation to that of a deep network. This phenomenon is referred to as the *depth efficiency* of neural networks. This property is also attractive, but it's not clear that the real-world functions that we want to approximate fall into this category.

4.5.4 Large, structured inputs

We have discussed fully connected networks where every element of each layer contributes to every element of the subsequent one. However, these are not practical for large,

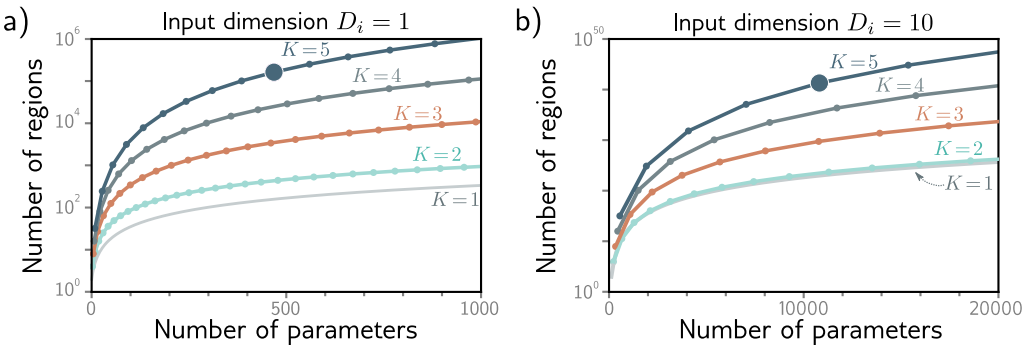


Figure 4.7 The maximum number of linear regions for neural networks increases rapidly with the network depth. a) Network with $D_i = 1$ input. Each curve represents a fixed number of hidden layers K, as we vary the number of hidden units D per layer. For a fixed parameter budget (horizontal position), deeper networks produce more linear regions than shallower ones. A network with $K = 5$ layers and $D = 10$ hidden units per layer has 471 parameters (highlighted point) and can produce 161,051 regions. b) Network with $D_i = 10$ inputs. Each subsequent point along a curve represents ten hidden units. Here, a model with $K = 5$ layers and $D = 50$ hidden units per layer has 10,801 parameters (highlighted point) and can create more than 10^{40} linear regions.

structured inputs like images, where the input might comprise $\sim 10^6$ pixels. The number of parameters would be prohibitive, and moreover, we want different parts of the image to be processed similarly; there is no point in independently learning to recognize the same object at every possible position in the image.

The solution is to process local image regions in parallel and then gradually integrate information from increasingly large regions. This kind of local-to-global processing is difficult to specify without using multiple layers (see chapter 10).

4.5.5 Training and generalization

A further possible advantage of deep networks over shallow networks is their ease of fitting; it is usually easier to train moderately deep networks than to train shallow ones (see figure 20.2). It may be that over-parameterized deep models (i.e., those with more parameters than training examples) have a large family of roughly equivalent solutions that are easy to find. However, as we add more hidden layers, training becomes more difficult again. Many methods have been developed to mitigate this problem (see chapter 11).

Deep neural networks also seem to generalize to new data better than shallow ones. In practice, the best results for most tasks have been achieved using networks with tens or hundreds of layers. Neither of these phenomena are well understood, and we return to them in chapter 20.

4.6 Summary

In this chapter, we first considered what happens when we compose two shallow networks. We argued that the first network "folds" the input space, and the second network then applies a piecewise linear function. The effects of the second network are duplicated where the input space is folded onto itself.

We then showed that this composition of shallow networks is a special case of a deep network with two layers. We interpreted the ReLU functions in each layer as clipping the input functions in multiple places and creating more "joints" in the output function. We introduced the idea of hyperparameters, which for the networks we've seen so far, comprise the number of hidden layers and the number of hidden units in each.

Finally, we compared shallow and deep networks. We saw that (i) both networks can approximate any function given enough capacity, (ii) deep networks produce many more linear regions per parameter, (iii) some functions can be approximated much more efficiently by deep networks, (iv) large, structured inputs like images are best processed in multiple stages, and (v) in practice, the best results for most tasks are achieved using deep networks with many layers.

Now that we understand deep and shallow network models, we turn our attention to training them. In the next chapter, we discuss loss functions. For any given parameter values $\boldsymbol{\phi}$, the loss function returns a single number that indicates the mismatch between the model outputs and the ground truth predictions for a training dataset. In chapters 6 and 7, we deal with the training process itself, in which we seek the parameter values that minimize this loss.

Notes

Deep learning: It has long been understood that it is possible to build more complex functions by composing shallow neural networks or developing networks with more than one hidden layer. Indeed, the term "deep learning" was first used by Dechter (1986). However, interest was limited due to practical concerns; it was not possible to train such networks well. The modern era of deep learning was kick-started by startling improvements in image classification reported by Krizhevsky et al. (2012). This sudden progress was arguably due to the confluence of four factors: larger training datasets, improved processing power for training, the use of the ReLU activation function, and the use of stochastic gradient descent (see chapter 6). LeCun et al. (2015) present an overview of early advances in the modern era of deep learning.

Number of linear regions: For deep networks using a total of D hidden units with ReLU activations, the upper bound on the number of regions is 2^D (Montúfar et al., 2014). The same authors show that a deep ReLU network with D_i-dimensional input and K layers, each containing $D \geq D_i$ hidden units, has $\mathcal{O}\left((D/D_i)^{(K-1)D_i} D^{D_i}\right)$ linear regions. Montúfar (2017), Arora et al. (2016) and Serra et al. (2018) all provide tighter upper bounds that consider the possibility that each layer has different numbers of hidden units. Serra et al. (2018) provide an algorithm that counts the number of linear regions in a neural network, although it is only practical for very small networks.

If the number of hidden units D in each of the K layers is the same, and D is an integer multiple of the input dimensionality D_i, then the maximum number of linear regions N_r can be

computed exactly and is:

$$N_r = \left(\frac{D}{D_i}+1\right)^{D_i(K-1)} \cdot \sum_{j=0}^{D_i}\binom{D}{j}. \tag{4.17}$$

The first term in this expression corresponds to the first $K-1$ layers of the network, which can be thought of as repeatedly folding the input space. However, we now need to devote D/D_i hidden units to each input dimension to create these folds. The last term in this equation (a sum of binomial coefficients) is the number of regions that a shallow network can create and is attributable to the last layer. For further information, consult Montúfar et al. (2014), Pascanu et al. (2013), and Montúfar (2017).

Appendix B.2 Binomial coefficient

Universal approximation theorem: We argued in section 4.5.1 that if the layers of a deep network have enough hidden units, then the width version of the universal approximation theorem applies: there exists a network that can approximate any given continuous function on a compact subset of $\mathbb{R}^{D_i}$ to arbitrary accuracy. Lu et al. (2017) proved that there exists a network with ReLU activation functions and at least D_i+4 hidden units in each layer can approximate any specified D_i-dimensional Lebesgue integrable function to arbitrary accuracy given enough layers. This is known as the *depth version* of the universal approximation theorem.

Depth efficiency: Several results show that there are functions that can be realized by deep networks but not by any shallow network whose capacity is bounded above exponentially. In other words, it would take an exponentially larger number of units in a shallow network to describe these functions accurately. This is known as the *depth efficiency* of neural networks.

Telgarsky (2016) shows that for any integer k, it is possible to construct networks with one input, one output, and $\mathcal{O}[k^3]$ layers of constant width, which cannot be realized with $\mathcal{O}[k]$ layers and less than 2^k width. Perhaps surprisingly, Eldan & Shamir (2016) showed that when there are multivariate inputs, there is a three-layer network that cannot be realized by any two-layer network if the capacity is sub-exponential in the input dimension. Cohen et al. (2016), Safran & Shamir (2017), and Poggio et al. (2017) also demonstrate functions that deep networks can approximate efficiently, but shallow ones cannot. Liang & Srikant (2016) show that for a broad class of functions, including univariate functions, shallow networks require exponentially more hidden units than deep networks for a given upper bound on the approximation error.

Width efficiency: Lu et al. (2017) investigate whether there are wide shallow networks (i.e., shallow networks with lots of hidden units) that cannot be realized by narrow networks whose depth is not substantially larger. They show that there exist classes of wide, shallow networks that can only be expressed by narrow networks with polynomial depth. This is known as the *width efficiency* of neural networks. This polynomial lower bound on width is less restrictive than the exponential lower bound on depth, suggesting that depth is more important. Vardi et al. (2022) subsequently showed that the price for making the width small is only a linear increase in the network depth for networks with ReLU activations.

Problems

Problem 4.1* Consider composing the two neural networks in figure 4.8. Draw a plot of the relationship between the input x and output y' for $x \in [-1, 1]$.

Problem 4.2 Identify the four hyperparameters in figure 4.6.

Problem 4.3 Using the non-negative homogeneity property of the ReLU function (see problem 3.5), show that:

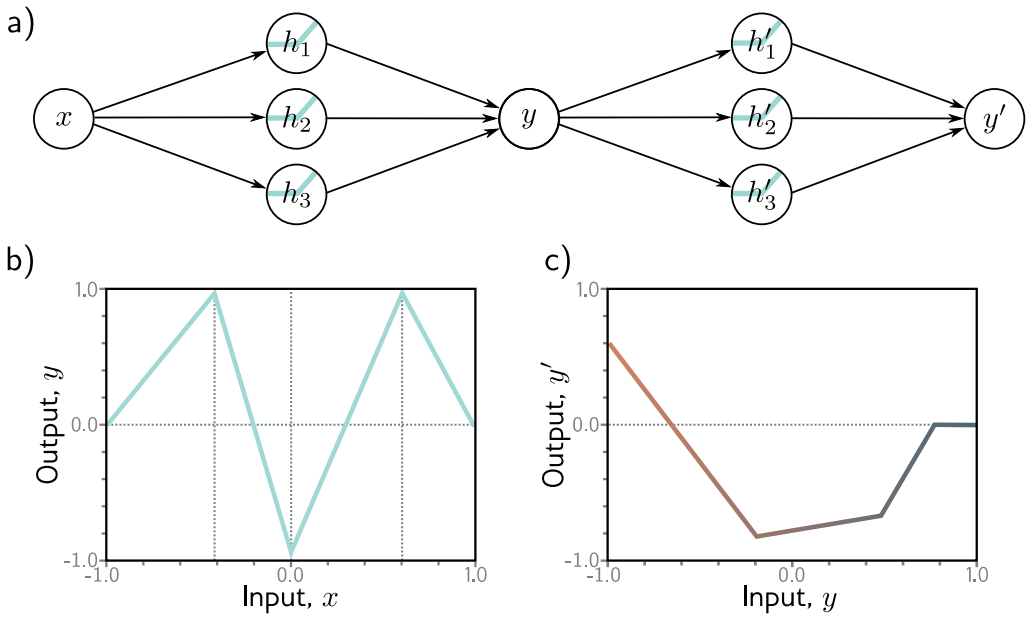


Figure 4.8 Composition of two networks for problem 4.1. a) The output y of the first network becomes the input to the second. b) The first network computes this function with output values $y \in [-1, 1]$. c) The second network computes this function on the input range $y \in [-1, 1]$.

$$\text{ReLU}\Big[\boldsymbol{\beta}_1+\lambda_1\cdot\boldsymbol{\Omega}_1\text{ReLU}\left[\boldsymbol{\beta}_0+\lambda_0\cdot\boldsymbol{\Omega}_0\mathbf{x}\right]\Big]=\lambda_0\lambda_1\cdot\text{ReLU}\left[\frac{1}{\lambda_0\lambda_1}\boldsymbol{\beta}_1+\boldsymbol{\Omega}_1\text{ReLU}\left[\frac{1}{\lambda_0}\boldsymbol{\beta}_0+\boldsymbol{\Omega}_0\mathbf{x}\right]\right], \tag{4.18}$$

where λ_0 and λ_1 are non-negative scalars. From this, we see that the weight matrices can be rescaled by any magnitude as long as the biases are also adjusted, and the scale factors can be re-applied at the end of the network.

Problem 4.4 Write out the equations for a deep neural network that takes $D_i = 5$ inputs, $D_o = 4$ outputs and has three hidden layers of sizes $D_1 = 20$, $D_2 = 10$, and $D_3 = 7$, respectively, in both the forms of equations 4.15 and 4.16. What are the sizes of each weight matrix $\boldsymbol{\Omega}_\bullet$ and bias vector $\boldsymbol{\beta}_\bullet$?

Problem 4.5 Consider a deep neural network with $D_i = 5$ inputs, $D_o = 1$ output, and $K = 20$ hidden layers containing $D = 30$ hidden units each. What is the depth of this network? What is the width?

Problem 4.6 Consider a network with $D_i = 1$ input, $D_o = 1$ output, and $K = 10$ layers, with $D = 10$ hidden units in each. Would the number of weights increase more if we increased the depth by one or the width by one? Provide your reasoning.

Problem 4.7 Choose values for the parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$ for the shallow neural network in equation 3.1 (with ReLU activation functions) that will define an identity function over a finite range $x \in [a, b]$.

Problem 4.8* Figure 4.9 shows the activations in the three hidden units of a shallow network (as in figure 3.3). The slopes in the hidden units are 1.0, 1.0, and -1.0, respectively, and the "joints" in the hidden units are at positions 1/6, 2/6, and 4/6. Find values of ϕ_0, ϕ_1, ϕ_2, and ϕ_3 that will combine the hidden unit activations as $\phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$ to create a function with four linear regions that oscillate between output values of zero and one. The slope of the leftmost region should be positive, the next one negative, and so on. How many linear regions will we create if we compose this network with itself? How many will we create if we compose it with itself K times?

Problem 4.9* Following problem 4.8, is it possible to create a function with three linear regions that oscillates back and forth between output values of zero and one using a shallow network with two hidden units? Is it possible to create a function with five linear regions that oscillates in the same way using a shallow network with four hidden units?

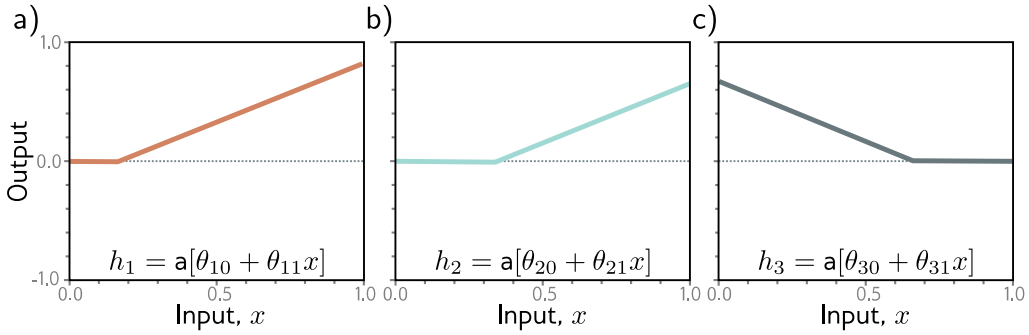


Figure 4.9 Hidden unit activations for problem 4.8. a) First hidden unit has a joint at position $x = 1/6$ and a slope of one in the active region. b) Second hidden unit has a joint at position $x = 2/6$ and a slope of one in the active region. c) Third hidden unit has a joint at position $x = 4/6$ and a slope of minus one in the active region.

Problem 4.10 Consider a deep neural network with a single input, a single output, and K hidden layers, each of which contains D hidden units. Show that this network will have a total of $3D + 1 + (K - 1)D(D + 1)$ parameters.

Problem 4.11* Consider two neural networks that map a scalar input x to a scalar output y. The first network is shallow and has $D = 95$ hidden units. The second is deep and has $K = 10$ layers, each containing $D = 5$ hidden units. How many parameters does each network have? How many linear regions can each network make (see equation 4.17)? Which would run faster?

Chapter 5

Loss functions

The last three chapters described linear regression, shallow neural networks, and deep neural networks. Each represents a family of functions that map input to output, where the particular member of the family is determined by the model parameters $\boldsymbol{\phi}$. When we train these models, we seek the parameters that produce the best possible mapping from input to output for the task we are considering. This chapter defines what is meant by the "best possible" mapping.

That definition requires a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. A *loss function* or *cost function* $L[\boldsymbol{\phi}]$ returns a single number that describes the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and their corresponding ground-truth outputs $\mathbf{y}_i$. During training, we seek parameter values $\boldsymbol{\phi}$ that minimize the loss and hence map the training inputs to the outputs as closely as possible. We saw one example of a loss function in chapter 2; the least squares loss function is suitable for univariate regression problems for which the target is a real number $y \in \mathbb{R}$. It computes the sum of the squares of the deviations between the model predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and the true values y_i.

Appendix A Number sets

This chapter provides a framework that both justifies the choice of the least squares criterion for real-valued outputs and allows us to build loss functions for other prediction types. We consider *binary classification*, where the prediction $y \in \{0, 1\}$ is one of two categories, *multiclass classification*, where the prediction $y \in \{1, 2, \ldots, K\}$ is one of K categories, and more complex cases. In the following two chapters, we address model training, where the goal is to find the parameter values that minimize these loss functions.

5.1 Maximum likelihood

In this section, we develop a recipe for constructing loss functions. Consider a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that computes an output from input $\mathbf{x}$. Until now, we have implied that the model directly computes a prediction $\mathbf{y}$. We now shift perspective and consider the model as computing a conditional probability distribution $Pr(\mathbf{y}|\mathbf{x})$ over possible outputs $\mathbf{y}$ given input $\mathbf{x}$. The loss encourages each training output $\mathbf{y}_i$ to have a high probability under the distribution $Pr(\mathbf{y}_i|\mathbf{x}_i)$ computed from the corresponding input $\mathbf{x}_i$ (figure 5.1).

Appendix C.1.3 Conditional probability

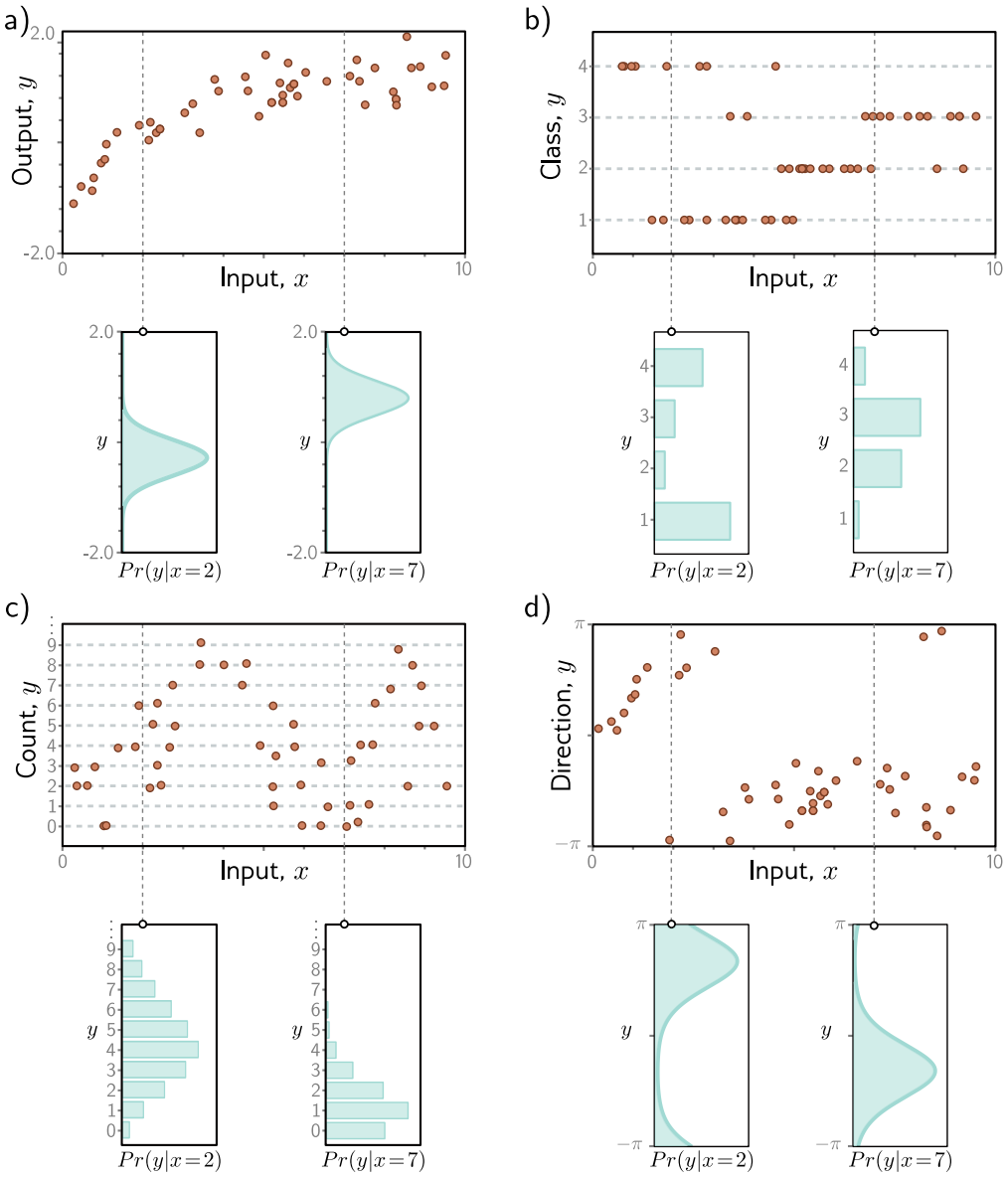


Figure 5.1 Predicting distributions over outputs. a) Regression task, where the goal is to predict a real-valued output y from the input x based on training data $\{x_i, y_i\}$ (orange points). For each input value x, the machine learning model predicts a distribution $Pr(y|x)$ over the output $y \in \mathbb{R}$ (cyan curves show distributions for $x = 2.0$ and $x = 7.0$). The loss function aims to maximize the probability of the observed training outputs y_i under the distribution predicted from the corresponding inputs x_i. b) To predict discrete classes $y \in \{1, 2, 3, 4\}$ in a classification task, we use a discrete probability distribution, so the model predicts a different histogram over the four possible values of y_i for each value of x_i. c) To predict counts $y \in \{0, 1, 2, \ldots\}$ and d) direction $y \in (-\pi, \pi]$, we use distributions defined over positive integers and circular domains, respectively.

5.1.1 Computing a distribution over outputs

This raises the question of exactly how a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ can be adapted to compute a probability distribution. The solution is simple. First, we choose a parametric distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ defined on the output domain $\mathbf{y}$. Then we use the network to compute one or more of the parameters $\boldsymbol{\theta}$ of this distribution.

For example, suppose the prediction domain is the set of real numbers, so $y \in \mathbb{R}$. Here, we might choose the univariate normal distribution, which is defined on $\mathbb{R}$. This distribution is defined by the mean μ and variance σ^2, so $\boldsymbol{\theta} = \{\mu, \sigma^2\}$. The machine learning model might predict the mean μ, and the variance σ^2 could be treated as an unknown constant.

5.1.2 Maximum likelihood criterion

The model now computes different distribution parameters $\boldsymbol{\theta}_i = \mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for each training input $\mathbf{x}_i$. Each observed training output $\mathbf{y}_i$ should have high probability under its corresponding distribution $Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)$. Hence, we choose the model parameters $\boldsymbol{\phi}$ so that they maximize the combined probability across all I training examples:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]. \end{aligned} \tag{5.1}$$

The combined probability term is the *likelihood* of the parameters, and hence equation 5.1 is known as the *maximum likelihood* criterion.[1]

Here we are implicitly making two assumptions. First, we assume that the data are identically distributed (the form of the probability distribution over the outputs $\mathbf{y}_i$ is the same for each data point). Second, we assume that the conditional distributions $Pr(\mathbf{y}_i|\mathbf{x}_i)$ of the output given the input are independent, so the total likelihood of the training data decomposes as:

Appendix C.1.5 Independence

$$Pr(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_I|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_I) = \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i). \tag{5.2}$$

In other words, we assume the data are *independent and identically distributed* (*i.i.d.*).

[1] A conditional probability $Pr(z|\psi)$ can be considered in two ways. As a function of z, it is a probability distribution that sums to one. As a function of ψ, it is known as a *likelihood* and does not generally sum to one.

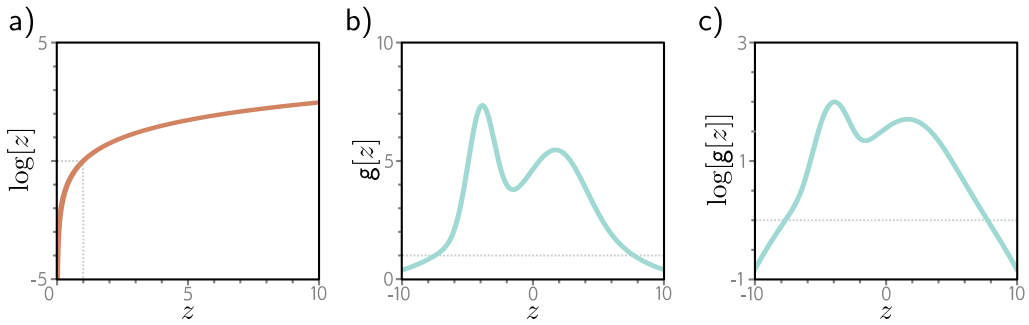


Figure 5.2 The log transform. a) The log function is monotonically increasing. If $z > z'$, then $\log[z] > \log[z']$. It follows that the maximum of any function $g[z]$ will be at the same position as the maximum of $\log[g[z]]$. b) A function $g[z]$. c) The logarithm of this function $\log[g[z]]$. All positions on $g[z]$ with a positive slope retain a positive slope after the log transform, and those with a negative slope retain a negative slope. The position of the maximum remains the same.

5.1.3 Maximizing log-likelihood

The maximum likelihood criterion (equation 5.1) is not very practical. Each term $Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])$ can be small, so the product of many of these terms can be tiny. It may be difficult to represent this quantity with finite precision arithmetic. Fortunately, we can equivalently maximize the logarithm of the likelihood:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\text{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right] \\ &= \underset{\boldsymbol{\phi}}{\text{argmax}}\left[\log\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]\right] \\ &= \underset{\boldsymbol{\phi}}{\text{argmax}}\left[\sum_{i=1}^{I} \log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\Big]\right]. \qquad (5.3)\end{aligned}$$

This *log-likelihood* criterion is equivalent because the logarithm is a monotonically increasing function: if $z > z'$, then $\log[z] > \log[z']$ and vice versa (figure 5.2). It follows that when we change the model parameters $\boldsymbol{\phi}$ to improve the log-likelihood criterion, we also improve the original maximum likelihood criterion. It also follows that the overall maxima of the two criteria must be in the same place, so the best model parameters $\hat{\boldsymbol{\phi}}$ are the same in both cases. However, the log-likelihood criterion has the practical advantage of using a sum of terms, not a product, so representing it with finite precision isn't problematic.

5.1.4 Minimizing negative log-likelihood

Finally, we note that, by convention, model fitting problems are framed in terms of minimizing a loss. To convert the maximum log-likelihood criterion to a minimization problem, we multiply by minus one, which gives us the *negative log-likelihood criterion*:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\Big]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L[\boldsymbol{\phi}]\Big], \end{aligned} \tag{5.4}$$

which is what forms the final loss function $L[\boldsymbol{\phi}]$.

5.1.5 Inference

The network no longer directly predicts the outputs $\mathbf{y}$ but instead determines a probability distribution over $\mathbf{y}$. When we perform inference, we often want a point estimate rather than a distribution, so we return the maximum of the distribution:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmax}}\Big[Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\hat{\boldsymbol{\phi}}])\Big]. \tag{5.5}$$

It is usually possible to find an expression for this in terms of the distribution parameters $\boldsymbol{\theta}$ predicted by the model. For example, in the univariate normal distribution, the maximum occurs at the mean μ.

5.2 Recipe for constructing loss functions

The recipe for constructing loss functions for training data $\{\mathbf{x}_i,\mathbf{y}_i\}$ using the maximum likelihood approach is hence:

1. Choose a suitable probability distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ defined over the domain of the predictions $\mathbf{y}$ with distribution parameters $\boldsymbol{\theta}$.
2. Set the machine learning model $\mathbf{f}[\mathbf{x},\boldsymbol{\phi}]$ to predict one or more of these parameters, so $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x},\boldsymbol{\phi}]$ and $Pr(\mathbf{y}|\boldsymbol{\theta}) = Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\boldsymbol{\phi}])$.
3. To train the model, find the network parameters $\hat{\boldsymbol{\phi}}$ that minimize the negative log-likelihood loss function over the training dataset pairs $\{\mathbf{x}_i,\mathbf{y}_i\}$:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L[\boldsymbol{\phi}]\Big] = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\Big]\right]. \tag{5.6}$$

4. To perform inference for a new test example $\mathbf{x}$, return either the full distribution $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\hat{\boldsymbol{\phi}}])$ or the maximum of this distribution.

We devote most of the rest of this chapter to constructing loss functions for common prediction types using this recipe.

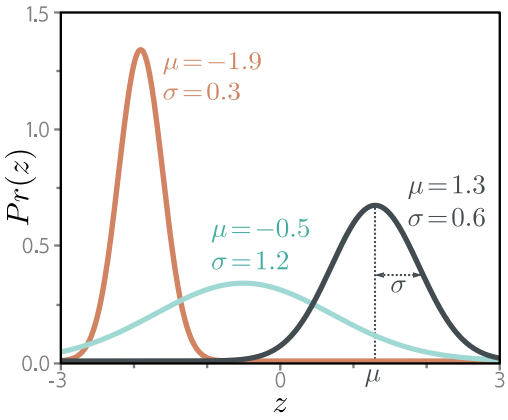


Figure 5.3 The univariate normal distribution (also known as the Gaussian distribution) is defined on the real line $z \in \mathbb{R}$ and has parameters μ and σ^2. The mean μ determines the position of the peak. The positive root of the variance σ^2 (the standard deviation) determines the width of the distribution. Since the total probability density sums to one, the peak becomes higher as the variance decreases and the distribution becomes narrower.

5.3 Example 1: univariate regression

We start by considering univariate regression models. Here the goal is to predict a single scalar output $y \in \mathbb{R}$ from input $\mathbf{x}$ using a model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$. Following the recipe, we choose a probability distribution over the output domain y. We select the univariate normal (figure 5.3), which is defined over $y \in \mathbb{R}$. This distribution has two parameters (mean μ and variance σ^2) and has a probability density function:

$$Pr(y|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]. \tag{5.7}$$

Second, we set the machine learning model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ to compute one or more of the parameters of this distribution. Here, we just compute the mean so $\mu = \mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$:

$$Pr(y|\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}], \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}])^2}{2\sigma^2}\right]. \tag{5.8}$$

We aim to find the parameters $\boldsymbol{\phi}$ that make the training data $\{\mathbf{x}_i, y_i\}$ most probable under this distribution (figure 5.4). To accomplish this, we choose a loss function $L[\boldsymbol{\phi}]$ based on the negative log-likelihood:

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\sum_{i=1}^{I} \log\left[Pr(y_i|\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}], \sigma^2)\right] \\ &= -\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_i-\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]. \end{aligned} \tag{5.9}$$

When we train the model, we seek parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

5.3.1 Least squares loss function

Now let's perform some algebraic manipulations on the loss function. We seek:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]\right]\\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\left(\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right]-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right)\right]\\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2\right], \end{aligned} \tag{5.10}$$

where we have removed the first term between the second and third lines because it does not depend on $\boldsymbol{\phi}$. We have removed the denominator between the third and fourth lines, as this is just a constant scaling factor that does not affect the position of the minimum.

The result of these manipulations is the least squares loss function that we originally introduced when we discussed linear regression in chapter 2:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I}\big(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]\big)^2. \tag{5.11}$$

Notebook 5.1 Least squares loss

We see that the least squares loss function follows naturally from the assumptions that the prediction errors are (i) independent and (ii) drawn from a normal distribution with mean $\mu = \mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]$ (figure 5.4).

5.3.2 Inference

The network no longer directly predicts y but instead predicts the mean $\mu = \mathrm{f}[\mathbf{x},\boldsymbol{\phi}]$ of the normal distribution over y. When we perform inference, we usually want a single "best" point estimate $\hat{y}$, so we take the maximum of the predicted distribution:

$$\hat{y} = \underset{y}{\operatorname{argmax}}\left[Pr(y|\mathrm{f}[\mathbf{x},\hat{\boldsymbol{\phi}},\sigma^2])\right]. \tag{5.12}$$

For the univariate normal, the maximum position is determined by the mean parameter μ (figure 5.3). This is precisely what the model computed, so $\hat{y} = \mathrm{f}[\mathbf{x},\hat{\boldsymbol{\phi}}]$.

5.3.3 Estimating variance

To formulate the least squares loss function, we assumed that the network predicted the mean of a normal distribution. The final expression in equation 5.11 (perhaps surpris-

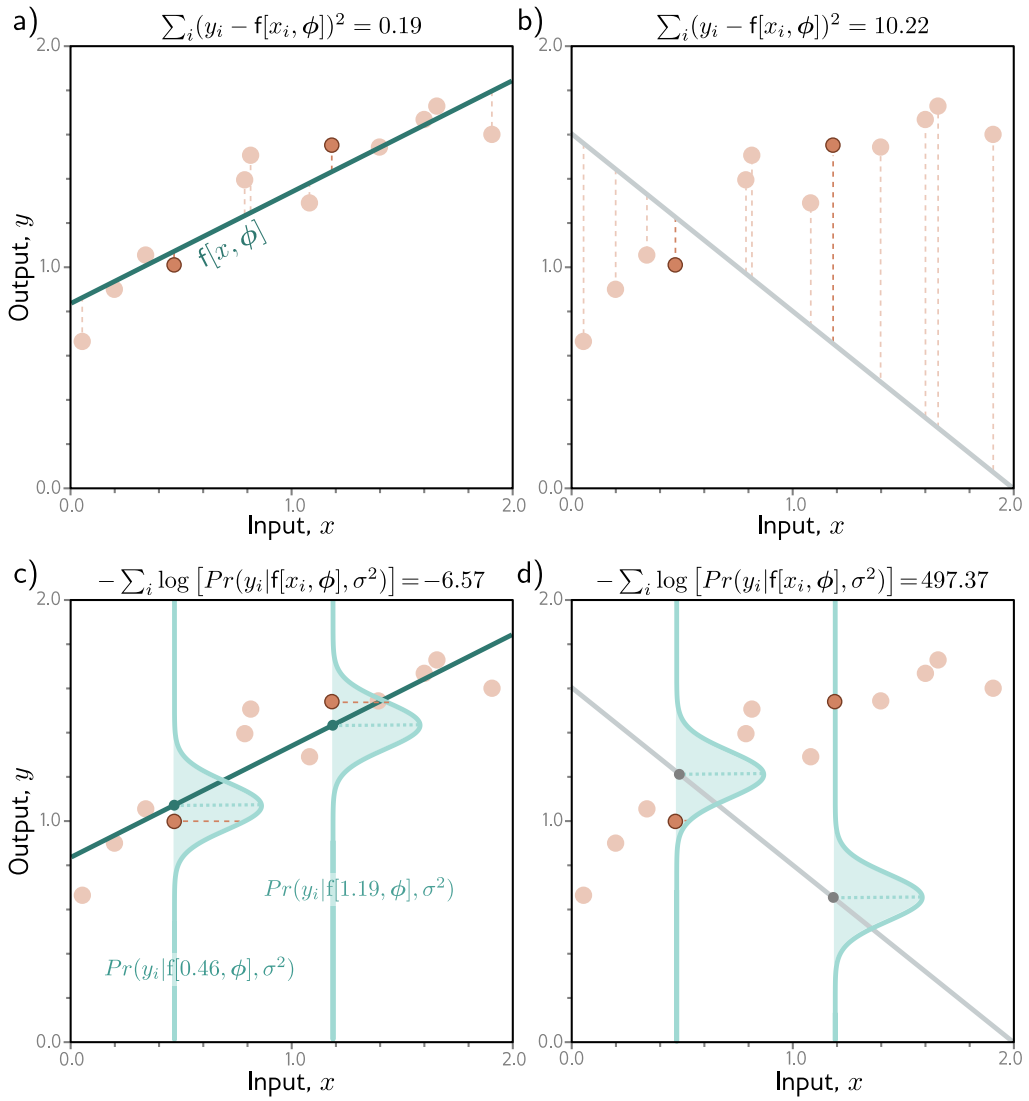


Figure 5.4 Equivalence of least squares and maximum likelihood loss for the normal distribution. a) Consider the linear model from figure 2.2. The least squares criterion minimizes the sum of the squares of the deviations (dashed lines) between the model prediction $\mathrm{f}[x_i, \boldsymbol{\phi}]$ (green line) and the true output values y_i (orange points). Here the fit is good, so these deviations are small (e.g., for the two highlighted points). b) For these parameters, the fit is bad, and the squared deviations are large. c) The least squares criterion follows from the assumption that the model predicts the mean of a normal distribution over the outputs and that we maximize the probability. For the first case, the model fits well, so the probability $Pr(y_i|x_i)$ of the data (horizontal orange dashed lines) is large (and the negative log probability is small). d) For the second case, the model fits badly, so the probability is small and the negative log probability is large.

ingly) does not depend on the variance σ^2. However, there is nothing to stop us from treating σ^2 as a parameter of the model and minimizing equation 5.9 with respect to both the model parameters $\boldsymbol{\phi}$ and the distribution variance σ^2:

$$\hat{\boldsymbol{\phi}}, \hat{\sigma}^2 = \underset{\boldsymbol{\phi},\sigma^2}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[-\frac{(y_i - \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\sigma^2} \right] \right] \right]. \tag{5.13}$$

In inference, the model predicts the mean $\mu = \mathrm{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$ from the input, and we learned the variance $\hat{\sigma}^2$ during the training process. The former is the best prediction. The latter tells us about the uncertainty of the prediction.

5.3.4 Heteroscedastic regression

The model above assumes that the variance of the data is constant everywhere. However, this might be unrealistic. When the uncertainty of the model varies as a function of the input data, we refer to this as *heteroscedastic* (as opposed to *homoscedastic*, where the uncertainty is constant).

A simple way to model this is to train a neural network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that computes both the mean and the variance. For example, consider a shallow network with two outputs. We denote the first output as $\mathrm{f}_1[\mathbf{x}, \boldsymbol{\phi}]$ and use this to predict the mean, and we denote the second output as $\mathrm{f}_2[\mathbf{x}, \boldsymbol{\phi}]$ and use it to predict the variance.

There is one complication; the variance must be positive, but we can't guarantee that the network will always produce a positive output. To ensure that the computed variance is positive, we pass the second network output through a function that maps an arbitrary value to a positive one. A suitable choice is the squaring function, giving:

$$\begin{aligned} \mu &= \mathrm{f}_1[\mathbf{x}, \boldsymbol{\phi}] \\ \sigma^2 &= \mathrm{f}_2[\mathbf{x}, \boldsymbol{\phi}]^2, \end{aligned} \tag{5.14}$$

which results in the loss function:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \left(\log \left[\frac{1}{\sqrt{2\pi \mathrm{f}_2[\mathbf{x}_i, \boldsymbol{\phi}]^2}} \right] - \frac{(y_i - \mathrm{f}_1[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\mathrm{f}_2[\mathbf{x}_i, \boldsymbol{\phi}]^2} \right) \right]. \tag{5.15}$$

Homoscedastic and heteroscedastic models are compared in figure 5.5.

5.4 Example 2: binary classification

In *binary classification*, the goal is to assign the data $\mathbf{x}$ to one of two discrete classes $y \in \{0, 1\}$. In this context, we refer to y as a *label*. Examples of binary classification include (i) predicting whether a restaurant review is positive ($y = 1$) or negative ($y = 0$) from text data $\mathbf{x}$ and (ii) predicting whether a tumor is present ($y = 1$) or absent ($y = 0$) from an MRI scan $\mathbf{x}$.

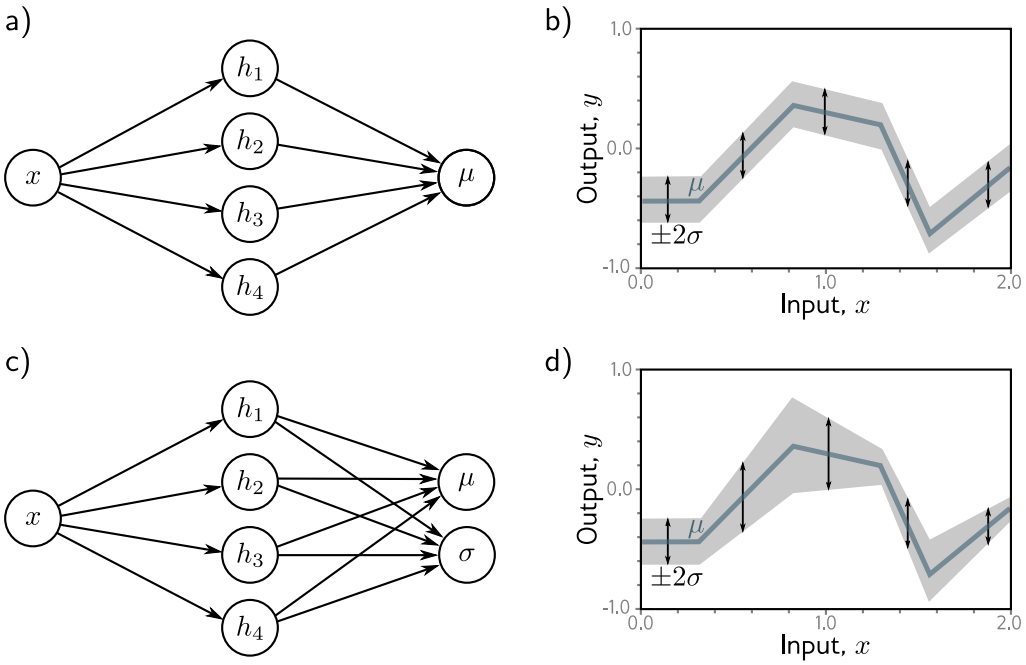


Figure 5.5 Homoscedastic vs. heteroscedastic regression. a) A shallow neural network for homoscedastic regression predicts just the mean μ of the output distribution from the input x. b) The result is that while the mean (blue line) is a piecewise linear function of the input x, the variance is constant everywhere (arrows and gray region show ± 2 standard deviations). c) A shallow neural network for heteroscedastic regression also predicts the variance σ^2 (or, more precisely, computes its square root, which we then square). d) The standard deviation now also becomes a piecewise linear function of the input x.

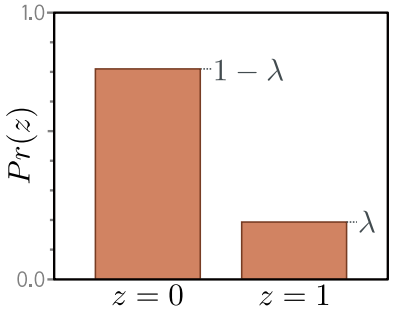


Figure 5.6 Bernoulli distribution. The Bernoulli distribution is defined on the domain $z \in \{0, 1\}$ and has a single parameter λ that denotes the probability of observing $z = 1$. It follows that the probability of observing $z = 0$ is $1 - \lambda$.

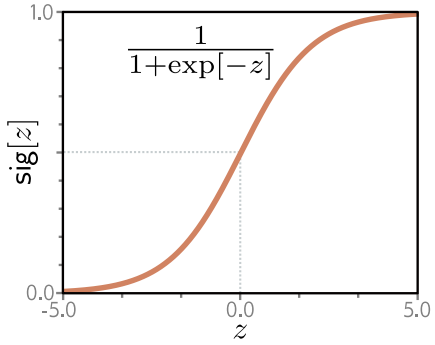


Figure 5.7 Logistic sigmoid function. This function maps the real line $z \in \mathbb{R}$ to numbers between zero and one, so $\text{sig}[z] \in [0, 1]$. An input of 0 is mapped to 0.5. Negative inputs are mapped to numbers below 0.5, and positive inputs to numbers above 0.5.

Once again, we follow the recipe from section 5.2 to construct the loss function. First, we choose a probability distribution over the output space $y \in \{0, 1\}$. A suitable choice is the Bernoulli distribution, which is defined on the domain $\{0, 1\}$. This has a single parameter $\lambda \in [0, 1]$ that represents the probability that y takes the value one (figure 5.6):

$$Pr(y|\lambda) = \begin{cases} 1-\lambda & y = 0 \\ \lambda & y = 1 \end{cases}, \tag{5.16}$$

which can equivalently be written as:

$$Pr(y|\lambda) = (1-\lambda)^{1-y} \cdot \lambda^{y}. \tag{5.17}$$

Second, we set the machine learning model $\text{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict the single distribution parameter λ. However, λ can only take values in the range $[0, 1]$, and we cannot guarantee that the network output will lie in this range. Consequently, we pass the network output through a function that maps the real numbers $\mathbb{R}$ to $[0, 1]$. A suitable function is the *logistic sigmoid* (figure 5.7):

Problem 5.1

$$\text{sig}[z] = \frac{1}{1+\exp[-z]}. \tag{5.18}$$

Hence, we predict the distribution parameter as $\lambda = \text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]$. The likelihood is now:

$$Pr(y|\mathbf{x}) = (1 - \text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]])^{1-y} \cdot \text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]^{y}. \tag{5.19}$$

This is depicted in figure 5.8 for a shallow neural network model. The loss function is the negative log-likelihood of the training set:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} -(1-y_i)\log\Big[1 - \text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big] - y_i \log\Big[\text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big]. \tag{5.20}$$

For reasons to be explained in section 5.7, this is known as the *binary cross-entropy loss*.

Notebook 5.2 Binary cross-entropy loss

Problem 5.2

The transformed model output $\text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]$ predicts the parameter λ of the Bernoulli distribution. This represents the probability that $y = 1$, and it follows that $1 - \lambda$ represents the probability that $y = 0$. When we perform inference, we may want a point estimate of y, so we set $y = 1$ if $\lambda > 0.5$ and $y = 0$ otherwise.

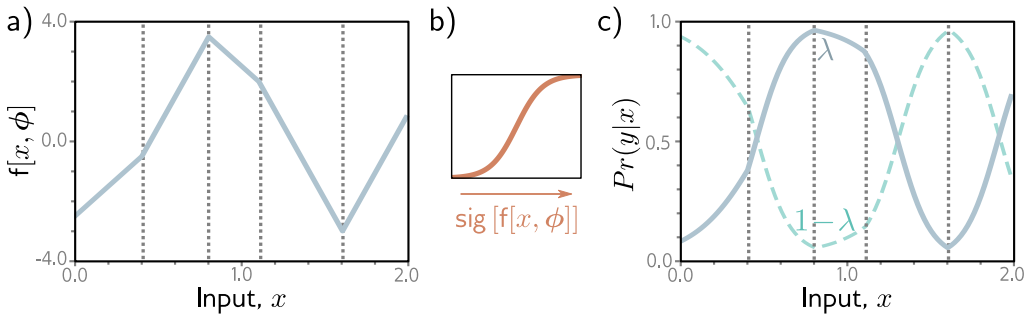


Figure 5.8 Binary classification model. a) The network output is a piecewise linear function that can take arbitrary real values. b) This is transformed by the logistic sigmoid function, which compresses these values to the range $[0, 1]$. c) The transformed output predicts the probability λ that $y = 1$ (solid line). The probability that $y = 0$ is hence $1 - \lambda$ (dashed line). For any fixed x (vertical slice), we retrieve the two values of a Bernoulli distribution similar to that in figure 5.6. The loss function favors model parameters that produce large values of λ at positions x_i that are associated with positive examples $y_i = 1$ and small values of λ at positions associated with negative examples $y_i = 0$.

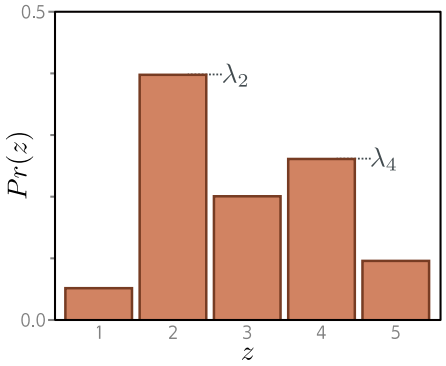


Figure 5.9 Categorical distribution. The categorical distribution assigns probabilities to $K > 2$ categories, with associated probabilities $\lambda_1, \lambda_2, \ldots, \lambda_K$. Here, there are five categories, so $K = 5$. To ensure that this is a valid probability distribution, each parameter λ_k must lie in the range $[0, 1]$, and all K parameters must sum to one.

5.5 Example 3: multiclass classification

The goal of *multiclass classification* is to assign an input data example $\mathbf{x}$ to one of $K > 2$ classes, so $y \in \{1, 2, \ldots, K\}$. Real-world examples include (i) predicting which of $K = 10$ digits y is present in an image $\mathbf{x}$ of a handwritten number and (ii) predicting which of K possible words y follows an incomplete sentence $\mathbf{x}$.

We once more follow the recipe from section 5.2. We first choose a distribution over the prediction space y. In this case, we have $y \in \{1, 2, \ldots, K\}$, so we choose the *categorical distribution* (figure 5.9), which is defined on this domain. This has K parameters $\lambda_1, \lambda_2, \ldots, \lambda_K$, which determine the probability of each category:

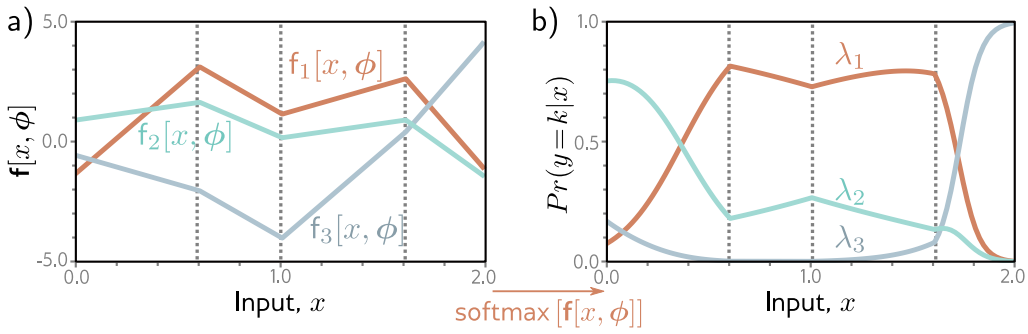


Figure 5.10 Multiclass classification for $K=3$ classes. a) The network has three piecewise linear outputs, which can take arbitrary values. b) After the softmax function, these outputs are constrained to be non-negative and sum to one. Hence, for a given input $\mathbf{x}$, we compute valid parameters for the categorical distribution: any vertical slice of this plot produces three values sum to one and would form the heights of the bars in a categorical distribution similar to figure 5.9.

$$Pr(y=k) = \lambda_k. \tag{5.21}$$

The parameters are constrained to take values between zero and one, and they must collectively sum to one to ensure a valid probability distribution.

Then we use a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with K outputs to compute these K parameters from the input $\mathbf{x}$. Unfortunately, the network outputs will not necessarily obey the aforementioned constraints. Consequently, we pass the K outputs of the network through a function that ensures these constraints are respected. A suitable choice is the *softmax* function (figure 5.10). This takes an arbitrary vector of length K and returns a vector of the same length but where the elements are now in the range $[0, 1]$ and sum to one. The k^{th} output of the softmax function is:

$$\text{softmax}_k[\mathbf{z}] = \frac{\exp[z_k]}{\sum_{k'=1}^{K} \exp[z_{k'}]}, \tag{5.22}$$

Appendix B.1.3 Exponential function

where the exponential functions ensure positivity, and the sum in the denominator ensures that the K numbers sum to one.

The likelihood that input $\mathbf{x}$ has label $y = k$ (figure 5.10) is hence:

$$Pr(y=k|\mathbf{x}) = \text{softmax}_k\Big[\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]\Big]. \tag{5.23}$$

The loss function is the negative log-likelihood of the training data:

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\sum_{i=1}^{I} \log\Big[\text{softmax}_{y_i}\Big[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]\Big]\Big] \\ &= -\sum_{i=1}^{I}\left(\text{f}_{y_i}[\mathbf{x}_i, \boldsymbol{\phi}] - \log\left[\sum_{k'=1}^{K}\exp[\,\text{f}_{k'}[\mathbf{x}_i, \boldsymbol{\phi}]]\right]\right), \end{aligned} \tag{5.24}$$

where $\text{f}_k[\mathbf{x}, \boldsymbol{\phi}]$ denotes the k^{th} output of the neural network. For reasons that will be explained in section 5.7, this is known as the *multiclass cross-entropy loss*.

The transformed model output represents a categorical distribution over possible classes $y \in \{1, 2, \ldots, K\}$. For a point estimate, we take the most probable category $\hat{y} = \text{argmax}_k[Pr(y = k|\mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}])]$. This corresponds to whichever curve is highest for that value of $\mathbf{x}$ in figure 5.10.

Notebook 5.3 Multiclass cross-entropy loss

5.5.1 Predicting other data types

In this chapter, we have focused on regression and classification because these problems are widespread. However, to make different types of predictions, we simply choose an appropriate distribution over that domain and apply the recipe in section 5.2. Figure 5.11 enumerates a series of probability distributions and their prediction domains. Some of these are explored in the problems at the end of the chapter.

Problems 5.3–5.6

5.6 Multiple outputs

Often, we wish to make more than one prediction with the same model, so the target output $\mathbf{y}$ is a vector. For example, we might want to predict a molecule's melting and boiling point (a multivariate regression problem, figure 1.2b) or the object class at every point in an image (a multivariate classification problem, figure 1.4a). While it is possible to define multivariate probability distributions and use a neural network to model their parameters as a function of the input, it is more usual to treat each prediction as *independent*.

Independence implies that we treat the probability $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}])$ as a product of univariate terms for each element $y_d \in \mathbf{y}$:

Appendix C.1.5 Independence

$$Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]) = \prod_d Pr(y_d|\mathbf{f}_d[\mathbf{x}, \boldsymbol{\phi}]), \tag{5.25}$$

where $\mathbf{f}_d[\mathbf{x}, \boldsymbol{\phi}]$ is the d^{th} set of network outputs, which describe the parameters of the distribution over y_d. For example, to predict multiple continuous variables $y_d \in \mathbb{R}$, we use a normal distribution for each y_d, and the network outputs $\text{f}_d[\mathbf{x}, \boldsymbol{\phi}]$ predict the means of these distributions. To predict multiple discrete variables $y_d \in \{1, 2, \ldots, K\}$, we use a categorical distribution for each y_d. Here, each set of network outputs $\mathbf{f}_d[\mathbf{x}, \boldsymbol{\phi}]$ predicts the K values that contribute to the categorical distribution for y_d.

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots, K]$	Plackett-Luce	ranking

Figure 5.11 Distributions for loss functions for different prediction types.

When we minimize the negative log probability, this product becomes a sum of terms:

$$L[\boldsymbol{\phi}] = -\sum_{i=1}^{I} \log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\Big] = -\sum_{i=1}^{I}\sum_{d} \log\Big[Pr(y_{id}|\mathbf{f}_d[\mathbf{x}_i, \boldsymbol{\phi}])\Big]. \tag{5.26}$$

where y_{id} is the d^{th} output from the i^{th} training example.

Problems 5.7–5.10

To make two or more prediction types simultaneously, we similarly assume the errors in each are independent. For example, to predict wind direction and strength, we might choose the von Mises distribution (defined on circular domains) for the direction and the exponential distribution (defined on positive real numbers) for the strength. The independence assumption implies that the joint likelihood of the two predictions is the product of individual likelihoods. These terms will become additive when we compute the negative log-likelihood.

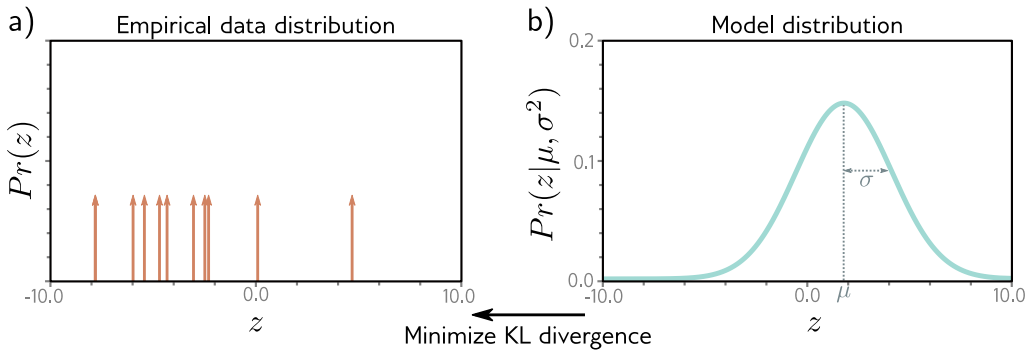


Figure 5.12 Cross-entropy method. a) Empirical distribution of training samples (arrows denote Dirac delta functions). b) Model distribution (a normal distribution with parameters $\boldsymbol{\theta} = \mu, \sigma^2$). In the cross-entropy approach, we minimize the distance (KL divergence) between these two distributions as a function of the model parameters $\boldsymbol{\theta}$.

5.7 Cross-entropy loss

In this chapter, we developed loss functions that minimize negative log-likelihood. However, the term *cross-entropy* loss is also commonplace. In this section, we describe the cross-entropy loss and show that it is equivalent to using negative log-likelihood.

The cross-entropy loss is based on the idea of finding parameters $\boldsymbol{\theta}$ that minimize the distance between the empirical distribution $q(y)$ of the observed data y and a model distribution $Pr(y|\boldsymbol{\theta})$ (figure 5.12). The distance between two probability distributions $q(z)$ and $p(z)$ can be evaluated using the Kullback-Leibler (KL) divergence:

Appendix C.5.1 KL Divergence

$$D_{KL}\big[q||p\big] = \int_{-\infty}^{\infty} q(z)\log\big[q(z)\big]dz - \int_{-\infty}^{\infty} q(z)\log\big[p(z)\big]dz. \tag{5.27}$$

Now consider that we observe an empirical data distribution at points $\{y_i\}_{i=1}^{I}$. We can describe this as a weighted sum of point masses:

$$q(y) = \frac{1}{I}\sum_{i=1}^{I}\delta[y - y_i], \tag{5.28}$$

where $\delta[\bullet]$ is the Dirac delta function. We want to minimize the KL divergence between the model distribution $Pr(y|\boldsymbol{\theta})$ and this empirical distribution:

Appendix B.1.3 Dirac delta function

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[\int_{-\infty}^{\infty} q(y)\log\big[q(y)\big]dy - \int_{-\infty}^{\infty} q(y)\log\big[Pr(y|\boldsymbol{\theta})\big]dy\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty} q(y)\log\big[Pr(y|\boldsymbol{\theta})\big]dy\right],\end{aligned} \tag{5.29}$$

where the first term disappears, as it has no dependence on $\boldsymbol{\theta}$. The remaining second term is known as the *cross-entropy*. It can be interpreted as the amount of uncertainty that remains in one distribution after taking into account what we already know from the other. Now, we substitute in the definition of $q(y)$ from equation 5.28:

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\theta}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty}\left(\frac{1}{I}\sum_{i=1}^{I}\delta[y-y_i]\right)\log\big[Pr(y|\boldsymbol{\theta})\big]dy\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\frac{1}{I}\sum_{i=1}^{I}\log\big[Pr(y_i|\boldsymbol{\theta})\big]\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\big[Pr(y_i|\boldsymbol{\theta})\big]\right]. \qquad (5.30)\end{aligned}$$

The product of the two terms in the first line corresponds to pointwise multiplying the point masses in figure 5.12a with the logarithm of the distribution in figure 5.12b. We are left with a finite set of weighted probability masses centered on the data points. In the last line, we have eliminated the constant scaling factor $1/I$, as this does not affect the position of the minimum.

In machine learning, the distribution parameters $\boldsymbol{\theta}$ are computed by the model $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$, so we have:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\big[Pr(y_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\big]\right]. \qquad (5.31)$$

This is precisely the negative log-likelihood criterion from the recipe in section 5.2.

It follows that the negative log-likelihood criterion (from maximizing the data likelihood) and the cross-entropy criterion (from minimizing the distance between the model and empirical data distributions) are equivalent.

5.8 Summary

We previously considered neural networks as directly predicting outputs $\mathbf{y}$ from data $\mathbf{x}$. In this chapter, we shifted perspective to think about neural networks as computing the parameters $\boldsymbol{\theta}$ of probability distributions $Pr(\mathbf{y}|\boldsymbol{\theta})$ over the output space. This led to a principled approach to building loss functions. We selected model parameters $\boldsymbol{\phi}$ that maximized the likelihood of the observed data under these distributions. We saw that this is equivalent to minimizing the negative log-likelihood.

The least squares criterion for regression is a natural consequence of this approach; it follows from the assumption that y is normally distributed and that we are predicting the mean. We also saw how the regression model could be (i) extended to estimate the uncertainty over the prediction and (ii) extended to make that uncertainty dependent on the input (the heteroscedastic model). We applied the same approach to both binary and multiclass classification and derived loss functions for each. We discussed how to

tackle more complex data types and how to deal with multiple outputs. Finally, we argued that cross-entropy is an equivalent way to think about fitting models.

In previous chapters, we developed neural network models. In this chapter, we developed loss functions for deciding how well a model describes the training data for a given set of parameters. The next chapter considers model training, in which we aim to find the model parameters that minimize this loss.

Notes

Losses based on the normal distribution: Nix & Weigend (1994) and Williams (1996) investigated heteroscedastic nonlinear regression in which both the mean and the variance of the output are functions of the input. In the context of unsupervised learning, Burda et al. (2016) use a loss function based on a multivariate normal distribution with diagonal covariance, and Dorta et al. (2018) use a loss function based on a normal distribution with full covariance.

Robust regression: Qi et al. (2020) investigate the properties of regression models that minimize mean absolute error rather than mean squared error. This loss function follows from assuming a Laplace distribution over the outputs and estimates the median output for a given input rather than the mean. Barron (2019) presents a loss function that parameterizes the degree of robustness. When interpreted in a probabilistic context, it yields a family of univariate probability distributions that includes the normal and Cauchy distributions as special cases.

Estimating quantiles: Sometimes, we may not want to estimate the mean or median in a regression task but may instead want to predict a quantile. For example, this is useful for risk models, where we want to know that the true value will be less than the predicted value 90% of the time. This is known as *quantile regression* (Koenker & Hallock, 2001). This could be done by fitting a heteroscedastic regression model and then estimating the quantile based on the predicted normal distribution. Alternatively, the quantiles can be estimated directly using *quantile loss* (also known as *pinball loss*). In practice, this minimizes the absolute deviations of the data from the model but weights the deviations in one direction more than the other. Recent work has investigated simultaneously predicting multiple quantiles to get an idea of the overall distribution shape (Rodrigues & Pereira, 2020).

Class imbalance and focal loss: Lin et al. (2017c) address data imbalance in classification problems. If the number of examples for some classes is much greater than for others, then the standard maximum likelihood loss does not work well; the model may concentrate on becoming more confident about well-classified examples from the dominant classes and classify less well-represented classes poorly. Lin et al. (2017c) introduce *focal loss*, which adds a single extra parameter that down-weights the effect of well-classified examples to improve performance.

Learning to rank: Cao et al. (2007), Xia et al. (2008), and Chen et al. (2009) all used the Plackett-Luce model in loss functions for learning to rank data. This is the *listwise* approach to learning to rank as the model ingests an entire list of objects to be ranked at once. Alternative approaches are the *pointwise* approach, in which the model ingests a single object, and the *pairwise* approach, where the model ingests pairs of objects. Chen et al. (2009) summarize different approaches for learning to rank.

Other data types: Fan et al. (2020) use a loss based on the beta distribution for predicting values between zero and one. Jacobs et al. (1991) and Bishop (1994) investigated *mixture density networks* for multimodal data. These model the output as a mixture of Gaussians

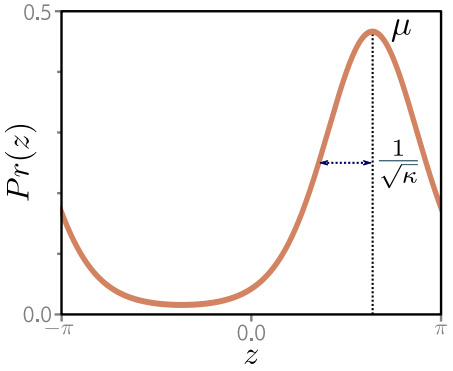


Figure 5.13 The von Mises distribution is defined over the circular domain $(-\pi, \pi]$. It has two parameters. The mean μ determines the position of the peak. The concentration $\kappa > 0$ acts like the inverse of the variance. Hence $1/\sqrt{\kappa}$ is roughly equivalent to the standard deviation in a normal distribution.

(see figure 5.14) that is conditional on the input. Prokudin et al. (2018) used the von Mises distribution to predict direction (see figure 5.13). Fallah et al. (2009) constructed loss functions for prediction counts using the Poisson distribution (see figure 5.15). Ng et al. (2017) used loss functions based on the gamma distribution to predict duration.

Non-probabilistic approaches: It is not strictly necessary to adopt the probabilistic approach discussed in this chapter, but this has become the default in recent years; any loss function that aims to reduce the distance between the model output and the training outputs will suffice, and distance can be defined in any way that seems sensible. There are several well-known non-probabilistic machine learning models for classification, including support vector machines (Vapnik, 1995; Cristianini & Shawe-Taylor, 2000), which use *hinge loss*, and AdaBoost (Freund & Schapire, 1997), which uses *exponential loss*.

Problems

Problem 5.1 Show that the logistic sigmoid function $\text{sig}[z]$ becomes 0 as $z \to -\infty$, is 0.5 when $z = 0$, and becomes 1 when $z \to \infty$, where:

$$\text{sig}[z] = \frac{1}{1+\exp[-z]}. \tag{5.32}$$

Problem 5.2 The loss L for binary classification for a single training pair $\{\mathbf{x}, y\}$ is:

$$L = -(1-y)\log\Big[1-\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\Big] - y\log\Big[\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\Big], \tag{5.33}$$

where $\text{sig}[\bullet]$ is defined in equation 5.32. Plot this loss as a function of the transformed network output $\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]] \in [0,1]$ (i) when the training label $y = 0$ and (ii) when $y = 1$.

Problem 5.3* Suppose we want to build a model that predicts the direction y in radians of the prevailing wind based on local measurements of barometric pressure $\mathbf{x}$. A suitable distribution over circular domains is the von Mises distribution (figure 5.13):

$$Pr(y|\mu,\kappa) = \frac{\exp\big[\kappa\cos[y-\mu]\big]}{2\pi\cdot\text{Bessel}_0[\kappa]}, \tag{5.34}$$

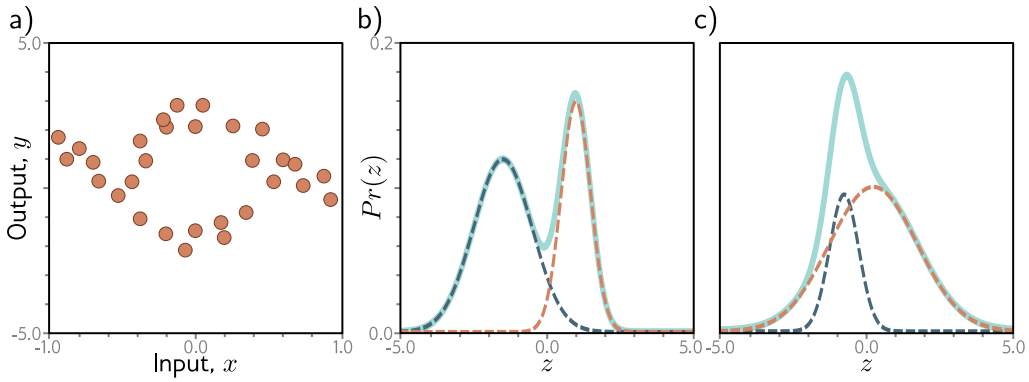


Figure 5.14 Multimodal data and mixture of Gaussians density. a) Example training data where, for intermediate values of the input x, the corresponding output y follows one of two paths. For example, at $x = 0$, the output y might be roughly -2 or $+3$ but is unlikely to be between these values. b) The mixture of Gaussians is a probability model suited to this kind of data. As the name suggests, the model is a weighted sum (solid cyan curve) of two or more normal distributions with different means and variances (here, two weighted distributions, dashed blue and orange curves). When the means are far apart, this forms a multimodal distribution. c) When the means are close, the mixture can model unimodal but non-normal densities.

where μ is a measure of the mean direction and κ is a measure of concentration (i.e., the inverse of the variance). The term $\text{Bessel}_0[\kappa]$ is a modified Bessel function of the first kind of order 0. Use the recipe from section 5.2 to develop a loss function for learning the parameter μ of a model $\text{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict the most likely wind direction. Your solution should treat the concentration κ as constant. How would you perform inference?

Problem 5.4* Sometimes, the outputs y for input $\mathbf{x}$ are multimodal (figure 5.14a); there is more than one valid prediction for a given input. Here, we might use a weighted sum of normal components as the distribution over the output. This is known as a *mixture of Gaussians* model. For example, a mixture of two Gaussians has parameters $\boldsymbol{\theta} = \{\lambda, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2\}$:

$$Pr(y|\lambda, \mu_1, \mu_2, \sigma_1^2, \sigma_2^2) = \frac{\lambda}{\sqrt{2\pi\sigma_1^2}} \exp\left[\frac{-(y-\mu_1)^2}{2\sigma_1^2}\right] + \frac{1-\lambda}{\sqrt{2\pi\sigma_2^2}} \exp\left[\frac{-(y-\mu_2)^2}{2\sigma_2^2}\right], \tag{5.35}$$

where $\lambda \in [0, 1]$ controls the relative weight of the two components, which have means μ_1, μ_2 and variances σ_1^2, σ_2^2, respectively. This model can represent a distribution with two peaks (figure 5.14b) or a distribution with one peak but a more complex shape (figure 5.14c).
Use the recipe from section 5.2 to construct a loss function for training a model $\mathbf{f}[x, \boldsymbol{\phi}]$ that takes input x, has parameters $\boldsymbol{\phi}$, and predicts a mixture of two Gaussians. The loss should be based on I training data pairs $\{x_i, y_i\}$. What problems do you foresee when performing inference?

Problem 5.5 Consider extending the model from problem 5.3 to predict the wind direction using a mixture of two von Mises distributions. Write an expression for the likelihood $Pr(y|\boldsymbol{\theta})$ for this model. How many outputs will the network need to produce?

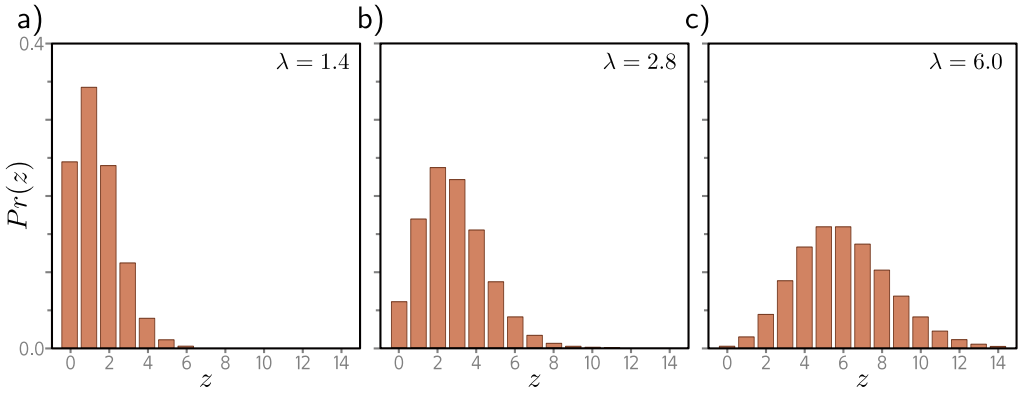


Figure 5.15 Poisson distribution. This discrete distribution is defined over non-negative integers $z \in \{0, 1, 2, \ldots\}$. It has a single parameter $\lambda \in \mathbb{R}^+$, which is known as the rate and is the mean of the distribution. a–c) Poisson distributions with rates of 1.4, 2.8, and 6.0, respectively.

Problem 5.6 Consider building a model to predict the number of pedestrians $y \in \{0, 1, 2, \ldots\}$ that will pass a given point in the city in the next minute, based on data $\mathbf{x}$ that contains information about the time of day, the longitude and latitude, and the type of neighborhood. A suitable distribution for modeling counts is the Poisson distribution (figure 5.15). This has a single parameter $\lambda > 0$ called the *rate* that represents the mean of the distribution. The distribution has probability density function:

$$Pr(y = k) = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{5.36}$$

Design a loss function for this model assuming we have access to I training pairs $\{\mathbf{x}_i, y_i\}$.

Problem 5.7 Consider a multivariate regression problem where we predict ten outputs, so $\mathbf{y} \in \mathbb{R}^{10}$, and model each with an independent normal distribution where the means μ_d are predicted by the network, and variances σ^2 are constant. Write an expression for the likelihood $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}])$. Show that minimizing the negative log-likelihood of this model is still equivalent to minimizing a sum of squared terms if we don't estimate the variance σ^2.

Problem 5.8* Construct a loss function for making multivariate predictions $\mathbf{y} \in \mathbb{R}^{D_o}$ based on independent normal distributions with different variances σ_d^2 for each dimension. Assume a heteroscedastic model so that both the means μ_d and variances σ_d^2 vary as a function of the data.

Problem 5.9* Consider a multivariate regression problem in which we predict the height of a person in meters and their weight in kilos from data $\mathbf{x}$. Here, the units take quite different ranges. What problems do you see this causing? Propose two solutions to these problems.

Problem 5.10 Extend the model from problem 5.3 to predict both the wind direction and the wind speed and define the associated loss function.

Chapter 6

Fitting models

Chapters 3 and 4 described shallow and deep neural networks. These represent families of piecewise linear functions, where the parameters determine the particular function. Chapter 5 introduced the loss — a single number representing the mismatch between the network predictions and the ground truth for a training set.

The loss depends on the network parameters, and this chapter considers how to find the parameter values that minimize this loss. This is known as *learning* the network's parameters or simply as *training* or *fitting* the model. The process is to choose initial parameter values and then iterate the following two steps: (i) compute the derivatives (gradients) of the loss with respect to the parameters, and (ii) adjust the parameters based on the gradients to decrease the loss. After many iterations, we hope to reach the overall minimum of the loss function.

This chapter tackles the second of these steps; we consider algorithms that adjust the parameters to decrease the loss. Chapter 7 discusses how to initialize the parameters and compute the gradients for neural networks.

6.1 Gradient descent

To fit a model, we need a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek parameters $\boldsymbol{\phi}$ for the model $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that map the inputs $\mathbf{x}_i$ to the outputs $\mathbf{y}_i$ as well as possible. To this end, we define a loss function $L[\boldsymbol{\phi}]$ that returns a single number that quantifies the mismatch in this mapping. The goal of an *optimization algorithm* is to find parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L[\boldsymbol{\phi}]\Big]. \tag{6.1}$$

There are many families of optimization algorithms, but the standard methods for training neural networks are iterative. These algorithms initialize the parameters heuristically and then adjust them repeatedly in such a way that the loss decreases.

The simplest method in this class is *gradient descent*. This starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates two steps:

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{6.2}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{6.3}$$

where the positive scalar α determines the magnitude of the change.

The first step computes the gradient of the loss function at the current position. This determines the *uphill* direction of the loss function. The second step moves a small distance α *downhill* (hence the negative sign). The parameter α may be fixed (in which case, we call it a *learning rate*), or we may perform a *line search* where we try several values of α to find the one that most decreases the loss.

Notebook 6.1
Line search

At the minimum of the loss function, the surface must be flat (or we could improve further by going downhill). Hence, the gradient will be zero, and the parameters will stop changing. In practice, we monitor the gradient magnitude and terminate the algorithm when it becomes too small.

6.1.1 Linear regression example

Consider applying gradient descent to the 1D linear regression model from chapter 2. The model $\mathrm{f}[x, \boldsymbol{\phi}]$ maps a scalar input x to a scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, which represent the y-intercept and the slope:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{6.4}$$

Given a dataset $\{x_i, y_i\}$ containing I input/output pairs, we choose the least squares loss function:

$$\begin{aligned} L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2, \end{aligned} \tag{6.5}$$

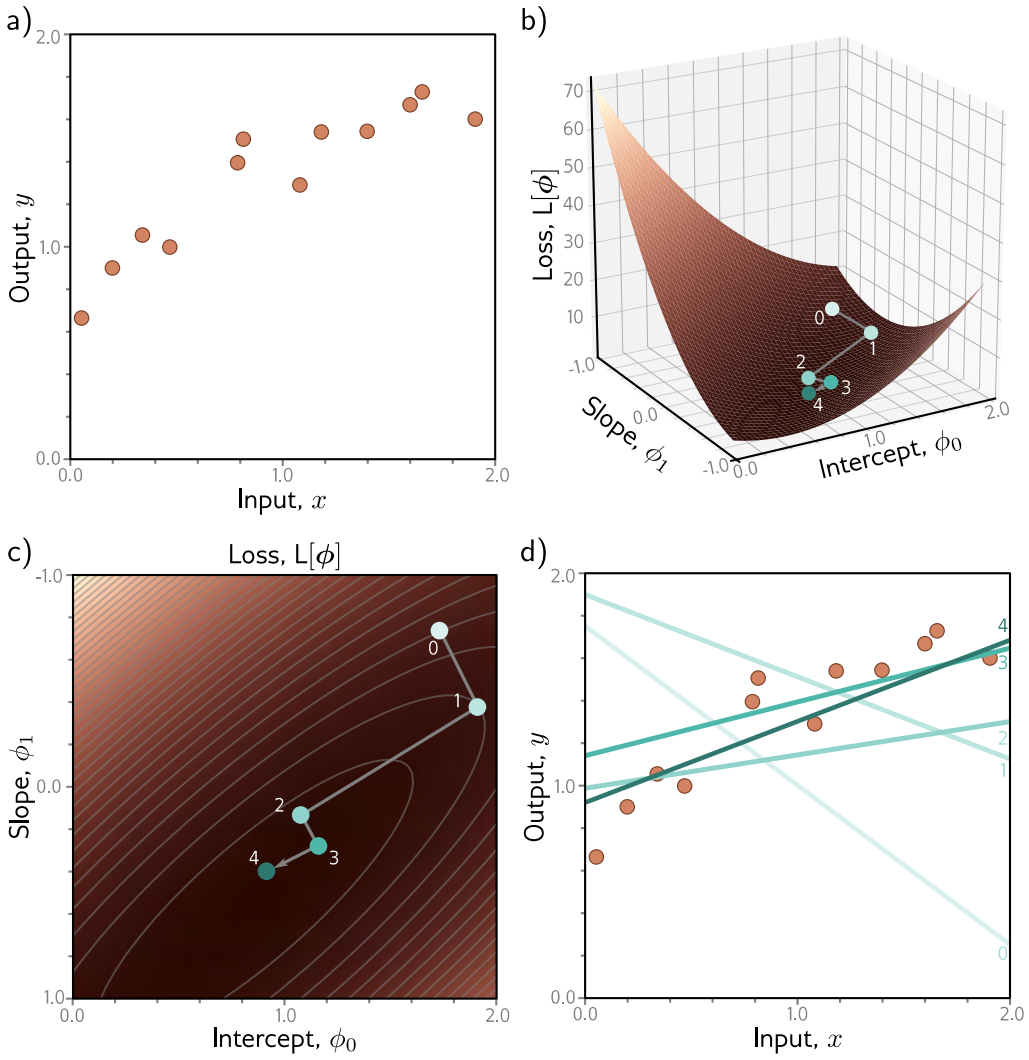


Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

where the term $\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2$ is the individual contribution to the loss from the i^{th} training example.

The derivative of the loss function with respect to the parameters can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} \frac{\partial \ell_i}{\partial \boldsymbol{\phi}}, \tag{6.6}$$

Problem 6.1

where these are given by:

$$\frac{\partial \ell_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial \ell_i}{\partial \phi_0} \\ \frac{\partial \ell_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}. \tag{6.7}$$

Notebook 6.2
Gradient descent

Figure 6.1 shows the progression of this algorithm as we iteratively compute the derivatives according to equations 6.6 and 6.7 and then update the parameters using the rule in equation 6.3. In this case, we have used a line search procedure to find the value of α that decreases the loss the most at each iteration.

6.1.2 Gabor model example

Loss functions for linear regression problems (figure 6.1c) always have a single well-defined global minimum. More formally, they are *convex*, which means that no chord (line segment between two points on the surface) intersects the function. Convexity implies that wherever we initialize the parameters, we are bound to reach the minimum if we keep walking downhill; the training procedure can't fail.

Problem 6.2

Unfortunately, loss functions for most nonlinear models, including both shallow and deep networks, are *non-convex*. Visualizing neural network loss functions is challenging due to the number of parameters. Hence, we first explore a simpler nonlinear model with two parameters to gain insight into the properties of non-convex loss functions:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{32.0}\right). \tag{6.8}$$

Problems 6.3–6.5

This *Gabor model* maps scalar input x to scalar output y and consists of a sinusoidal component (creating an oscillatory function) multiplied by a negative exponential component (causing the amplitude to decrease as we move from the center). It has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where $\phi_0 \in \mathbb{R}$ determines the mean position of the function and $\phi_1 \in \mathbb{R}^+$ stretches or squeezes it along the x-axis (figure 6.2).

Consider a training set of I examples $\{x_i, y_i\}$ (figure 6.3). The least squares loss function for I training examples is defined as:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2. \tag{6.9}$$

Once more, the goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

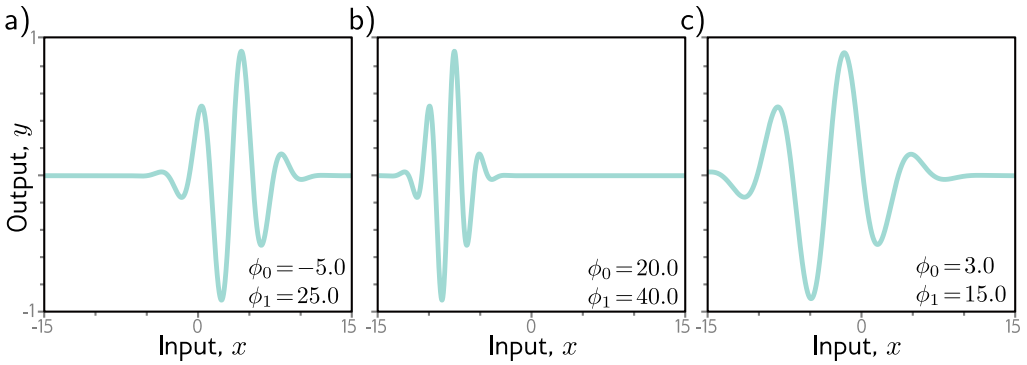


Figure 6.2 Gabor model. This nonlinear model maps scalar input x to scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$. It describes a sinusoidal function that decreases in amplitude with distance from its center. Parameter $\phi_0 \in \mathbb{R}$ determines the position of the center. As ϕ_0 increases, the function moves left. Parameter $\phi_1 \in \mathbb{R}^+$ squeezes the function along the x-axis relative to the center. As ϕ_1 increases, the function narrows. a–c) Model with different parameters.

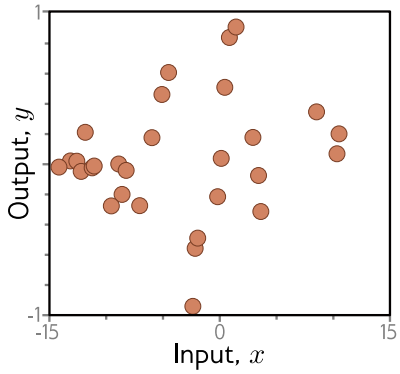


Figure 6.3 Training data for fitting the Gabor model. The training dataset contains 28 input/output examples $\{x_i, y_i\}$. These data were created by uniformly sampling $x_i \in [-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding normally distributed noise.

6.1.3 Local minima and saddle points

Figure 6.4 depicts the loss function associated with the Gabor model for this dataset. There are numerous *local minima* (cyan circles). Here the gradient is zero, and the loss increases if we move in any direction, but we are *not* at the overall minimum of the function. The point with the lowest loss is known as the *global minimum* and is depicted by the gray circle.

Problem 6.6

If we start in a random position and use gradient descent to go downhill, there is no guarantee that we will wind up at the global minimum and find the best parameters (figure 6.5a). It's equally or even more likely that the algorithm will terminate in one of the local minima. Furthermore, there is no way of knowing whether there is a better solution elsewhere.

Problems 6.7–6.8

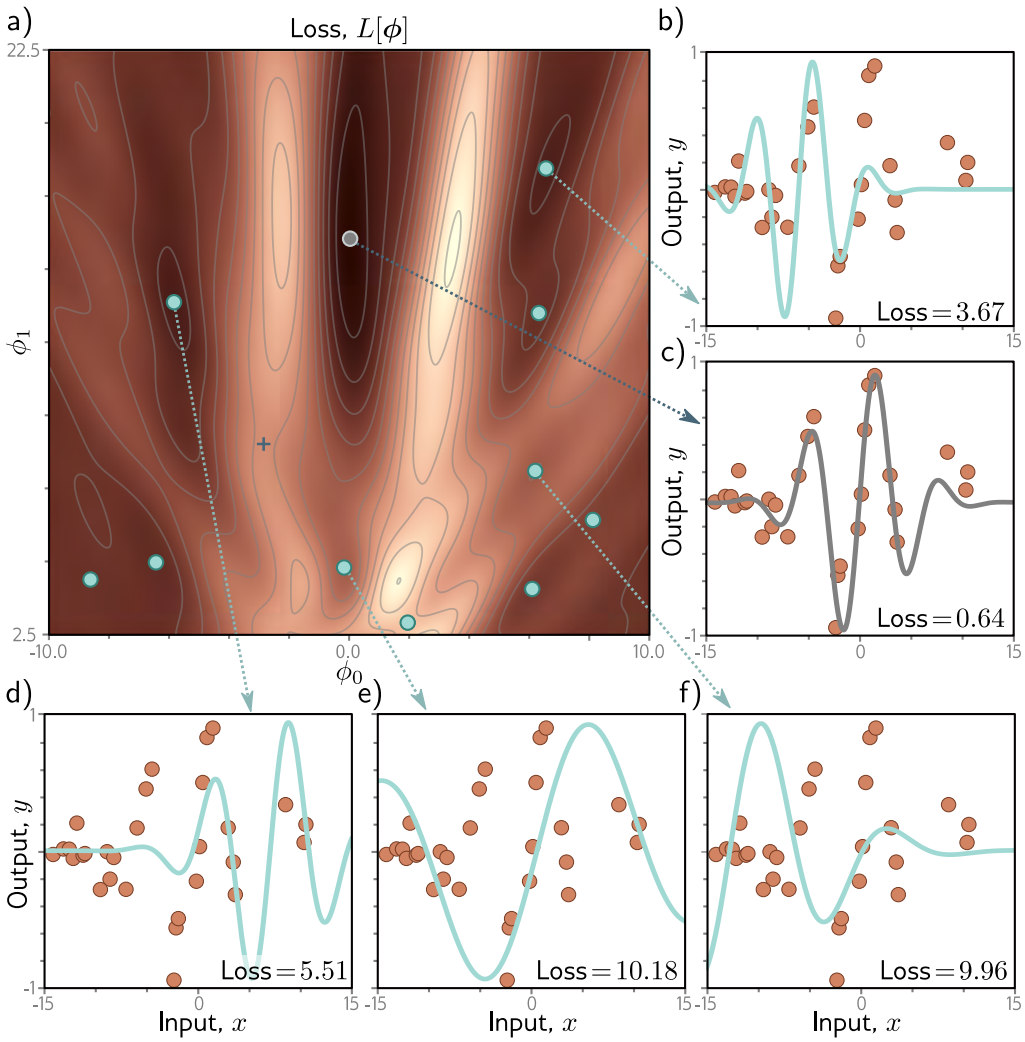


Figure 6.4 Loss function for the Gabor model. a) The loss function is non-convex, with multiple local minima (cyan circles) in addition to the global minimum (gray circle). It also contains saddle points where the gradient is locally zero, but the function increases in one direction and decreases in the other. The blue cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically. b–f) Models associated with the different minima. In each case, there is no small change that decreases the loss. Panel (c) shows the global minimum, which has a loss of 0.64.

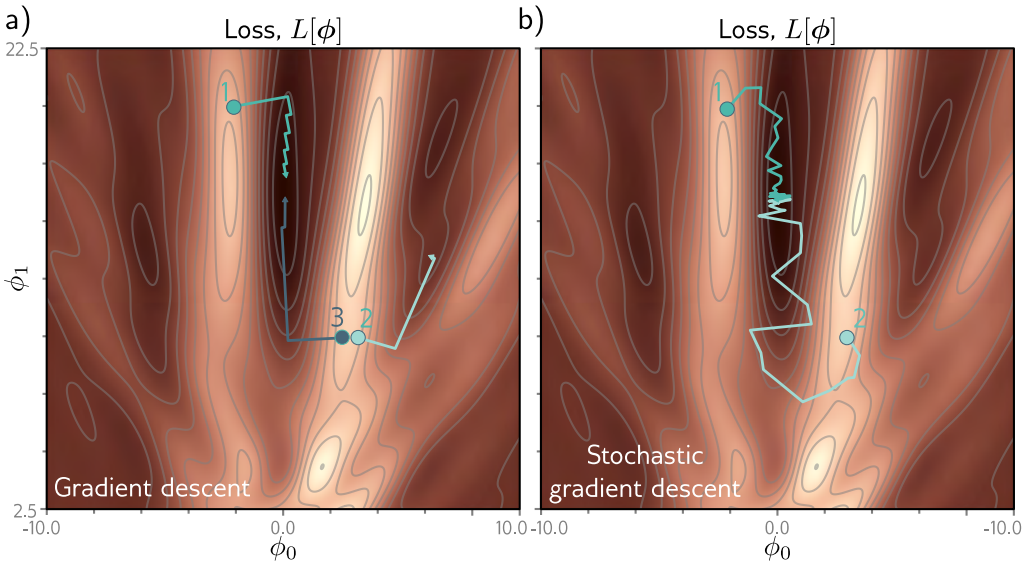


Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent with line search. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized outside this valley (e.g., point 2), it will descend toward one of the local minima. b) Stochastic gradient descent adds noise to the optimization process, so it is possible to escape from the wrong valley (e.g., point 2) and still reach the global minimum.

In addition, the loss function contains *saddle points* (e.g., the blue cross in figure 6.4). Here, the gradient is zero, but the function increases in some directions and decreases in others. If the current parameters are not exactly at the saddle point, then gradient descent can escape by moving downhill. However, the surface near the saddle point is flat, so it's hard to be sure that training hasn't converged; if we terminate the algorithm when the gradient is small, we may erroneously stop near a saddle point.

6.2 Stochastic gradient descent

The Gabor model has two parameters, so we could find the global minimum by either (i) exhaustively searching the parameter space or (ii) repeatedly starting gradient descent from different positions and choosing the result with the lowest loss. However, neural network models can have millions of parameters, so neither approach is practical. In short, using gradient descent to find the global optimum of a high-dimensional loss function is challenging. We can find *a* minimum, but there is no way to tell whether this

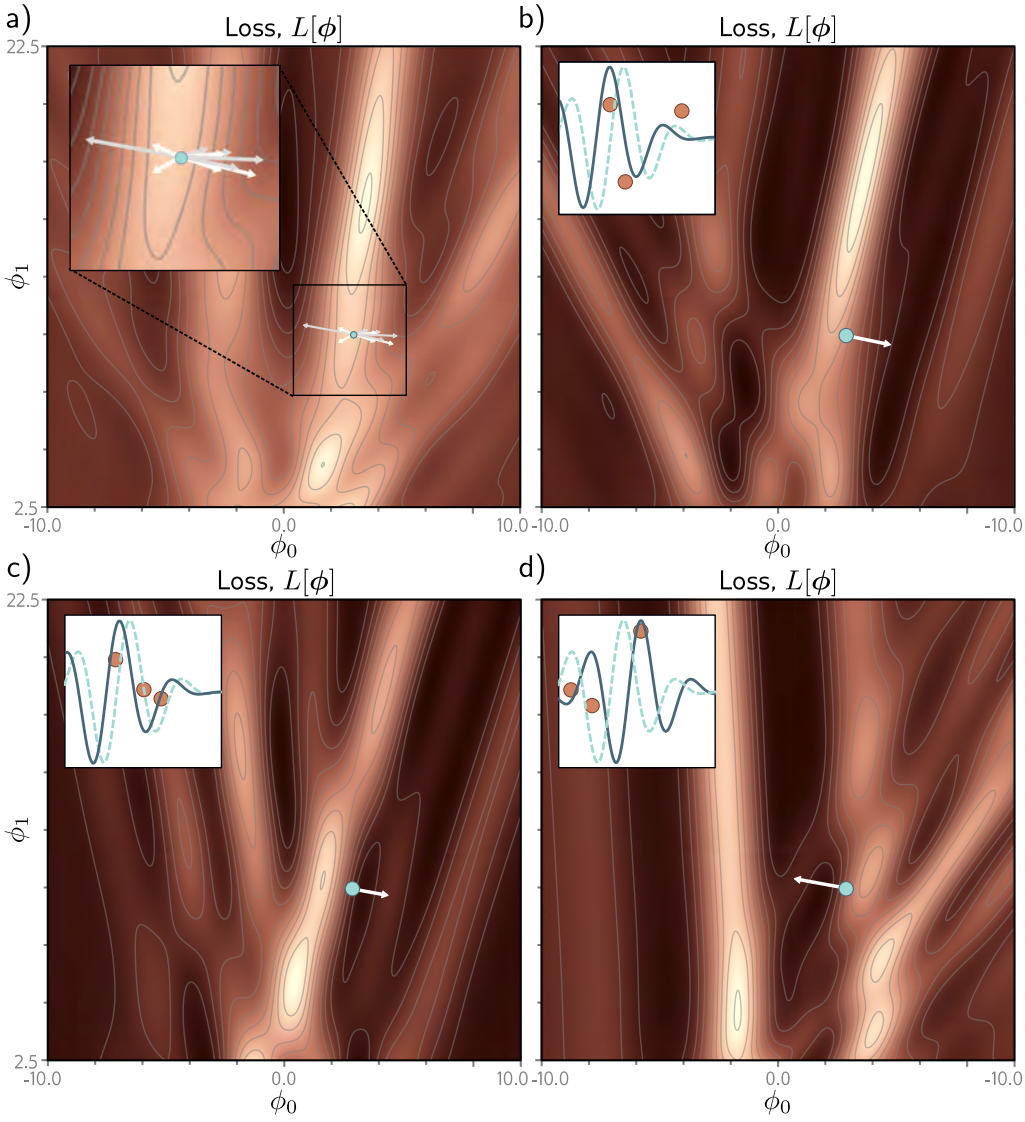


Figure 6.6 Alternative view of SGD for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. b) Loss function for one possible batch. The SGD algorithm moves in the downhill direction on this function for a distance that is determined by the learning rate and the local gradient magnitude. The current model (dashed function in inset) changes to better fit the batch data (solid function). c) A different batch creates a different loss function and results in a different update. d) For this batch, the algorithm moves *downhill* with respect to the batch loss function but *uphill* with respect to the global loss function in panel (a). This is how SGD can escape local minima.

is the global minimum or even a good one.

One of the main problems is that the final destination of a gradient descent algorithm is entirely determined by the starting point. *Stochastic gradient descent (SGD)* attempts to remedy this problem by adding some noise to the gradient at each step. The solution still moves downhill on average, but at any given iteration, the direction chosen is not necessarily in the steepest downhill direction. Indeed, it might not be downhill at all. The SGD algorithm has the possibility of moving temporarily uphill and hence jumping from one "valley" of the loss function to another (figure 6.5b).

Notebook 6.3 Stochastic gradient descent

6.2.1 Batches and epochs

The mechanism for introducing randomness is simple. At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone. This subset is known as a *minibatch* or *batch* for short. The update rule for the model parameters $\boldsymbol{\phi}_t$ at iteration t is hence:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{6.10}$$

where $\mathcal{B}_t$ is a set containing the indices of the input/output pairs in the current batch and, as before, ℓ_i is the loss due to the i^{th} pair. The term α is the learning rate, and together with the gradient magnitude, determines the distance moved at each iteration. The learning rate is chosen at the start of the procedure and does not depend on the local properties of the function.

The batches are usually drawn from the dataset without replacement. The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again. A single pass through the entire training dataset is referred to as an *epoch*. A batch may be as small as a single example or as large as the whole dataset. The latter case is called *full-batch gradient descent* and is identical to regular (non-stochastic) gradient descent.

Problem 6.9

An alternative interpretation of SGD is that it computes the gradient of a different loss function at each iteration; the loss function depends on both the model and the training data and hence will differ for each randomly selected batch. In this view, SGD performs deterministic gradient descent on a constantly changing loss function (figure 6.6). However, despite this variability, the expected loss and expected gradients at any point remain the same as for gradient descent.

6.2.2 Properties of stochastic gradient descent

SGD has several attractive features. First, although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration. Hence, the updates tend to be sensible even if they are not optimal. Second, because it draws training examples without replacement and iterates through the dataset, the training examples all still contribute equally. Third, it is less computationally expensive to compute the gradient

from just a subset of the training data. Fourth, it can (in principle) escape local minima. Fifth, it reduces the chances of getting stuck near saddle points; it is likely that at least some of the possible batches will have a significant gradient at any point on the loss function. Finally, there is some evidence that SGD finds parameters for neural networks that cause them to generalize well to new data in practice (see section 9.2).

SGD does not necessarily "converge" in the traditional sense. However, the hope is that when we are close to the global minimum, all the data points will be well described by the model. Consequently, the gradient will be small, whichever batch is chosen, and the parameters will cease to change much. In practice, SGD is often applied with a *learning rate schedule*. The learning rate α starts at a high value and is decreased by a constant factor every N epochs. The logic is that in the early stages of training, we want the algorithm to explore the parameter space, jumping from valley to valley to find a sensible region. In later stages, we are roughly in the right place and are more concerned with fine-tuning the parameters, so we decrease α to make smaller changes.

6.3 Momentum

A common modification to stochastic gradient descent is to add a *momentum* term. We update the parameters with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned} \tag{6.11}$$

where $\mathbf{m}_t$ is the momentum (which drives the update at iteration t), $\beta \in [0, 1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.

The recursive formulation of the momentum calculation means that the gradient step is an infinite weighted sum of all the previous gradients, where the weights get smaller as we move back in time. The effective learning rate increases if all these gradients are aligned over multiple iterations but decreases if the gradient direction repeatedly changes as the terms in the sum cancel out. The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys (figure 6.7).

Problem 6.10

6.3.1 Nesterov accelerated momentum

Notebook 6.4 Momentum

The momentum term can be considered a coarse prediction of where the SGD algorithm will move next. Nesterov accelerated momentum (figure 6.8) computes the gradients at this predicted point rather than at the current point:

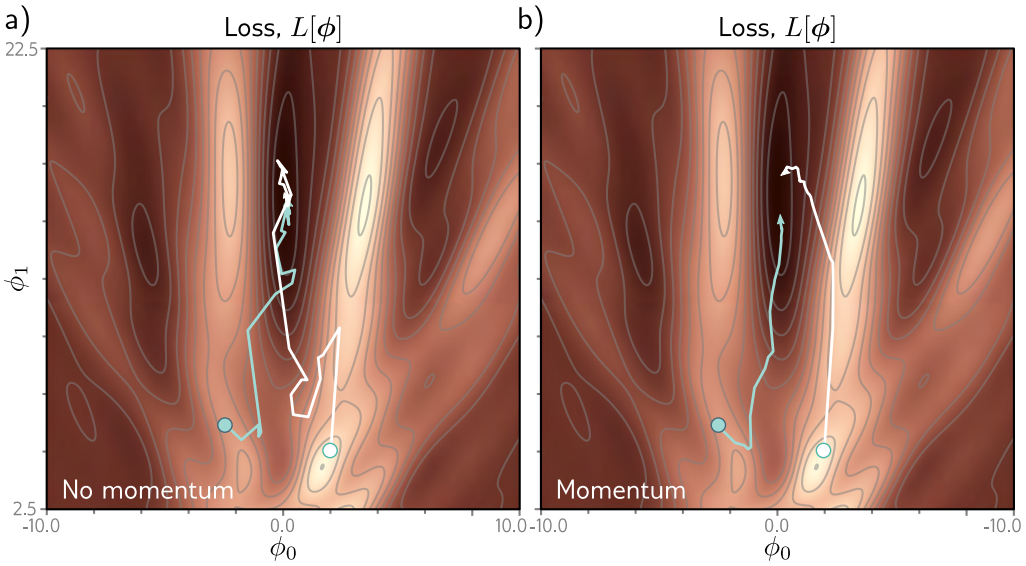


Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

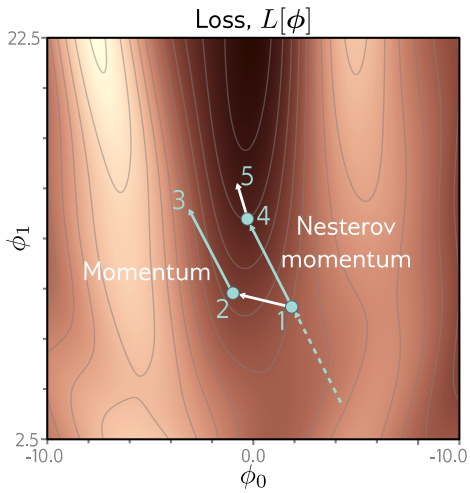


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t - \alpha\beta\cdot\mathbf{m}_t]}{\partial\boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha\cdot\mathbf{m}_{t+1},\end{aligned} \tag{6.12}$$

where now the gradients are evaluated at $\boldsymbol{\phi}_t - \alpha\beta\cdot\mathbf{m}_t$. One way to think about this is that the gradient term now corrects the path provided by momentum alone.

6.4 Adam

Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further). When the gradient of the loss surface is much steeper in one direction than another, it is difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable (figures 6.9a–b).

A straightforward approach is to normalize the gradients so that we move a fixed distance (governed by the learning rate) in each direction. To do this, we first measure the gradient $\mathbf{m}_{t+1}$ and the pointwise squared gradient $\mathbf{v}_{t+1}$:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial\boldsymbol{\phi}}\right)^2.\end{aligned} \tag{6.13}$$

Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha\cdot\frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}}+\epsilon}, \tag{6.14}$$

where the square root and division are both pointwise, α is the learning rate, and ϵ is a small constant that prevents division by zero when the gradient magnitude is zero. The term $\mathbf{v}_{t+1}$ is the squared gradient, and the positive root of this is used to normalize the gradient itself, so all that remains is the sign in each coordinate direction. The result is that the algorithm moves a fixed distance α along each coordinate, where the direction is determined by whichever way is downhill (figure 6.9c). This simple algorithm makes good progress in both directions but will not converge unless it happens to land exactly at the minimum. Instead, it will bounce back and forth around the minimum.

Adaptive moment estimation, or *Adam*, takes this idea and adds momentum to both the estimate of the gradient and the squared gradient:

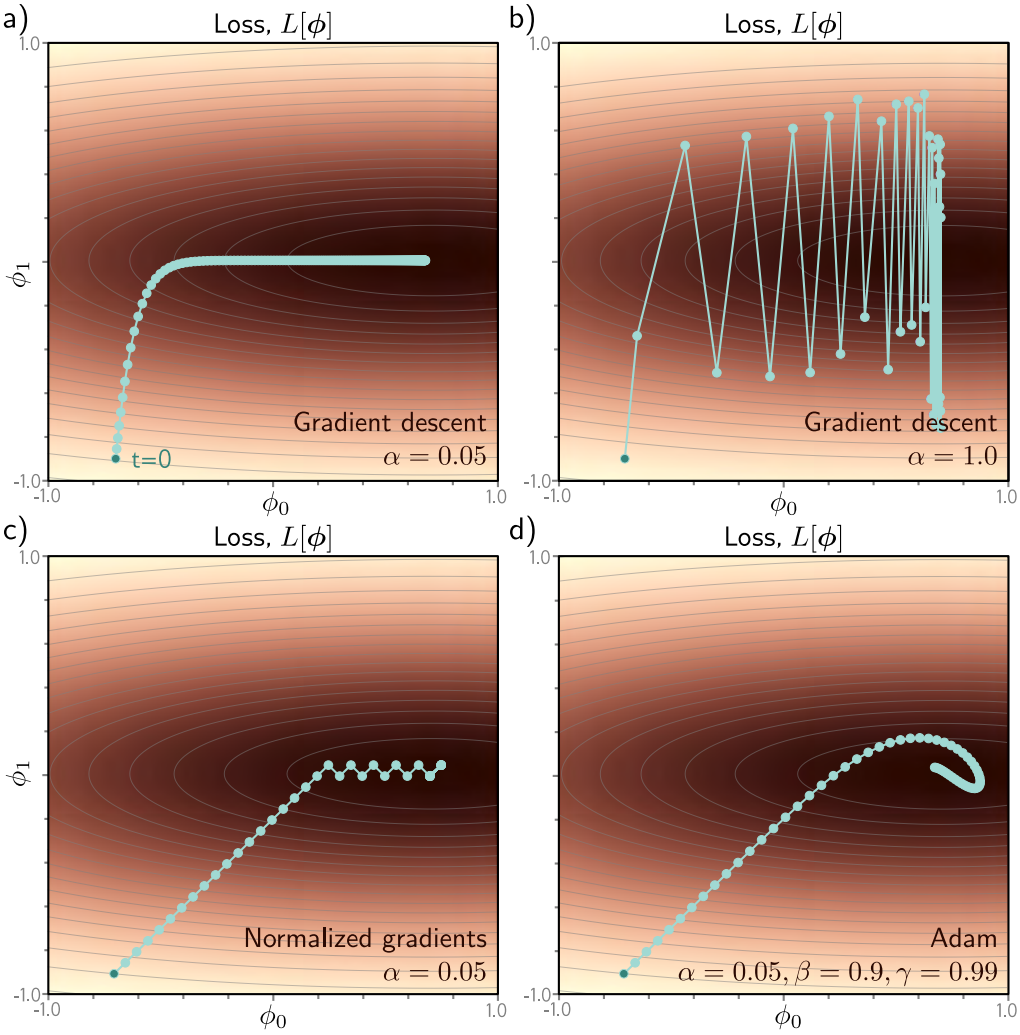


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned} \tag{6.15}$$

where β and γ are the momentum coefficients for the two statistics.

Using momentum is equivalent to taking a weighted average over the history of each of these statistics. At the start of the procedure, all the previous measurements are effectively zero, resulting in unrealistically small estimates. Consequently, we modify these statistics using the rule:

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned} \tag{6.16}$$

Since β and γ are in the range $[0, 1)$, the terms with exponents $t+1$ become smaller with each time step, the denominators become closer to one, and this modification has a diminishing effect.

Finally, we update the parameters as before, but with the modified terms:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}. \tag{6.17}$$

Notebook 6.5 Adam

The result is an algorithm that can converge to the overall minimum and makes good progress in every direction in the parameter space. Note that Adam is usually used in a stochastic setting where the gradients and their squares are computed from mini-batches:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i \in \mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\sum_{i \in \mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned} \tag{6.18}$$

and so the trajectory is noisy in practice.

As we shall see in chapter 7, the gradient magnitudes of neural network parameters can depend on their depth in the network. Adam helps compensate for this tendency and balances out changes across the different layers. In practice, Adam also has the advantage of being less sensitive to the initial learning rate because it avoids situations like those in figures 6.9a–b, so it doesn't need complex learning rate schedules.

6.5 Training algorithm hyperparameters

The choices of learning algorithm, batch size, learning rate schedule, and momentum coefficients are all considered *hyperparameters* of the training algorithm; these directly affect the final model performance but are distinct from the model parameters. Choosing these can be more art than science, and it's common to train many models with different hyperparameters and choose the best one. This is known as *hyperparameter search*. We return to this issue in chapter 8.

6.6 Summary

This chapter discussed model training. This problem was framed as finding parameters $\boldsymbol{\phi}$ that corresponded to the minimum of a loss function $L[\boldsymbol{\phi}]$. The gradient descent method measures the gradient of the loss function for the current parameters (i.e., how the loss changes when we make a small change to the parameters). Then it moves the parameters in the direction that decreases the loss fastest. This is repeated until convergence.

For nonlinear functions, the loss function may have both local minima (where gradient descent gets trapped) and saddle points (where gradient descent may appear to have converged but has not). Stochastic gradient descent helps mitigate these problems.[1] At each iteration, we use a different random subset of the data (a batch) to compute the gradient. This adds noise to the process and helps prevent the algorithm from getting trapped in a sub-optimal region of parameter space. Each iteration is also computationally cheaper since it only uses a subset of the data. We saw that adding a momentum term makes convergence more efficient. Finally, we introduced the Adam algorithm.

The ideas in this chapter apply to optimizing *any* model. The next chapter tackles two aspects of training specific to neural networks. First, we address how to compute the gradients of the loss with respect to the parameters of a neural network. This is accomplished using the famous backpropagation algorithm. Second, we discuss how to initialize the network parameters before optimization begins. Without careful initialization, the gradients used by the optimization can become extremely large or extremely small, which can hinder the training process.

Notes

Optimization algorithms: Optimization algorithms are used extensively throughout engineering, and it is generally more typical to use the term *objective function* rather than loss function or cost function. Gradient descent was invented by Cauchy (1847), and stochastic gradient descent dates back to at least Robbins & Monro (1951). A modern compromise between the two is stochastic variance-reduced descent (Johnson & Zhang, 2013), in which the full gradient is computed periodically, with stochastic updates interspersed. Reviews of optimization algorithms for neural networks can be found in Ruder (2016), Bottou et al. (2018), and Sun (2020). Bottou (2012) discusses best practice for SGD, including shuffling without replacement.

[1] Chapter 20 discusses the extent to which saddle points and local minima really *are* problems in deep learning. In practice, deep networks are surprisingly easy to train.

Convexity, minima, and saddle points: A function is convex if no chord (line segment between two points on the surface) intersects the function. This can be tested algebraically by considering the *Hessian matrix* (the matrix of second derivatives):

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_N} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} & \cdots & \frac{\partial^2 L}{\partial \phi_1 \partial \phi_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial \phi_N \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_N \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_N^2} \end{bmatrix}. \tag{6.19}$$

Appendix B.3.7 Eigenvalues

If the Hessian matrix is positive definite (has positive eigenvalues) for all possible parameter values, then the function is convex; the loss function will look like a smooth bowl (as in figure 6.1c), so training will be relatively easy. There will be a single global minimum and no local minima or saddle points.

For any loss function, the eigenvalues of the Hessian matrix at places where the gradient is zero allow us to classify this position as (i) a minimum (the eigenvalues are all positive), (ii) a maximum (the eigenvalues are all negative), or (iii) a saddle point (positive eigenvalues are associated with directions in which we are at a minimum and negative ones with directions where we are at a maximum).

Line search: Gradient descent with a fixed step size is inefficient because the distance moved depends entirely on the magnitude of the gradient. It moves a long distance when the function is changing fast (where perhaps it should be more cautious) but a short distance when the function is changing slowly (where perhaps it should explore further). For this reason, gradient descent methods are usually combined with a line search procedure in which we sample the function along the desired direction to try to find the optimal step size. One such approach is bracketing (figure 6.10). Another problem with gradient descent is that it tends to lead to inefficient oscillatory behavior when descending valleys (e.g., path 1 in figure 6.5a).

Beyond gradient descent: Numerous algorithms have been developed that remedy the problems of gradient descent. Most notable is the Newton method, which takes the curvature of the surface into account using the inverse of the Hessian matrix; if the gradient of the function is changing quickly, then it applies a more cautious update. This method eliminates the need for line search and does not suffer from oscillatory behavior. However, it has its own problems; in its simplest form, it moves toward the nearest extremum, but this may be a maximum if we are closer to the top of a hill than we are to the bottom of a valley. Moreover, computing the inverse Hessian is intractable when the number of parameters is large, as in neural networks.

Problem 6.11

Properties of SGD: The limit of SGD as the learning rate tends to zero is a stochastic differential equation. Jastrzębski et al. (2018) showed that this equation relies on the learning-rate to batch size ratio and that there is a relation between the learning rate to batch size ratio and the width of the minimum found. Wider minima are considered more desirable; if the loss function for test data is similar, then small errors in the parameter estimates will have little effect on test performance. He et al. (2019) prove a generalization bound for SGD that has a positive correlation with the ratio of batch size to learning rate. They train a large number of models on different architectures and datasets and find empirical evidence that test accuracy improves when the ratio of batch size to learning rate is low. Smith et al. (2018) and Goyal et al. (2018) also identified the ratio of batch size to learning rate as being important for generalization (see figure 20.10).

Momentum: The idea of using momentum to speed up optimization dates to Polyak (1964). Goh (2017) presents an in-depth discussion of the properties of momentum. The Nesterov

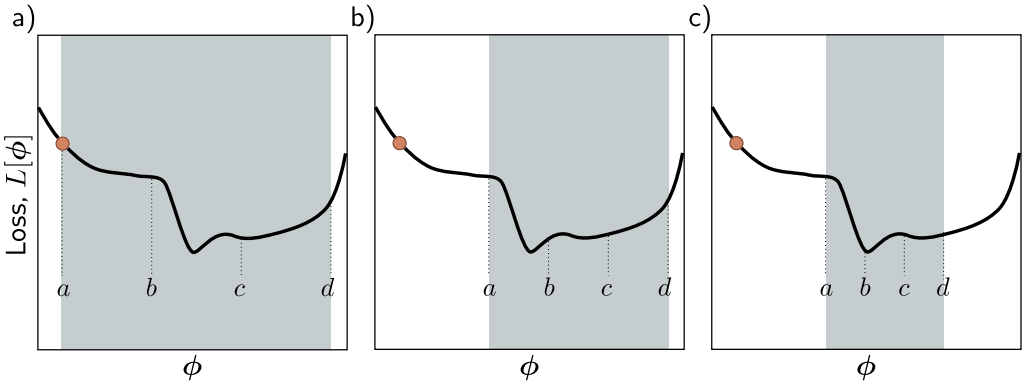


Figure 6.10 Line search using the bracketing approach. a) The current solution is at position a (orange point), and we wish to search the region $[a, d]$ (gray shaded area). We define two points b, c interior to the search region and evaluate the loss function at these points. Here $L[b] > L[c]$, so we eliminate the range $[a, b]$. b) We now repeat this procedure in the refined search region and find that $L[b] < L[c]$, so we eliminate the range $[c, d]$. c) We repeat this process until this minimum is closely bracketed.

accelerated gradient method was introduced by Nesterov (1983). Nesterov momentum was first applied in the context of stochastic gradient descent by Sutskever et al. (2013).

Adaptive training algorithms: AdaGrad (Duchi et al., 2011) is an optimization algorithm that addresses the possibility that some parameters may have to move further than others by assigning a different learning rate to each parameter. AdaGrad uses the cumulative squared gradient for each parameter to attenuate its learning rate. This has the disadvantage that the learning rates decrease over time, and learning can halt before the minimum is found. RMSProp (Hinton et al., 2012a) and AdaDelta (Zeiler, 2012) modified this algorithm to help prevent these problems by recursively updating the squared gradient term.

By far the most widely used adaptive training algorithm is adaptive moment optimization or Adam (Kingma & Ba, 2015). This combines the ideas of momentum (in which the gradient vector is averaged over time) and AdaGrad, AdaDelta, and RMSProp (in which a smoothed squared gradient term is used to modify the learning rate for each parameter). The original paper on the Adam algorithm provided a convergence proof for convex loss functions, but a counterexample was identified by Reddi et al. (2018), who developed a modification of Adam called AMSGrad, which does converge. Of course, in deep learning, the loss functions are non-convex, and Zaheer et al. (2018) subsequently developed an adaptive algorithm called YOGI and proved that it converges in this scenario. Regardless of these theoretical objections, the original Adam algorithm works well in practice and is widely used, not least because it works well over a broad range of hyperparameters and makes rapid initial progress.

One potential problem with adaptive training algorithms is that the learning rates are based on accumulated statistics of the observed gradients. At the start of training, when there are few samples, these statistics may be very noisy. This can be remedied by *learning rate warm-up* (Goyal et al., 2018), in which the learning rates are gradually increased over the first few thousand iterations. An alternative solution is rectified Adam (Liu et al., 2021a), which gradually

changes the momentum term over time in a way that helps avoid high variance. Dozat (2016) incorporated Nesterov momentum into the Adam algorithm.

SGD vs. Adam: There has been a lively discussion about the relative merits of SGD and Adam. Wilson et al. (2017) provided evidence that SGD with momentum can find lower minima than Adam, which generalizes better over a variety of deep learning tasks. However, this is strange since SGD is a special case of Adam (when $\beta = \gamma = 0$) once the modification term (equation 6.16) becomes one, which happens quickly. It is hence more likely that SGD outperforms Adam *when we use Adam's default hyperparameters*. Loshchilov & Hutter (2019) proposed AdamW, which substantially improves the performance of Adam in the presence of L2 regularization (see section 9.1). Choi et al. (2019) provide evidence that if we search for the best Adam hyperparameters, it performs just as well as SGD and converges faster. Keskar & Socher (2017) proposed a method called SWATS that starts using Adam (to make rapid initial progress) and then switches to SGD (to get better final generalization performance).

Exhaustive search: All the algorithms discussed in this chapter are iterative. A completely different approach is to quantize the network parameters and exhaustively search the resulting discretized parameter space using SAT solvers (Mézard & Mora, 2009). This approach has the potential to find the global minimum and provide a guarantee that there is no lower loss elsewhere but is only practical for very small models.

Problems

Problem 6.1 Show that the derivatives of the least squares loss function in equation 6.5 are given by the expressions in equation 6.7.

Problem 6.2 A surface is convex if the eigenvalues of the Hessian $\mathbf{H}[\boldsymbol{\phi}]$ are positive everywhere. In this case, the surface has a unique minimum, and optimization is easy. Find an algebraic expression for the Hessian matrix,

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} \end{bmatrix}, \tag{6.20}$$

for the linear regression model (equation 6.5). Prove that this function is convex by showing that the eigenvalues are always positive. This can be done by showing that both the trace and the determinant of the matrix are positive.

Appendix B.3.7 Eigenvalues

Appendix B.3.8 Trace

Appendix B.3.8 Determinant

Problem 6.3 Compute the derivatives of the least squares loss $L[\boldsymbol{\phi}]$ with respect to the parameters ϕ_0 and ϕ_1 for the Gabor model (equation 6.8).

Problem 6.4* The logistic regression model uses a linear function to assign an input $\mathbf{x}$ to one of two classes $y \in \{0, 1\}$. For a 1D input and a 1D output, it has two parameters, ϕ_0 and ϕ_1, and is defined by:

$$Pr(y = 1|x) = \text{sig}[\phi_0 + \phi_1 x], \tag{6.21}$$

where $\text{sig}[\bullet]$ is the logistic sigmoid function:

$$\text{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{6.22}$$

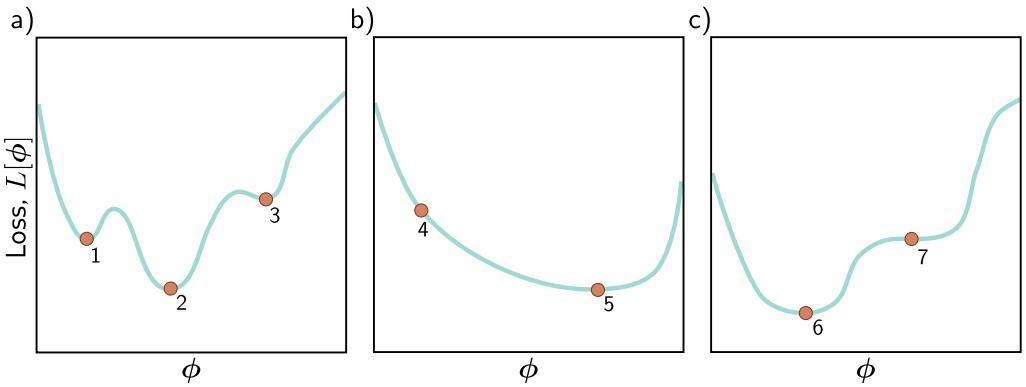


Figure 6.11 Three 1D loss functions for problem 6.6.

(i) Plot y against x for this model for different values of ϕ_0 and ϕ_1 and explain the qualitative meaning of each parameter. (ii) What is a suitable loss function for this model? (iii) Compute the derivatives of this loss function with respect to the parameters. (iv) Generate ten data points from a normal distribution with mean -1 and standard deviation 1 and assign them the label $y = 0$. Generate another ten data points from a normal distribution with mean 1 and standard deviation 1 and assign these the label $y = 1$. Plot the loss as a heatmap in terms of the two parameters ϕ_0 and ϕ_1. (v) Is this loss function convex? How could you prove this?

Problem 6.5* Compute the derivatives of the least squares loss with respect to the ten parameters of the simple neural network model introduced in equation 3.1:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x]. \tag{6.23}$$

Think carefully about what the derivative of the ReLU function $a[\bullet]$ will be.

Problem 6.6 Which of the functions in figure 6.11 is convex? Justify your answer. Characterize each of the points 1–7 as (i) a local minimum, (ii) the global minimum, or (iii) neither.

Problem 6.7* The gradient descent trajectory for path 1 in figure 6.5a oscillates back and forth inefficiently as it moves down the valley toward the minimum. It's also notable that it turns at right angles to the previous direction at each step. Provide a qualitative explanation for these phenomena. Propose a solution that might help prevent this behavior.

Problem 6.8* Can (non-stochastic) gradient descent with a *fixed* learning rate escape local minima?

Problem 6.9 We run the stochastic gradient descent algorithm for 1,000 iterations on a dataset of size 100 with a batch size of 20. For how many epochs did we train the model?

Problem 6.10 Show that the momentum term $\mathbf{m}_t$ (equation 6.11) is an infinite weighted sum of the gradients at the previous iterations and derive an expression for the coefficients (weights) of that sum.

Problem 6.11 What dimensions will the Hessian have if the model has one million parameters?

Chapter 7

Gradients and initialization

Chapter 6 introduced iterative optimization algorithms. These are general-purpose methods for finding the minimum of a function. In the context of neural networks, they find parameters that minimize the loss so that the model accurately predicts the training outputs from the inputs. The basic approach is to choose initial parameters randomly and then make a series of small changes that decrease the loss on average. Each change is based on the gradient of the loss with respect to the parameters at the current position.

This chapter discusses two issues that are specific to neural networks. First, we consider how to calculate the gradients efficiently. This is a serious challenge since the largest models at the time of writing have $\sim 10^{12}$ parameters, and the gradient needs to be computed for every parameter at every iteration of the training algorithm. Second, we consider how to initialize the parameters. If this is not done carefully, the initial losses and their gradients can be extremely large or small. In either case, this impedes the training process.

7.1 Problem definitions

Consider a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with multivariate input $\mathbf{x}$, parameters $\boldsymbol{\phi}$, and three hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \\ \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3, \end{aligned} \tag{7.1}$$

where the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of the input. The model parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_0, \boldsymbol{\Omega}_0, \boldsymbol{\beta}_1, \boldsymbol{\Omega}_1, \boldsymbol{\beta}_2, \boldsymbol{\Omega}_2, \boldsymbol{\beta}_3, \boldsymbol{\Omega}_3\}$ consist of the bias vectors $\boldsymbol{\beta}_k$ and weight matrices $\boldsymbol{\Omega}_k$ between every layer (figure 7.1).

We also have individual loss terms ℓ_i, which return the negative log-likelihood of the ground truth label y_i given the model prediction $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for training input $\mathbf{x}_i$. For example, this might be the least squares loss $\ell_i = (\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}] - y_i)^2$. The total loss is the sum of these terms over the training data:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i. \tag{7.2}$$

The most commonly used optimization algorithm for training neural networks is stochastic gradient descent (SGD), which updates the parameters as:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{7.3}$$

where α is the learning rate, and $\mathcal{B}_t$ contains the batch indices at iteration t. To compute this update, we need to calculate the derivatives:

$$\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} \quad \text{and} \quad \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k}, \tag{7.4}$$

for the parameters $\{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$ at every layer $k \in \{0, 1, \ldots, K\}$ and for each index i in the batch. The first part of this chapter describes the *backpropagation algorithm*, which computes these derivatives efficiently.

Problem 7.1

In the second part of the chapter, we consider how to initialize the network parameters before we commence training. We describe methods to choose the initial weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$ so that training is stable.

7.2 Computing derivatives

The derivatives of the loss tell us how the loss changes when we make a small change to the parameters. Optimization algorithms exploit this information to manipulate the parameters so that the loss becomes smaller. The *backpropagation algorithm* computes these derivatives. The mathematical details are somewhat involved, so we first make two observations that provide some intuition.

Observation 1: Each weight (element of $\boldsymbol{\Omega}_k$) multiplies the activation at a source hidden unit and adds the result to a destination hidden unit in the next layer. It follows that the effect of any small change to the weight is amplified or attenuated by the activation at the source hidden unit. Hence, we run the network for each data example in the batch and store the activations of all the hidden units. This is known as the *forward pass* (figure 7.1). The stored activations will subsequently be used to compute the gradients.

Observation 2: A small change in a bias or weight causes a ripple effect of changes through the subsequent network. The change modifies the value of its destination hidden

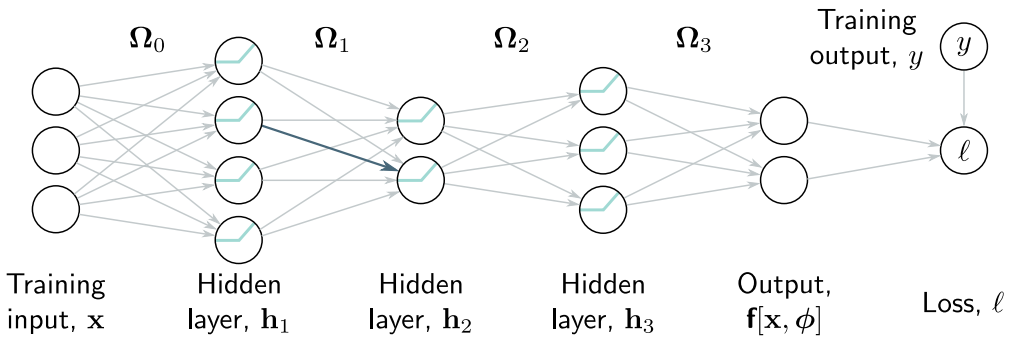


Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss ℓ with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the blue weight is applied to the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the blue weight will double too. Hence, to compute the derivatives of the weights, we need to calculate and store the activations at the hidden layers. This is known as the *forward pass* since it involves running the network equations sequentially.

unit. This, in turn, changes the values of the hidden units in the subsequent layer, which will change the hidden units in the layer after that, and so on, until a change is made to the model output and, finally, the loss.

Hence, to know how changing a parameter modifies the loss, we also need to know how changes to every subsequent hidden layer will, in turn, modify their successor. These same quantities are required when considering other parameters in the same or earlier layers. It follows that we can calculate them once and reuse them. For example, consider computing the effect of a small change in weights that feed into hidden layers $\mathbf{h}_3$, $\mathbf{h}_2$, and $\mathbf{h}_1$, respectively:

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_3$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_3$ changes the model output $\mathbf{f}$, and (ii) how a change in this output changes the loss ℓ (figure 7.2a).

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_2$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_2$ affects $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes the model output, and (iii) how this output changes the loss (figure 7.2b).

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_1$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_1$ affects layer $\mathbf{h}_2$, (ii) how a change in layer $\mathbf{h}_2$ affects layer $\mathbf{h}_3$, (iii) how layer $\mathbf{h}_3$ changes the model output, and (iv) how the model output changes the loss (figure 7.2c).

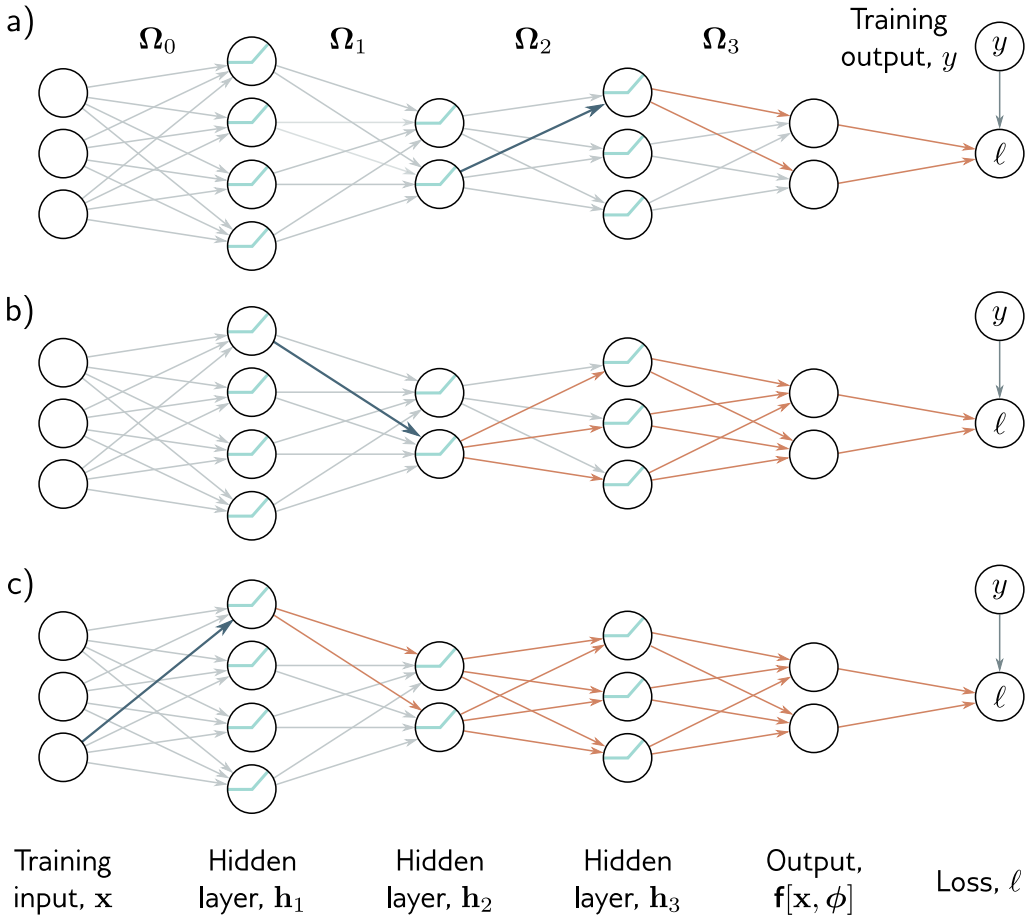


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

As we move backward through the network, we see that most of the terms we need were already calculated in the previous step, so we do not need to re-compute them. Proceeding backward through the network in this way to compute the derivatives is known as the *backward pass*.

The ideas behind backpropagation are relatively easy to understand. However, the derivation requires matrix calculus because the bias and weight terms are vectors and matrices, respectively. To help grasp the underlying mechanics, the following section derives backpropagation for a simpler toy model with scalar parameters. We then apply the same approach to a deep neural network in section 7.4.

7.3 Toy example

Consider a model $\mathrm{f}[x, \boldsymbol{\phi}]$ with eight scalar parameters $\boldsymbol{\phi} = \{\beta_0, \omega_0, \beta_1, \omega_1, \beta_2, \omega_2, \beta_3, \omega_3\}$ that consists of a composition of the functions $\sin[\bullet], \exp[\bullet]$, and $\cos[\bullet]$:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x]\big]\Big], \tag{7.5}$$

and a least squares loss function $L[\boldsymbol{\phi}] = \sum_i \ell_i$ with individual terms:

$$\ell_i = (\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i)^2, \tag{7.6}$$

where, as usual, x_i is the i^{th} training input, and y_i is the i^{th} training output. You can think of this as a simple neural network with one input, one output, one hidden unit at each layer, and different activation functions $\sin[\bullet], \exp[\bullet]$, and $\cos[\bullet]$ between each layer.

We aim to compute the derivatives:

$$\frac{\partial \ell_i}{\partial \beta_0}, \quad \frac{\partial \ell_i}{\partial \omega_0}, \quad \frac{\partial \ell_i}{\partial \beta_1}, \quad \frac{\partial \ell_i}{\partial \omega_1}, \quad \frac{\partial \ell_i}{\partial \beta_2}, \quad \frac{\partial \ell_i}{\partial \omega_2}, \quad \frac{\partial \ell_i}{\partial \beta_3}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial \omega_3}.$$

Of course, we could find expressions for these derivatives by hand and compute them directly. However, some of these expressions are quite complex. For example:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \omega_0} = & -2\Big(\beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\big]\Big] - y_i\Big) \\ & \cdot \omega_1\omega_2\omega_3 \cdot x_i \cdot \cos[\beta_0 + \omega_0 \cdot x_i] \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big] \\ & \cdot \sin\Big[\beta_2 + \omega_2 \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big]\Big].\end{aligned} \tag{7.7}$$

Such expressions are awkward to derive and code without mistakes and do not exploit the inherent redundancy; notice that the three exponential terms are the same.

The backpropagation algorithm is an efficient method for computing all of these derivatives at once. It consists of (i) a forward pass, in which we compute and store a series of intermediate values and the network output, and (ii) a backward pass, in which

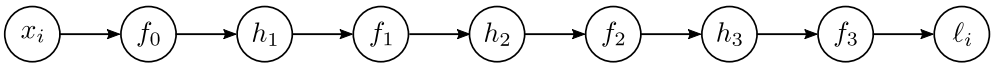


Figure 7.3 Backpropagation forward pass. We compute and store each of the intermediate variables in turn until we finally calculate the loss.

we calculate the derivatives of each parameter, starting at the end of the network, and reusing previous calculations as we move toward the start.

Forward pass: We treat the computation of the loss as a series of calculations:

$$\begin{aligned} f_0 &= \beta_0 + \omega_0 \cdot x_i \\ h_1 &= \sin[f_0] \\ f_1 &= \beta_1 + \omega_1 \cdot h_1 \\ h_2 &= \exp[f_1] \\ f_2 &= \beta_2 + \omega_2 \cdot h_2 \\ h_3 &= \cos[f_2] \\ f_3 &= \beta_3 + \omega_3 \cdot h_3 \\ \ell_i &= (f_3 - y_i)^2. \end{aligned} \tag{7.8}$$

We compute and store the values of the intermediate variables f_k and h_k (figure 7.3).

Backward pass #1: We now compute the derivatives of ℓ_i with respect to these intermediate variables, but in reverse order:

$$\frac{\partial \ell_i}{\partial f_3}, \quad \frac{\partial \ell_i}{\partial h_3}, \quad \frac{\partial \ell_i}{\partial f_2}, \quad \frac{\partial \ell_i}{\partial h_2}, \quad \frac{\partial \ell_i}{\partial f_1}, \quad \frac{\partial \ell_i}{\partial h_1}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial f_0}. \tag{7.9}$$

The first of these derivatives is straightforward:

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y_i). \tag{7.10}$$

The next derivative can be calculated using the chain rule:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}. \tag{7.11}$$

The left-hand side asks how ℓ_i changes when h_3 changes. The right-hand side says we can decompose this into (i) how f_3 changes when h_3 changes and (ii) how ℓ_i changes when f_3 changes. In the original equations, h_3 changes f_3, which changes ℓ_i, and the derivatives

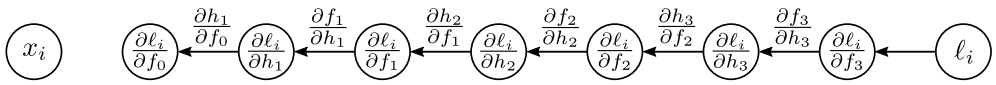


Figure 7.4 Backpropagation backward pass #1. We work backward from the end of the function computing the derivatives $\partial \ell_i / \partial f_k$ and $\partial \ell_i / \partial h_k$ of the loss with respect to the intermediate quantities. Each derivative is computed from the previous one by multiplying by terms of the form $\partial f_k / \partial h_k$ or $\partial h_k / \partial f_{k-1}$.

represent the effects of this chain. Notice that we already computed the second of these derivatives, and the other is the derivative of $\beta_3 + \omega_3 \cdot h_3$ with respect to h_3, which is ω_3.

We continue in this way, computing the derivatives of the output with respect to these intermediate quantities (figure 7.4):

$$\begin{aligned}
\frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right).
\end{aligned} \tag{7.12}$$

Problem 7.2

In each case, we have already computed the quantities in the brackets in the previous step, and the last term has a simple expression. These equations embody Observation 2 from the previous section (figure 7.2); we can reuse the previously computed derivatives if we calculate them in reverse order.

Backward pass #2: Finally, we consider how the loss ℓ_i changes when we change the parameters $\{\beta_k\}$ and $\{\omega_k\}$. Once more, we apply the chain rule (figure 7.5):

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\
\frac{\partial \ell_i}{\partial \omega_k} &= \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}.
\end{aligned} \tag{7.13}$$

In each case, the second term on the right-hand side was computed in equation 7.12. When $k > 0$, we have $f_k = \beta_k + \omega_k \cdot h_k$, so:

$$\frac{\partial f_k}{\partial \beta_k} = 1 \qquad \text{and} \qquad \frac{\partial f_k}{\partial \omega_k} = h_k. \tag{7.14}$$

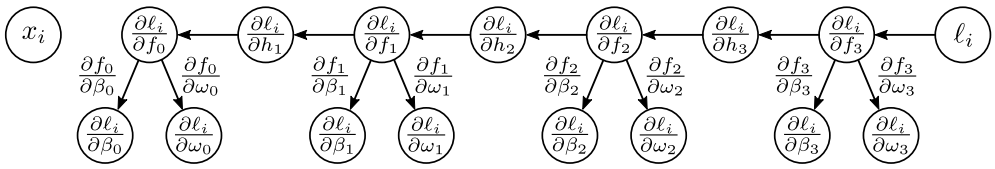


Figure 7.5 Backpropagation backward pass #2. Finally, we compute the derivatives $\partial \ell_i / \partial \beta_k$ and $\partial \ell_i / \partial \omega_k$. Each derivative is computed by multiplying the term $\partial \ell_i / \partial f_k$ by $\partial f_k / \partial \beta_k$ or $\partial f_k / \partial \omega_k$ as appropriate.

This is consistent with Observation 1 from the previous section; the effect of a change in the weight ω_k is proportional to the value of the source variable h_k (which was stored in the forward pass). The final derivatives from the term $f_0 = \beta_0 + \omega_0 \cdot x_i$ are:

Notebook 7.1 Backpropagation in toy model

$$\frac{\partial f_0}{\partial \beta_0} = 1 \qquad \text{and} \qquad \frac{\partial f_0}{\partial \omega_0} = x_i. \tag{7.15}$$

Backpropagation is both simpler and more efficient than computing the derivatives individually, as in equation 7.7.[1]

7.4 Backpropagation algorithm

Now we repeat this process for a three-layer network (figure 7.1). The intuition and much of the algebra are identical. The main differences are that intermediate variables $\mathbf{f}_k, \mathbf{h}_k$ are vectors, the biases $\boldsymbol{\beta}_k$ are vectors, the weights $\boldsymbol{\Omega}_k$ are matrices, and we are using ReLU functions rather than simple algebraic functions like $\cos[\bullet]$.

Forward pass: We write the network as a series of sequential calculations:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_1 &= \mathbf{a}[\mathbf{f}_0] \\
\mathbf{f}_1 &= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1 \\
\mathbf{h}_2 &= \mathbf{a}[\mathbf{f}_1] \\
\mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2 \\
\mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] \\
\mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3 \\
\ell_i &= \mathrm{l}[\mathbf{f}_3, y_i],
\end{aligned} \tag{7.16}$$

[1] Note that we did not actually need the derivatives $\partial l_i / \partial h_k$ of the loss with respect to the activations. In the final backpropagation algorithm, we will not compute these explicitly.

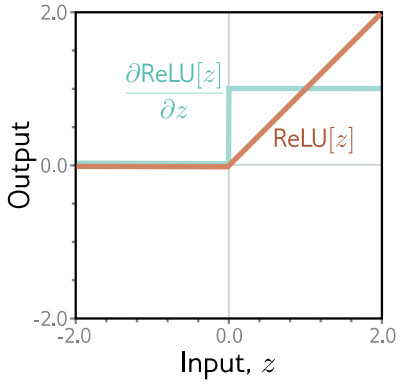


Figure 7.6 Derivative of rectified linear unit. The rectified linear unit (orange curve) returns zero when the input is less than zero and returns the input otherwise. Its derivative (cyan curve) returns zero when the input is less than zero (since the slope here is zero) and one when the input is greater than zero (since the slope here is one).

where $\mathbf{f}_{k-1}$ represents the pre-activations at the k^{th} hidden layer (i.e., the values before the ReLU function $\mathbf{a}[\bullet]$) and $\mathbf{h}_k$ contains the activations at the k^{th} hidden layer (i.e., after the ReLU function). The term $l[\mathbf{f}_3, y_i]$ represents the loss function (e.g., least squares or binary cross-entropy loss). In the forward pass, we work through these calculations and store all the intermediate quantities.

Backward pass #1: Now let's consider how the loss changes when we modify the pre-activations $\mathbf{f}_0, \mathbf{f}_1, \mathbf{f}_2$. Applying the chain rule, the expression for the derivative of the loss ℓ_i with respect to $\mathbf{f}_2$ is:

Appendix B.5 Matrix calculus

$$\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}. \tag{7.17}$$

The three terms on the right-hand side have sizes $D_3 \times D_3, D_3 \times D_f$, and $D_f \times 1$, respectively, where D_3 is the number of hidden units in the third layer, and D_f is the dimensionality of the model output $\mathbf{f}_3$.

Similarly, we can compute how the loss changes when we change $\mathbf{f}_1$ and $\mathbf{f}_0$:

$$\frac{\partial \ell_i}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}\right) \tag{7.18}$$

$$\frac{\partial \ell_i}{\partial \mathbf{f}_0} = \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_0}\frac{\partial \mathbf{f}_1}{\partial \mathbf{h}_1}\left(\frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}\right). \tag{7.19}$$

Problem 7.3

Note that in each case, the term in brackets was computed in the previous step. By working backward through the network, we can reuse the previous computations.

Problems 7.4–7.5

Moreover, the terms themselves are simple. Working backward through the right-hand side of equation 7.17, we have:

- The derivative $\partial \ell_i / \partial \mathbf{f}_3$ of the loss ℓ_i with respect to the network output $\mathbf{f}_3$ will depend on the loss function but usually has a simple form.
- The derivative $\partial \mathbf{f}_3 / \partial \mathbf{h}_3$ of the network output with respect to hidden layer $\mathbf{h}_3$ is:

$$\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} = \frac{\partial}{\partial \mathbf{h}_3}\left(\boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3\right) = \boldsymbol{\Omega}_3^T. \tag{7.20}$$

If you are unfamiliar with matrix calculus, this result is not obvious. It is explored in problem 7.6.

Problem 7.6

- The derivative $\partial \mathbf{h}_3/\partial \mathbf{f}_2$ of the output $\mathbf{h}_3$ of the activation function with respect to its input $\mathbf{f}_2$ will depend on the activation function. It will be a diagonal matrix since each activation only depends on the corresponding pre-activation. For ReLU functions, the diagonal terms are zero everywhere $\mathbf{f}_2$ is less than zero and one otherwise (figure 7.6). Rather than multiply by this matrix, we extract the diagonal terms as a vector $\mathbb{I}[\mathbf{f}_2 > 0]$ and pointwise multiply, which is more efficient.

Problems 7.7–7.8

The terms on the right-hand side of equations 7.18 and 7.19 have similar forms. As we progress back through the network, we alternately (i) multiply by the transpose of the weight matrices $\boldsymbol{\Omega}_k^T$ and (ii) threshold based on the inputs $\mathbf{f}_{k-1}$ to the hidden layer. These inputs were stored during the forward pass.

Backward pass #2: Now that we know how to compute $\partial \ell_i/\partial \mathbf{f}_k$, we can focus on calculating the derivatives of the loss with respect to the weights and biases. To calculate the derivatives of the loss with respect to the biases $\boldsymbol{\beta}_k$, we again use the chain rule:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\beta}_k}\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial}{\partial \boldsymbol{\beta}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial \ell_i}{\partial \mathbf{f}_k},\end{aligned} \tag{7.21}$$

which we already calculated in equations 7.17 and 7.18.

Similarly, the derivative for the weights matrix $\boldsymbol{\Omega}_k$, is given by:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\Omega}_k}\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial}{\partial \boldsymbol{\Omega}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial \ell_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T.\end{aligned} \tag{7.22}$$

Again, the progression from line two to line three is not obvious and is explored in problem 7.9. However, the result makes sense. The final line is a matrix of the same size as $\boldsymbol{\Omega}_k$. It depends linearly on $\mathbf{h}_k$, which was multiplied by $\boldsymbol{\Omega}_k$ in the original expression. This is also consistent with the initial intuition that the derivative of the weights in $\boldsymbol{\Omega}_k$ will be proportional to the values of the hidden units $\mathbf{h}_k$ that they multiply. Recall that we already computed these during the forward pass.

Problem 7.9

7.4.1 Backpropagation algorithm summary

We now briefly summarize the final backpropagation algorithm. Consider a deep neural network $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that takes input $\mathbf{x}_i$, has K hidden layers with ReLU activations, and individual loss term $\ell_i = \mathrm{l}[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], \mathbf{y}_i]$. The goal of backpropagation is to compute the derivatives $\partial \ell_i / \partial \boldsymbol{\beta}_k$ and $\partial \ell_i / \partial \boldsymbol{\Omega}_k$ with respect to the biases $\boldsymbol{\beta}_k$ and weights $\boldsymbol{\Omega}_k$.

Forward pass: We compute and store the following quantities:

$$\begin{aligned} \mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\ \mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] \qquad\qquad k \in \{1, 2, \ldots, K\} \\ \mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k. \qquad\qquad k \in \{1, 2, \ldots, K\} \end{aligned} \tag{7.23}$$

Backward pass: We start with the derivative $\partial \ell_i / \partial \mathbf{f}_K$ of the loss function ℓ_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned} \frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \qquad\qquad k \in \{K, K-1, \ldots, 1\} \\ \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T \qquad\qquad k \in \{K, K-1, \ldots, 1\} \\ \frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right), \qquad k \in \{K, K-1, \ldots, 1\} \end{aligned} \tag{7.24}$$

where $\odot$ denotes pointwise multiplication, and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned} \frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\ \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T. \end{aligned} \tag{7.25}$$

Problem 7.10

We calculate these derivatives for every training example in the batch and sum them together to retrieve the gradient for the SGD update.

Notebook 7.2 Backpropagation

Note that the backpropagation algorithm is extremely efficient; the most demanding computational step in both the forward and backward pass is matrix multiplication (by $\boldsymbol{\Omega}$ and $\boldsymbol{\Omega}^T$, respectively) which only requires additions and multiplications. However, it is not memory efficient; the intermediate values in the forward pass must all be stored, and this can limit the size of the model we can train.

7.4.2 Algorithmic differentiation

Although it's important to understand the backpropagation algorithm, it's unlikely that you will need to code it in practice. Modern deep learning frameworks such as PyTorch

and TensorFlow calculate the derivatives automatically, given the model specification. This is known as *algorithmic differentiation*.

Each functional component (linear transform, ReLU activation, loss function) in the framework knows how to compute its own derivative. For example, the PyTorch ReLU function $\mathbf{z}_{out} = \mathbf{relu}[\mathbf{z}_{in}]$ knows how to compute the derivative of its output $\mathbf{z}_{out}$ with respect to its input $\mathbf{z}_{in}$. Similarly, a linear function $\mathbf{z}_{out} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{z}_{in}$ knows how to compute the derivatives of the output $\mathbf{z}_{out}$ with respect to the input $\mathbf{z}_{in}$ and with respect to the parameters $\boldsymbol{\beta}$ and $\boldsymbol{\Omega}$. The algorithmic differentiation framework also knows the sequence of operations in the network and thus has all the information required to perform the forward and backward passes.

These frameworks exploit the massive parallelism of modern graphics processing units (GPUs). Computations such as matrix multiplication (which features in both the forward and backward pass) are naturally amenable to parallelization. Moreover, it's possible to perform the forward and backward passes for the entire batch in parallel if the model and intermediate results in the forward pass do not exceed the available memory.

Problem 7.11

Since the training algorithm now processes the entire batch in parallel, the input becomes a multi-dimensional *tensor*. In this context, a tensor can be considered the generalization of a matrix to arbitrary dimensions. Hence, a vector is a 1D tensor, a matrix is a 2D tensor, and a 3D tensor is a 3D grid of numbers. Until now, the training data have been 1D, so the input for backpropagation would be a 2D tensor where the first dimension indexes the batch element and the second indexes the data dimension. In subsequent chapters, we will encounter more complex structured input data. For example, in models where the input is an RGB image, the original data examples are 3D (height $\times$ width $\times$ channel). Here, the input to the learning framework would be a 4D tensor, where the extra dimension indexes the batch element.

7.4.3 Extension to arbitrary computational graphs

We have described backpropagation in a deep neural network that is naturally sequential; we calculate the intermediate quantities $\mathbf{f}_0, \mathbf{h}_1, \mathbf{f}_1, \mathbf{h}_2 \ldots, \mathbf{f}_k$ in turn. However, models need not be restricted to sequential computation. Later in this book, we will meet models with branching structures. For example, we might take the values in a hidden layer and process them through two different sub-networks before recombining.

Fortunately, the ideas of backpropagation still hold if the computational graph is acyclic. Modern algorithmic differentiation frameworks such as PyTorch and TensorFlow can handle arbitrary acyclic computational graphs.

Problems 7.12–7.13

7.5 Parameter initialization

The backpropagation algorithm computes the derivatives that are used by stochastic gradient descent and Adam to train the model. We now address how to initialize the parameters before we start training. To see why this is crucial, consider that during the forward pass, each set of pre-activations $\mathbf{f}_k$ is computed as:

$$\begin{aligned}\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k \\ &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathrm{a}[\mathbf{f}_{k-1}],\end{aligned} \tag{7.26}$$

where $\mathbf{a}[\bullet]$ applies the ReLU functions and $\boldsymbol{\Omega}_k$ and $\boldsymbol{\beta}_k$ are the weights and biases, respectively. Imagine that we initialize all the biases to zero and the elements of $\boldsymbol{\Omega}_k$ according to a normal distribution with mean zero and variance σ^2. Consider two scenarios:

- If the variance σ^2 is very small (e.g., 10^{-5}), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very small; the result will likely have a smaller magnitude than the input. In addition, the ReLU function clips values less than zero, so the range of $\mathbf{h}_k$ will be half that of $\mathbf{f}_{k-1}$. Consequently, the magnitudes of the pre-activations at the hidden layers will get smaller and smaller as we progress through the network.
- If the variance σ^2 is very large (e.g., 10^5), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very large; the result is likely to have a much larger magnitude than the input. The ReLU function halves the range of the inputs, but if σ^2 is large enough, the magnitudes of the pre-activations will still get larger as we progress through the network.

In these two situations, the values at the pre-activations can become so small or so large that they cannot be represented with finite precision floating point arithmetic.

Even if the forward pass is tractable, the same logic applies to the backward pass. Each gradient update (equation 7.24) consists of multiplying by $\boldsymbol{\Omega}^T$. If the values of $\boldsymbol{\Omega}$ are not initialized sensibly, then the gradient magnitudes may decrease or increase uncontrollably during the backward pass. These cases are known as the *vanishing gradient problem* and the *exploding gradient problem*, respectively. In the former case, updates to the model become vanishingly small. In the latter case, they become unstable.

7.5.1 Initialization for forward pass

We now present a mathematical version of the same argument. Consider the computation between adjacent pre-activations $\mathbf{f}$ and $\mathbf{f}'$ with dimensions D_h and $D_{h'}$, respectively:

$$\begin{aligned}\mathbf{h} &= \mathbf{a}[\mathbf{f}], \\ \mathbf{f}' &= \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}\end{aligned} \tag{7.27}$$

where $\mathbf{f}$ represents the pre-activations, $\boldsymbol{\Omega}$, and $\boldsymbol{\beta}$ represent the weights and biases, and $\mathbf{a}[\bullet]$ is the activation function.

Assume the pre-activations f_j in the input layer $\mathbf{f}$ have variance σ_f^2. Consider initializing the biases β_i to zero and the weights Ω_{ij} as normally distributed with mean zero and variance σ_Ω^2. Now we derive expressions for the mean and variance of the pre-activations $\mathbf{f}'$ in the subsequent layer.

The expectation (mean) $\mathbb{E}[f_i']$ of the intermediate values f_i' is:

Appendix C.2 Expectation

$$\begin{aligned}\mathbb{E}[f_i'] &= \mathbb{E}\left[\beta_i + \sum_{j=1}^{D_h}\Omega_{ij}h_j\right] \\ &= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}h_j\right] \\ &= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}\right]\mathbb{E}\left[h_j\right] \\ &= 0 + \sum_{j=1}^{D_h} 0 \cdot \mathbb{E}\left[h_j\right] = 0, \end{aligned} \tag{7.28}$$

where D_h is the dimensionality of the input layer $\mathbf{h}$. We have used the rules for manipulating expectations, and we have assumed that the distributions over the hidden units h_j and the network weights Ω_{ij} are independent between the second and third lines.

Appendix C.2.1 Expectation rules

Using this result, we see that the variance $\sigma_{f'}^2$ of the pre-activations f_i' is:

$$\begin{aligned}\sigma_{f'}^2 &= \mathbb{E}[f_i'^2] - \mathbb{E}[f_i']^2 \\ &= \mathbb{E}\left[\left(\beta_i + \sum_{j=1}^{D_h}\Omega_{ij}h_j\right)^2\right] - 0 \\ &= \mathbb{E}\left[\left(\sum_{j=1}^{D_h}\Omega_{ij}h_j\right)^2\right] \\ &= \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}^2\right]\mathbb{E}\left[h_j^2\right] \\ &= \sum_{j=1}^{D_h}\sigma_\Omega^2\mathbb{E}\left[h_j^2\right] = \sigma_\Omega^2\sum_{j=1}^{D_h}\mathbb{E}\left[h_j^2\right], \end{aligned} \tag{7.29}$$

where we have used the variance identity $\sigma^2 = \mathbb{E}[(z-\mathbb{E}[z])^2] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$. We have assumed once more that the distributions of the weights Ω_{ij} and the hidden units h_j are independent between lines three and four.

Appendix C.2.3 Variance identity

Assuming that the input distribution of pre-activations f_j is symmetric about zero, half of these pre-activations will be clipped by the ReLU function, and the second moment $\mathbb{E}[h_j^2]$ will be half the variance σ_f^2 of f_j (see problem 7.14):

Problem 7.14

$$\sigma_{f'}^2 = \sigma_\Omega^2\sum_{j=1}^{D_h}\frac{\sigma_f^2}{2} = \frac{1}{2}D_h\sigma_\Omega^2\sigma_f^2. \tag{7.30}$$

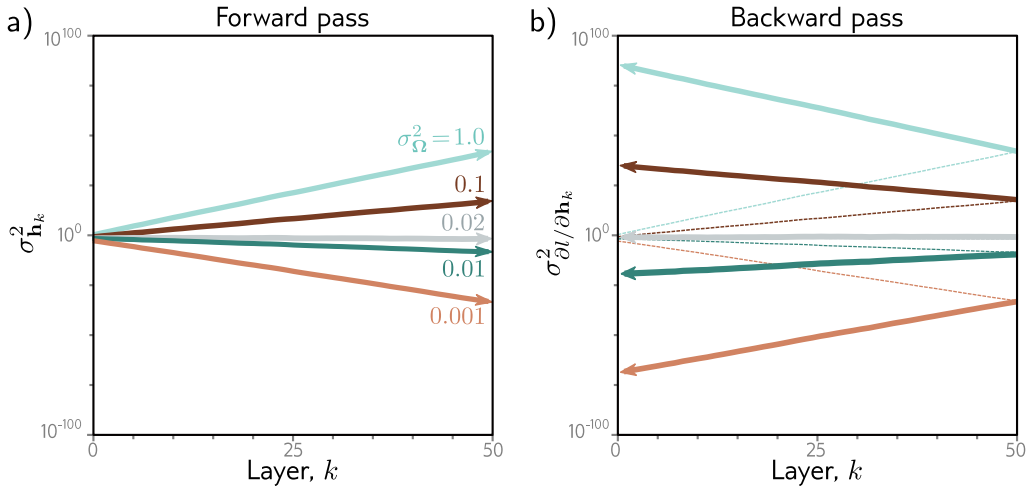


Figure 7.7 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100-dimensional input $\mathbf{x}$ initialized from a standard normal distribution, a single fixed target $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero, and the weight matrices $\boldsymbol{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_\Omega \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a) Variance of hidden unit activations computed in forward pass as a function of the network layer. For He initialization ($\sigma^2_\Omega = 2/D_h = 0.02$), the variance is stable. However, for larger values, it increases rapidly, and for smaller values, it decreases rapidly (note log scale). b) The variance of the gradients in the backward pass (solid lines) continues this trend; if we initialize with a value larger than 0.02, the magnitude of the gradients increases rapidly as we pass back through the network. If we initialize with a value smaller, then the magnitude decreases. These are known as the *exploding gradient* and *vanishing gradient* problems, respectively.

This, in turn, implies that if we want the variance $\sigma^2_{f'}$ of the subsequent pre-activations $\mathbf{f}'$ to be the same as the variance σ^2_f of the original pre-activations $\mathbf{f}$ during the forward pass, we should set:

$$\sigma^2_\Omega = \frac{2}{D_h}, \tag{7.31}$$

where D_h is the dimension of the original layer to which the weights were applied. This is known as *He initialization*.

7.5.2 Initialization for backward pass

A similar argument establishes how the variance of the gradients $\partial l/\partial f_k$ changes during the backward pass. During the backward pass, we multiply by the transpose $\boldsymbol{\Omega}^T$ of the weight matrix (equation 7.24), so the equivalent expression becomes:

$$\sigma_\Omega^2 = \frac{2}{D_{h'}}, \tag{7.32}$$

where $D_{h'}$ is the dimension of the layer that the weights feed into.

7.5.3 Initialization for both forward and backward pass

If the weight matrix $\boldsymbol{\Omega}$ is not square (i.e., there are different numbers of hidden units in the two adjacent layers, so D_h and $D_{h'}$ differ), then it is not possible to choose the variance to satisfy both equations 7.31 and 7.32 simultaneously. One possible compromise is to use the mean $(D_h + D_{h'})/2$ as a proxy for the number of terms, which gives:

$$\sigma_\Omega^2 = \frac{4}{D_h + D_{h'}}. \tag{7.33}$$

Figure 7.7 shows empirically that both the variance of the hidden units in the forward pass and the variance of the gradients in the backward pass remain stable when the parameters are initialized appropriately.

Problem 7.15

Notebook 7.3 Initialization

7.6 Example training code

The primary focus of this book is scientific; this is not a guide for implementing deep learning models. Nonetheless, in figure 7.8, we present PyTorch code that implements the ideas explored in this book so far. The code defines a neural network and initializes the weights. It creates random input and output datasets and defines a least squares loss function. The model is trained from the data using SGD with momentum in batches of size 10 over 100 epochs. The learning rate starts at 0.01 and halves every 10 epochs.

Problems 7.16–7.17

The takeaway is that although the underlying ideas in deep learning are quite complex, implementation is relatively simple. For example, all of the details of the backpropagation are hidden in the single line of code: `loss.backward()`.

7.7 Summary

The previous chapter introduced stochastic gradient descent (SGD), an iterative optimization algorithm that aims to find the minimum of a function. In the context of neural networks, this algorithm finds the parameters that minimize the loss function. SGD relies on the gradient of the loss function with respect to the parameters, which must be initialized before optimization. This chapter has addressed these two problems for deep neural networks.

The gradients must be evaluated for a very large number of parameters, for each member of the batch, and at each SGD iteration. It is hence imperative that the gradient

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
    nn.Linear(D_i, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
    if isinstance(layer_in, nn.Linear):
        nn.init.kaiming_normal_(layer_in.weight)
        layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 random data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

# loop over the dataset 100 times
for epoch in range(100):
    epoch_loss = 0.0
    # loop over batches
    for i, data in enumerate(data_loader):
        # retrieve inputs and labels for this batch
        x_batch, y_batch = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward pass
        pred = model(x_batch)
        loss = criterion(pred, y_batch)
        # backward pass
        loss.backward()
        # SGD update
        optimizer.step()
        # update statistics
        epoch_loss += loss.item()
    # print error
    print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')
    # tell scheduler to consider updating learning rate
    scheduler.step()
```

Figure 7.8 Sample code for training two-layer network on random data.

computation is efficient, and to this end, the backpropagation algorithm was introduced. Careful parameter initialization is also critical. The magnitudes of the hidden unit activations can either decrease or increase exponentially in the forward pass. The same is true of the gradient magnitudes in the backward pass, where these behaviors are known as the vanishing gradient and exploding gradient problems. Both impede training but can be avoided with appropriate initialization.

We've now defined the model and the loss function, and we can train a model for a given task. The next chapter discusses how to measure the model performance.

Notes

Backpropagation: Efficient reuse of partial computations while calculating gradients in computational graphs has been repeatedly discovered, including by Werbos (1974), Bryson et al. (1979), LeCun (1985), and Parker (1985). However, the most celebrated description of this idea was by Rumelhart et al. (1985) and Rumelhart et al. (1986), who also coined the term "backpropagation." This latter work kick-started a new phase of neural network research in the eighties and nineties; for the first time, it was practical to train networks with hidden layers. However, progress stalled due (in retrospect) to a lack of training data, limited computational power, and the use of sigmoid activations. Areas such as natural language processing and computer vision did not rely on neural network models until the remarkable image classification results of Krizhevsky et al. (2012) ushered in the modern era of deep learning.

The implementation of backpropagation in modern deep learning frameworks such as PyTorch and TensorFlow is an example of reverse-mode algorithmic differentiation. This is distinguished from forward-mode algorithmic differentiation in which the derivatives from the chain rule are accumulated while moving forward through the computational graph (see problem 7.13). Further information about algorithmic differentiation can be found in Griewank & Walther (2008) and Baydin et al. (2018).

Initialization: He initialization was first introduced by He et al. (2015). It follows closely from *Glorot* or *Xavier* initialization (Glorot & Bengio, 2010), which is very similar but does not consider the effect of the ReLU layer and so differs by a factor of two. Essentially the same method was proposed much earlier by LeCun et al. (2012) but with a slightly different motivation; in this case, sigmoidal activation functions were used, which naturally normalize the range of outputs at each layer, and hence help prevent an exponential increase in the magnitudes of the hidden units. However, if the pre-activations are too large, they fall into the flat regions of the sigmoid function and result in very small gradients. Hence, it is still important to initialize the weights sensibly. Klambauer et al. (2017) introduce the scaled exponential linear unit (SeLU) and show that, within a certain range of inputs, this activation function tends to make the activations in network layers automatically converge to mean zero and unit variance.

A completely different approach is to pass data through the network and then normalize by the empirically observed variance. *Layer-sequential unit variance initialization* (Mishkin & Matas, 2016) is an example of this kind of method, in which the weight matrices are initialized as orthonormal. GradInit (Zhu et al., 2021) randomizes the initial weights and temporarily fixes them while it learns non-negative scaling factors for each weight matrix. These factors are selected to maximize the decrease in the loss for a fixed learning rate subject to a constraint on the maximum gradient norm. *Activation normalization* or *ActNorm* adds a learnable scaling and offset parameter after each network layer at each hidden unit. They run an initial batch through the network and then choose the offset and scale so that the mean of the activations is zero and the variance one. After this, these extra parameters are learned as part of the model.

Closely related to these methods are schemes such as *BatchNorm* (Ioffe & Szegedy, 2015), in which the network normalizes the variance of each batch as part of its processing at every step. BatchNorm and its variants are discussed in chapter 11. Other initialization schemes have been proposed for specific architectures, including the *ConvolutionOrthogonal* initializer (Xiao et al., 2018a) for convolutional networks, *Fixup* (Zhang et al., 2019a) for residual networks, and *TFixup* (Huang et al., 2020a) and *DTFixup* (Xu et al., 2021b) for transformers.

Reducing memory requirements: Training neural networks is memory intensive. We must store both the model parameters and the pre-activations at the hidden units for every member of the batch during the forward pass. Two methods that decrease memory requirements are *gradient checkpointing* (Chen et al., 2016a) and *micro-batching* (Huang et al., 2019). In gradient checkpointing, the activations are only stored every N layers during the forward pass. During the backward pass, the intermediate missing activations are recalculated from the nearest checkpoint. In this manner, we can drastically reduce the memory requirements at the computational cost of performing the forward pass twice (problem 7.11). In micro-batching, the batch is subdivided into smaller parts, and the gradient updates are aggregated from each sub-batch before being applied to the network. A completely different approach is to build a reversible network (e.g., Gomez et al., 2017), in which the activations at the previous layer can be computed from the activations at the current one, so there is no need to cache anything during the forward pass (see chapter 16). Sohoni et al. (2019) review approaches to reducing memory requirements.

Distributed training: For sufficiently large models, the memory requirements or total required time may be too much for a single processor. In this case, we must use *distributed training*, in which training takes place in parallel across multiple processors. There are several approaches to parallelism. In *data parallelism*, each processor or *node* contains a full copy of the model but runs a subset of the batch (see Xing et al., 2015; Li et al., 2020b). The gradients from each node are aggregated centrally and then redistributed back to each node to ensure that the models remain consistent. This is known as *synchronous training.* The synchronization required to aggregate and redistribute the gradients can be a performance bottleneck, and this leads to the idea of asynchronous training. For example, in the *Hogwild!* algorithm (Recht et al., 2011), the gradient from a node is used to update a central model whenever it is ready. The updated model is then redistributed to the node. This means that each node may have a slightly different version of the model at any given time, so the gradient updates may be stale; however, it works well in practice. Other decentralized schemes have also been developed. For example, in Zhang et al. (2016a), the individual nodes update one another in a ring structure.

Data parallelism methods still assume that the entire model can be held in the memory of a single node. *Pipeline model parallelism* stores different layers of the network on different nodes and hence does not have this requirement. In a naïve implementation, the first node runs the forward pass for the batch on the first few layers and passes the result to the next node, which runs the forward pass on the next few layers and so on. In the backward pass, the gradients are updated in the opposite order. The obvious disadvantage of this approach is that each machine lies idle for most of the cycle. Various schemes revolving around each node processing micro-batches sequentially have been proposed to reduce this inefficiency (e.g., Huang et al., 2019; Narayanan et al., 2021a). Finally, in *tensor model parallelism*, computation at a single network layer is distributed across nodes (e.g., Shoeybi et al., 2019). A good overview of distributed training methods can be found in Narayanan et al. (2021b), who combine tensor, pipeline, and data parallelism to train a language model with one trillion parameters on 3072 GPUs.

Problems

Problem 7.1 A two-layer network with two hidden units in each layer can be defined as:

$$\begin{aligned} y &= \phi_0 + \phi_1 \mathrm{a}\Big[\psi_{01} + \psi_{11}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{21}\mathrm{a}[\theta_{02} + \theta_{12}x]\Big] \\ &\quad + \phi_2 \mathrm{a}\Big[\psi_{02} + \psi_{12}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{02} + \theta_{12}x]\Big], \end{aligned} \tag{7.34}$$

where the functions $a[\bullet]$ are ReLU functions. Compute the derivatives of the output y with respect to each of the 13 parameters $\boldsymbol{\phi}_\bullet$, $\theta_{\bullet\bullet}$, and $\psi_{\bullet\bullet}$ directly (i.e., not using the backpropagation algorithm). The derivative of the ReLU function with respect to its input $\partial a[z]/\partial z$ is the indicator function $\mathbb{I}[z > 0]$, which returns one if the argument is greater than zero and zero otherwise (figure 7.6).

Problem 7.2 Find an expression for the final term in each of the five chains of derivatives in equation 7.12.

Problem 7.3 What size are each of the terms in equation 7.19?

Problem 7.4 Calculate the derivative $\partial \ell_i / \partial f[\mathbf{x}_i, \boldsymbol{\phi}]$ for the least squares loss function:

$$\ell_i = (y_i - \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2. \tag{7.35}$$

Problem 7.5 Calculate the derivative $\partial \ell_i / \partial \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for the binary classification loss function:

$$\ell_i = -(1 - y_i)\log\Big[1 - \mathrm{sig}\big[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]\big]\Big] - y_i \log\Big[\mathrm{sig}\big[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]\big]\Big], \tag{7.36}$$

where the function $\mathrm{sig}[\bullet]$ is the logistic sigmoid and is defined as:

$$\mathrm{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{7.37}$$

Problem 7.6* Show that for $\mathbf{z} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{h}} = \boldsymbol{\Omega}^T,$$

where $\partial \mathbf{z}/\partial \mathbf{h}$ is a matrix containing the term $\partial z_i / \partial h_j$ in its i^{th} column and j^{th} row. To do this, first find an expression for the constituent elements $\partial z_i / \partial h_j$, and then consider the form that the matrix $\partial \mathbf{z}/\partial \mathbf{h}$ must take.

Problem 7.7 Consider the case where we use the logistic sigmoid (see equation 7.37) as an activation function, so $h = \mathrm{sig}[f]$. Compute the derivative $\partial h / \partial f$ for this activation function. What happens to the derivative when the input takes (i) a large positive value and (ii) a large negative value?

Problem 7.8 Consider using (i) the Heaviside function and (ii) the rectangular function as activation functions:

$$\mathrm{Heaviside}[z] = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}, \tag{7.38}$$

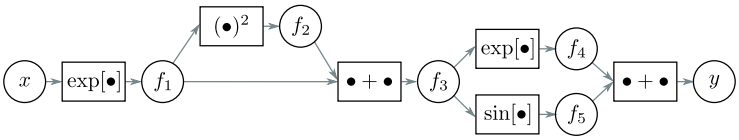


Figure 7.9 Computational graph for problem 7.12 and problem 7.13. Adapted from Domke (2010).

and

$$\text{rect}[z] = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}. \tag{7.39}$$

Discuss why these functions are problematic for neural network training with gradient-based optimization methods.

Problem 7.9* Consider a loss function $\ell[\mathbf{f}]$, where $\mathbf{f} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$. We want to find how the loss ℓ changes when we change $\boldsymbol{\Omega}$, which we'll express with a matrix that contains the derivative $\partial\ell/\partial\Omega_{ij}$ at the i^{th} row and j^{th} column. Find an expression for $\partial f_i/\partial\Omega_{ij}$ and, using the chain rule, show that:

$$\frac{\partial \ell}{\partial \boldsymbol{\Omega}} = \frac{\partial \ell}{\partial \mathbf{f}}\mathbf{h}^T. \tag{7.40}$$

Problem 7.10* Derive the equations for the backward pass of the backpropagation algorithm for a network that uses leaky ReLU activations, which are defined as:

$$\text{a}[z] = \text{ReLU}[z] = \begin{cases} \alpha \cdot z & z < 0 \\ z & z \geq 0 \end{cases}, \tag{7.41}$$

where α is a small positive constant (typically 0.1).

Problem 7.11 Consider training a network with fifty layers, where we only have enough memory to store the pre-activations at every tenth hidden layer during the forward pass. Explain how to compute the derivatives in this situation using gradient checkpointing.

Problem 7.12* This problem explores computing derivatives on general acyclic computational graphs. Consider the function:

$$y = \exp\left[\exp[x] + \exp[x]^2\right] + \sin[\exp[x] + \exp[x]^2]. \tag{7.42}$$

We can break this down into a series of intermediate computations so that:

$$\begin{aligned} f_1 &= \exp[x] \\ f_2 &= f_1^2 \\ f_3 &= f_1 + f_2 \\ f_4 &= \exp[f_3] \\ f_5 &= \sin[f_3] \\ y &= f_4 + f_5. \end{aligned} \tag{7.43}$$

The associated computational graph is depicted in figure 7.9. Compute the derivative $\partial y/\partial x$ by *reverse-mode differentiation*. In other words, compute in order:

$$\frac{\partial y}{\partial f_5}, \frac{\partial y}{\partial f_4}, \frac{\partial y}{\partial f_3}, \frac{\partial y}{\partial f_2}, \frac{\partial y}{\partial f_1} \text{ and } \frac{\partial y}{\partial x}, \tag{7.44}$$

using the chain rule in each case to make use of the derivatives already computed.

Problem 7.13* For the same function as in problem 7.12, compute the derivative $\partial y/\partial x$ by *forward-mode differentiation*. In other words, compute in order:

$$\frac{\partial f_1}{\partial x}, \frac{\partial f_2}{\partial x}, \frac{\partial f_3}{\partial x}, \frac{\partial f_4}{\partial x}, \frac{\partial f_5}{\partial x}, \text{ and } \frac{\partial y}{\partial x}, \tag{7.45}$$

using the chain rule in each case to make use of the derivatives already computed. Why do we not use forward-mode differentiation when we calculate the parameter gradients for deep networks?

Problem 7.14 Consider a random variable a with variance $\text{Var}[a] = \sigma^2$ and a symmetrical distribution around the mean $\mathbb{E}[a] = 0$. Prove that if we pass this variable through the ReLU function:

$$b = \text{ReLU}[a] = \begin{cases} 0 & a < 0 \\ a & a \geq 0 \end{cases}, \tag{7.46}$$

then the second moment of the transformed variable is $\mathbb{E}[b^2] = \sigma^2/2$.

Problem 7.15 What would you expect to happen if we initialized all of the weights and biases in the network to zero?

Problem 7.16 Implement the code in figure 7.8 in PyTorch and plot the training loss as a function of the number of epochs.

Problem 7.17 Change the code in figure 7.8 to tackle a binary classification problem. You will need to (i) change the targets y so they are binary, (ii) change the network to predict numbers between zero and one (iii) change the loss function appropriately.

Chapter 8

Measuring performance

Previous chapters described neural network models, loss functions, and training algorithms. This chapter considers how to measure the performance of the trained models. With sufficient capacity (i.e., number of hidden units), a neural network model will often perform perfectly on the training data. However, this does not necessarily mean it will generalize well to new test data.

We will see that the test errors have three distinct causes and that their relative contributions depend on (i) the inherent uncertainty in the task, (ii) the amount of training data, and (iii) the choice of model. The latter dependency raises the issue of hyperparameter search. We discuss how to select both the model hyperparameters (e.g., the number of hidden layers and the number of hidden units in each) and the learning algorithm hyperparameters (e.g., the learning rate and batch size).

8.1 Training a simple model

We explore model performance using the MNIST-1D dataset (figure 8.1). This consists of ten classes $y \in \{0, 1, \ldots, 9\}$, representing the digits 0–9. The data are derived from 1D templates for each of the digits. Each data example $\mathbf{x}$ is created by randomly transforming one of these templates and adding noise. The full training dataset $\{\mathbf{x}_i, y_i\}$ consists of $I = 4000$ training examples, each consisting of $D_i = 40$ dimensions representing the horizontal offset at 40 positions. The ten classes are drawn uniformly during data generation, so there are $\sim$400 examples of each class.

We use a network with $D_i = 40$ inputs and $D_o = 10$ outputs which are passed through a softmax function to produce class probabilities (see section 5.5). The network has two hidden layers with $D = 100$ hidden units each. It is trained using stochastic gradient descent with batch size 100 and learning rate 0.1 for 6000 steps (150 epochs) with a multiclass cross-entropy loss (equation 5.24). Figure 8.2 shows that the training error decreases as training proceeds. The training data are classified perfectly after about 4000 steps. The training loss also decreases, eventually approaching zero.

Problem 8.1

However, this doesn't imply that the classifier is perfect; the model might have mem-

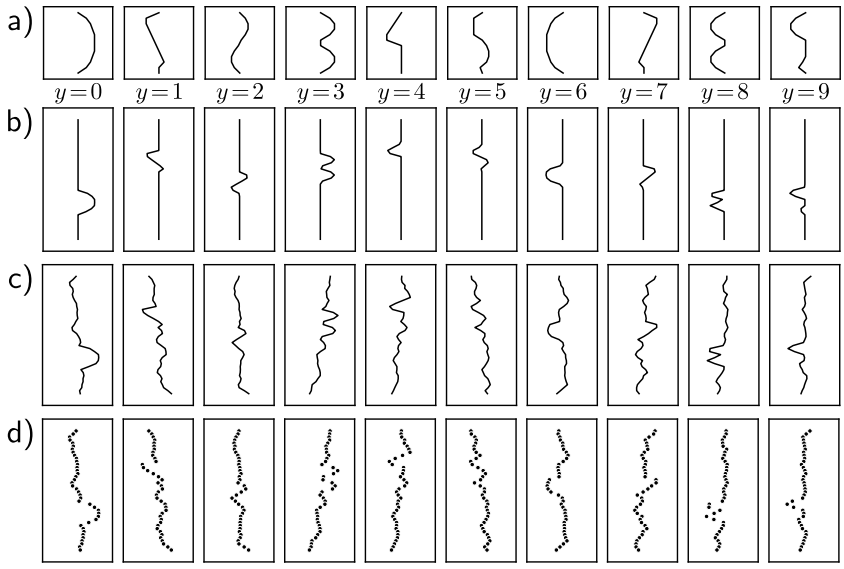


Figure 8.1 MNIST-1D. a) Templates for 10 classes $y \in \{0, \ldots, 9\}$, based on digits 0–9. b) Training examples $\mathbf{x}$ are created by randomly transforming a template and c) adding noise. d) The horizontal offset of the transformed template is then sampled at 40 vertical positions. Adapted from (Greydanus, 2020)

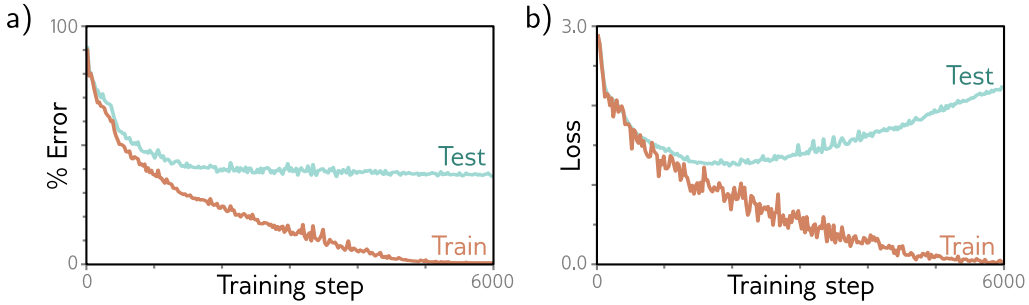


Figure 8.2 MNIST-1D results. a) Percent classification error as a function of the training step. The training set errors decrease to zero, but the test errors do not drop below $\sim 40\%$. This model doesn't generalize well to new test data. b) Loss as a function of the training step. The training loss decreases steadily toward zero. The test loss decreases at first but subsequently increases as the model becomes increasingly confident about its (wrong) predictions.

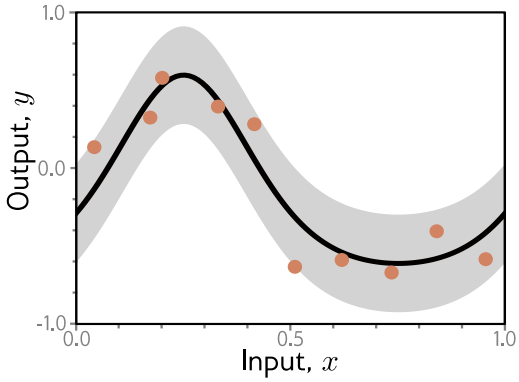


Figure 8.3 Regression function. Solid black line shows ground truth function. To generate I training examples $\{x_i, y_i\}$, the input space $x \in [0, 1]$ is divided into I equal segments and one sample x_i is drawn from a uniform distribution within each segment. The corresponding value y_i is created by evaluating the function at x_i and adding Gaussian noise (gray region shows ± 2 standard deviations). The test data are generated in the same way.

orized the training set but be unable to predict new examples. To estimate the true performance, we need a separate *test set* of input/output pairs $\{\mathbf{x}_i, y_i\}$. To this end, we generate 1000 more examples using the same process. Figure 8.2a also shows the errors for this test data as a function of the training step. These decrease as training proceeds, but only to around 40%. This is better than the chance error rate of 90% error rate but far worse than for the training set; the model has not *generalized* well to the test data.

Notebook 8.1 MNIST-1D performance

The test loss (figure 8.2b) decreases for the first 1500 training steps but then increases again. At this point, the test error rate is fairly constant; the model makes the same mistakes but with increasing confidence. This decreases the probability of the correct answers and thus increases the negative log-likelihood. This increasing confidence is a side-effect of the softmax function; the pre-softmax activations are driven to increasingly extreme values to make the probability of the training data approach one (see figure 5.10).

8.2 Sources of error

We now consider the sources of the errors that occur when a model fails to generalize. To make this easier to visualize, we revert to a 1D linear least squares regression problem where we know exactly how the ground truth data were generated. Figure 8.3 shows a quasi-sinusoidal function; both training and test data are generated by sampling input values in the range $[0, 1]$, passing them through this function, and adding Gaussian noise with a fixed variance.

We fit a simplified shallow neural net to this data (figure 8.4). The weights and biases that connect the input layer to the hidden layer are chosen so that the "joints" of the function are evenly spaced across the interval. If there are D hidden units, then these joints will be at $0, 1/D, 2/D, \ldots, (D-1)/D$. This model can represent any piecewise linear function with D equally sized regions in the range $[0, 1]$. As well as being easy to understand, this model also has the advantage that it can be fit in closed form without the need for stochastic optimization algorithms (see problem 8.3). Consequently, we can guarantee to find the global minimum of the loss function during training.

Problems 8.2–8.3

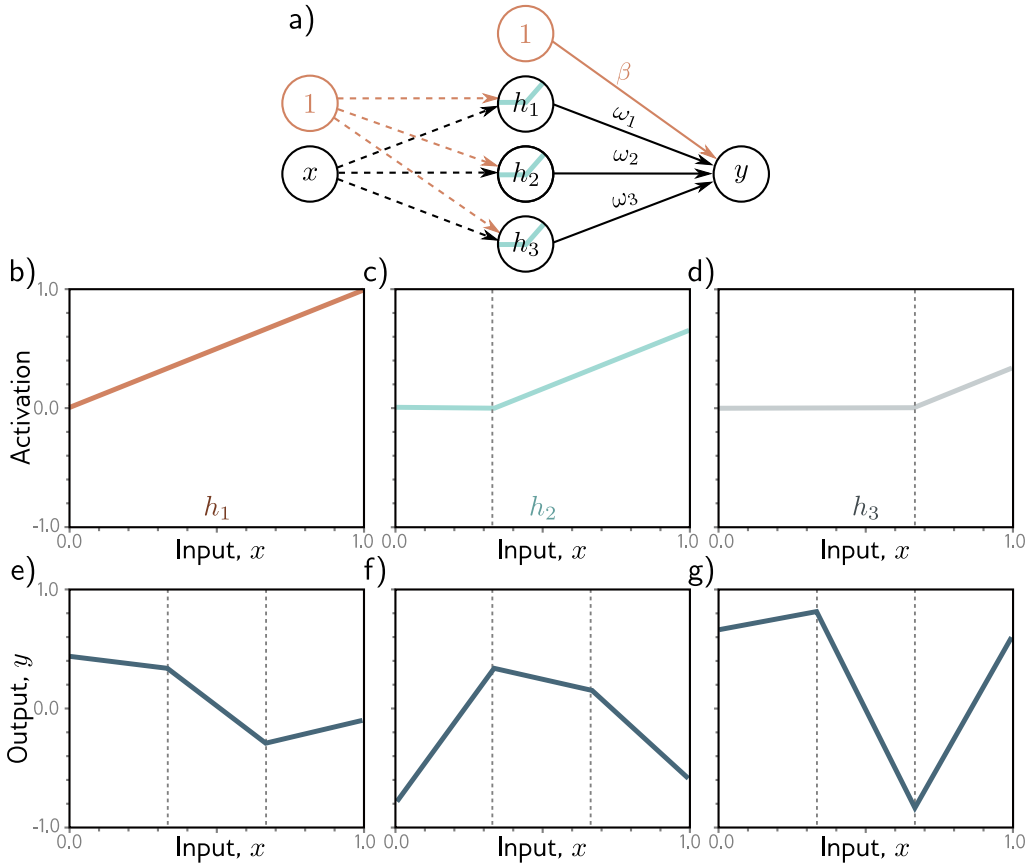


Figure 8.4 Simplified neural network with three hidden units. a) The weights and biases between the input and hidden layer are fixed (dashed arrows). b–d) They are chosen so that the hidden unit activations have slope one, and their joints are equally spaced across the interval, with joints at $x = 0$, $x = 1/3$, and $x = 2/3$, respectively. Modifying the remaining parameters $\boldsymbol{\phi} = \{\beta, \omega_1, \omega_2, \omega_3\}$ can create any piecewise linear function over $x \in [0, 1]$ with joints at $1/3$ and $2/3$. e–g) Three example functions with different values of the parameters $\boldsymbol{\phi}$.

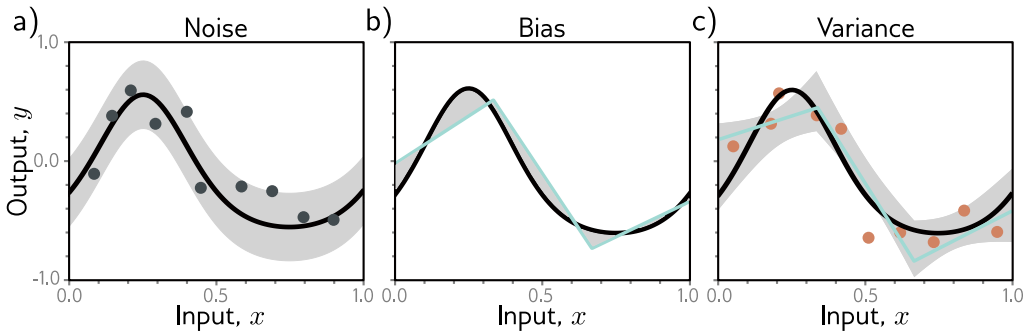


Figure 8.5 Sources of test error. a) Noise. Data generation is noisy, so even if the model exactly replicates the true underlying function (black line), the noise in the test data (gray points) means that some error will remain (gray region represents two standard deviations). b) Bias. Even with the best possible parameters, the three-region model (cyan line) cannot exactly fit the true function (black line). This bias is another source of error (gray regions represent signed error). c) Variance. In practice, we have limited noisy training data (orange points). When we fit the model, we don't recover the best possible function from panel (b) but a slightly different function (cyan line) that reflects idiosyncrasies of the training data. This provides an additional source of error (gray region represents two standard deviations). Figure 8.6 shows how this region was calculated.

8.2.1 Noise, bias, and variance

There are three possible sources of error, which are known as *noise*, *bias*, and *variance* respectively (figure 8.5):

Noise The data generation process includes the addition of noise, so there are multiple possible valid outputs y for each input x (figure 8.5a). This source of error is insurmountable for the test data. Note that it does not necessarily limit the training performance; we will likely never see the same input x twice during training, so it is still possible to fit the training data perfectly.

Noise may arise because there is a genuine stochastic element to the data generation process, because some of the data are mislabeled, or because there are further explanatory variables that were not observed. In rare cases, noise may be absent; for example, a network might approximate a function that is deterministic but requires significant computation to evaluate. However, noise is usually a fundamental limitation on the possible test performance.

Bias A second potential source of error may occur because the model is not flexible enough to fit the true function perfectly. For example, the three-region neural network model cannot exactly describe the quasi-sinusoidal function, even when the parameters are chosen optimally (figure 8.5b). This is known as *bias*.

Variance We have limited training examples, and there is no way to distinguish systematic changes in the underlying function from noise in the underlying data. When we fit a model, we do not get the closest possible approximation to the true underlying function. Indeed, for different training datasets, the result will be slightly different each time. This additional source of variability in the fitted function is termed *variance* (figure 8.5c). In practice, there might also be additional variance due to the stochastic learning algorithm, which does not necessarily converge to the same solution each time.

8.2.2 Mathematical formulation of test error

We now make the notions of noise, bias, and variance mathematically precise. Consider a 1D regression problem where the data generation process has additive noise with variance σ^2 (e.g., figure 8.3); we can observe different outputs y for the same input x, so for each x, there is a distribution $Pr(y|x)$ with expected value (mean) $\mu[x]$:

Appendix C.2 Expectation

$$\mu[x] = \mathbb{E}_y[y[x]] = \int y[x] Pr(y|x) dy, \tag{8.1}$$

and fixed noise $\sigma^2 = \mathbb{E}_y\left[(\mu[x] - y[x])^2\right]$. Here we have used the notation $y[x]$ to specify that we are considering the output y at a given input position x.

Now consider a least squares loss between the model prediction $\mathrm{f}[x, \boldsymbol{\phi}]$ at position x and the observed value $y[x]$ at that position:

$$\begin{aligned} L[x] &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - y[x]\big)^2 \\ &= \Big(\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big) + \big(\mu[x] - y[x]\big)\Big)^2 \\ &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)\big(\mu[x] - y[x]\big) + \big(\mu[x] - y[x]\big)^2, \end{aligned} \tag{8.2}$$

where we have both added and subtracted the mean $\mu[x]$ of the underlying function in the second line and have expanded out the squared term in the third line.

The underlying function is stochastic, so this loss depends on the particular $y[x]$ we observe. The expected loss is:

$$\begin{aligned} \mathbb{E}_y\big[L[x]\big] &= \mathbb{E}_y\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)\big(\mu[x] - y[x]\big) + \big(\mu[x] - y[x]\big)^2\Big] \\ &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)\big(\mu[x] - \mathbb{E}_y[y[x]]\big) + \mathbb{E}_y\left[(\mu[x] - y[x])^2\right] \\ &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big) \cdot 0 + \mathbb{E}_y\Big[\big(\mu[x] - y[x]\big)^2\Big] \\ &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + \sigma^2, \end{aligned} \tag{8.3}$$

where we have made use of the rules for manipulating expectations. In the second line, we have distributed the expectation operator and removed it from terms with no dependence on $y[x]$, and in the third line, we note that the second term is zero since $\mathbb{E}_y[y[x]] = \mu[x]$ by definition. Finally, in the fourth line, we have substituted in the definition of the

Appendix C.2.1 Expectation rules

noise σ^2. We can see that the expected loss has been broken down into two terms; the first term is the squared deviation between the model and the true function mean, and the second term is the noise.

The first term can be further partitioned into bias and variance. The parameters $\boldsymbol{\phi}$ of the model $\mathrm{f}[x, \boldsymbol{\phi}]$ depend on the training dataset $\mathcal{D} = \{x_i, y_i\}$, so more properly, we should write $\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]]$. The training dataset is a random sample from the data generation process; with a different sample of training data, we would learn different parameter values. The expected model output $\mathrm{f}_\mu[x]$ with respect to all possible datasets $\mathcal{D}$ is hence:

$$\mathrm{f}_\mu[x] = \mathbb{E}_{\mathcal{D}}\Big[\mathrm{f}\big[x, \boldsymbol{\phi}[\mathcal{D}]\big]\Big]. \tag{8.4}$$

Returning to the first term of equation 8.3, we add and subtract $\mathrm{f}_\mu[x]$ and expand:

$$\begin{aligned}
&\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mu[x]\big)^2 \qquad (8.5)\\
&\quad = \Big(\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mathrm{f}_\mu[x]\big) + \big(\mathrm{f}_\mu[x] - \mu[x]\big)\Big)^2 \\
&\quad = \big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mathrm{f}_\mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mathrm{f}_\mu[x]\big)\big(\mathrm{f}_\mu[x] - \mu[x]\big) + \big(\mathrm{f}_\mu[x] - \mu[x]\big)^2.
\end{aligned}$$

We then take the expectation with respect to the training dataset $\mathcal{D}$:

$$\mathbb{E}_{\mathcal{D}}\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mu[x]\big)^2\Big] = \mathbb{E}_{\mathcal{D}}\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mathrm{f}_\mu[x]\big)^2\Big] + \big(\mathrm{f}_\mu[x] - \mu[x]\big)^2, \tag{8.6}$$

where we have simplified using similar steps as for equation 8.3. Finally, we substitute this result into equation 8.3:

$$\mathbb{E}_{\mathcal{D}}\Big[\mathbb{E}_y[L[x]]\Big] = \underbrace{\mathbb{E}_{\mathcal{D}}\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mathrm{f}_\mu[x]\big)^2\Big]}_{\text{variance}} + \underbrace{\big(\mathrm{f}_\mu[x] - \mu[x]\big)^2}_{\text{bias}} + \underbrace{\sigma^2}_{\text{noise}}. \tag{8.7}$$

This equation says that the expected loss after considering the uncertainty in the training data $\mathcal{D}$ and the test data y consists of three additive components. The variance is uncertainty in the fitted model due to the particular training dataset we sample. The bias is the systematic deviation of the model from the mean of the function we are modeling. The noise is the inherent uncertainty in the true mapping from input to output. These three sources of error will be present for any task. They combine additively for linear regression with a least squares loss. However, their interaction can be more complex for other types of problems.

8.3 Reducing error

In the previous section, we saw that test error results from three sources: noise, bias, and variance. The noise component is insurmountable; there is nothing we can do to circumvent this, and it represents a fundamental limit on model performance. However, it is possible to reduce the other two terms.

8.3.1 Reducing variance

Recall that the variance results from limited noisy training data. Fitting the model to two different training sets results in slightly different parameters. It follows we can reduce the variance by increasing the quantity of training data. This averages out the inherent noise and ensures that the input space is well sampled.

Figure 8.6 shows the effect of training with 6, 10, and 100 samples. For each dataset size, we show the best-fitting model for three training datasets. With only six samples, the fitted function is quite different each time: the variance is significant. As we increase the number of samples, the fitted models become very similar, and the variance reduces. In general, adding training data almost always improves test performance.

8.3.2 Reducing bias

The bias term results from the inability of the model to describe the true underlying function. This suggests that we can reduce this error by making the model more flexible. This is usually done by increasing the model *capacity*. For neural networks, this means adding more hidden units and/or hidden layers.

In the simplified model, adding capacity corresponds to adding more hidden units so that the interval $[0, 1]$ is divided into more linear regions. Figures 8.7a–c show that (unsurprisingly) this does indeed reduce the bias; as we increase the number of linear regions to ten, the model becomes flexible enough to fit the true function closely.

8.3.3 Bias-variance trade-off

However, figures 8.7d–f show an unexpected side-effect of increasing the model capacity. For a fixed-size training dataset, the variance term increases as the model capacity increases. Consequently, increasing the model capacity does not necessarily reduce the test error. This is known as the *bias-variance trade-off*.

Figure 8.8 explores this phenomenon. In panels a–c), we fit the simplified three-region model to three different datasets of fifteen points. Although the datasets differ, the final model is much the same; the noise in the dataset roughly averages out in each linear region. In panels d–f), we fit a model with ten regions to the same three datasets. This model has more flexibility, but this is disadvantageous; the model certainly fits the data better, and the training error will be lower, but much of the extra descriptive power is devoted to modeling the noise. This phenomenon is known as *overfitting*.

We've seen that as we add capacity to the model, the bias decreases, but the variance increases for a fixed-size training dataset. This suggests that there is an optimal capacity where the bias is not too large and the variance is still relatively small. Figure 8.9 shows how these terms vary numerically for the toy model as we increase the capacity, using the data from figure 8.8. For regression models, the total expected error is the sum of the bias and the variance, and this sum is minimized when the model capacity is four (i.e., with four hidden units and four linear regions in the range of the data).

Notebook 8.2
Bias-variance
trade-off

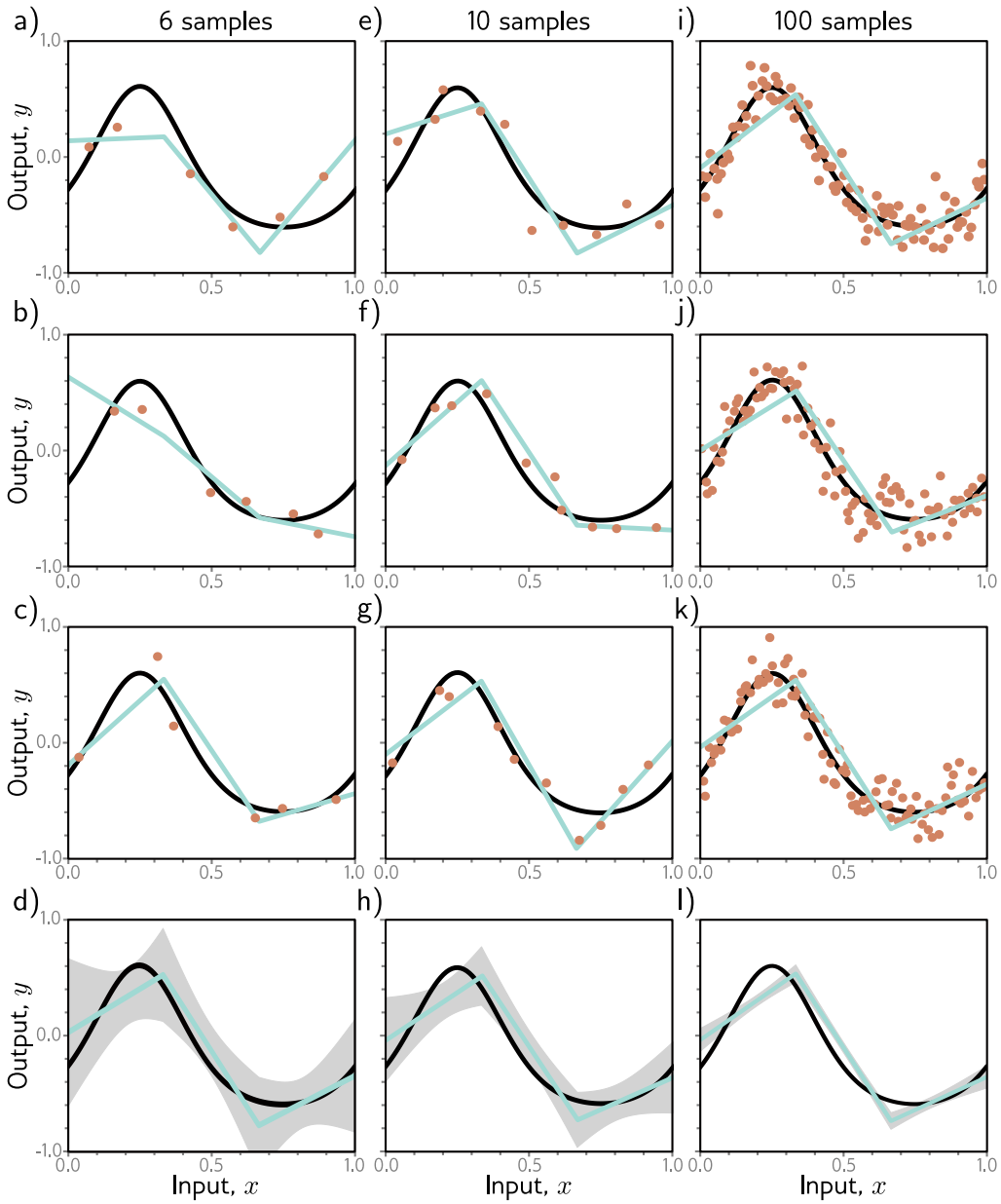


Figure 8.6 Reducing variance by increasing training data. a–c) The three-region model fitted to three different randomly sampled datasets of six points. The fitted model is quite different each time. d) We repeat this experiment many times and plot the mean model predictions (cyan line) and the variance of the model predictions (gray area shows two standard deviations). e–h) We do the same experiment, but this time with datasets of size ten. The variance of the predictions is reduced. i–l) We repeat this experiment with datasets of size 100. Now the fitted model is always similar, and the variance is small.

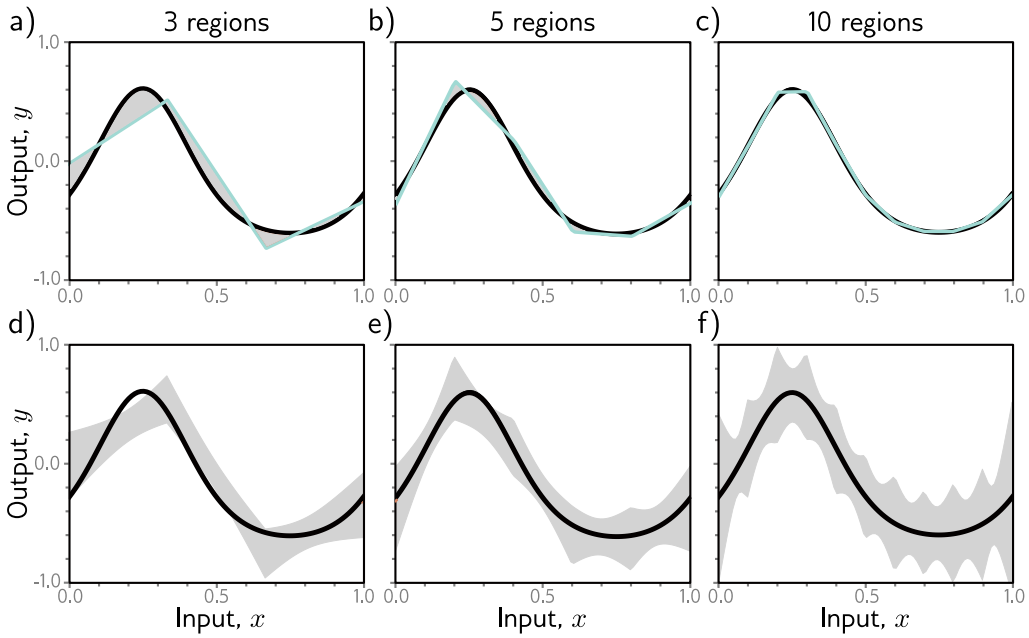


Figure 8.7 Bias and variance as a function of model capacity. a–c) As we increase the number of hidden units of the toy model, the number of linear regions increases, and the model becomes able to fit the true function closely; the bias (gray region) decreases. d–f) Unfortunately, increasing the model capacity has the side-effect of increasing the variance term (gray region). This is known as the bias-variance trade-off.

8.4 Double descent

In the previous section, we examined the bias-variance trade-off as we increased the capacity of a model. Let's now return to the MNIST-1D dataset and see whether this happens in practice. We use 10,000 training examples, test with another 5,000 examples and examine the training and test performance as we increase the capacity (number of parameters) in the model. We train the model with Adam and a step size of 0.005 using a full batch of 10,000 examples for 4000 steps.

Figure 8.10a shows the training and test error for a neural network with two hidden layers as the number of hidden units increases. The training error decreases as the capacity grows and quickly becomes close to zero. The vertical dashed line represents the capacity where the model has the same number of parameters as there are training examples, but the model memorizes the dataset before this point. The test error decreases as we add model capacity but does not increase as predicted by the bias-variance trade-off curve; it keeps decreasing.

In figure 8.10b, we repeat this experiment, but this time, we randomize 15% of the

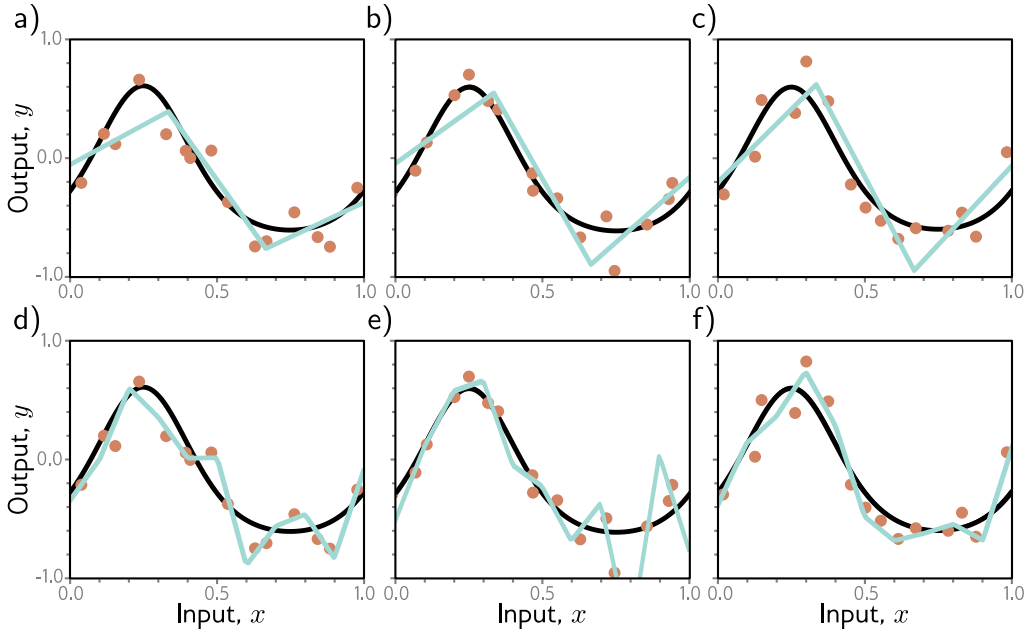
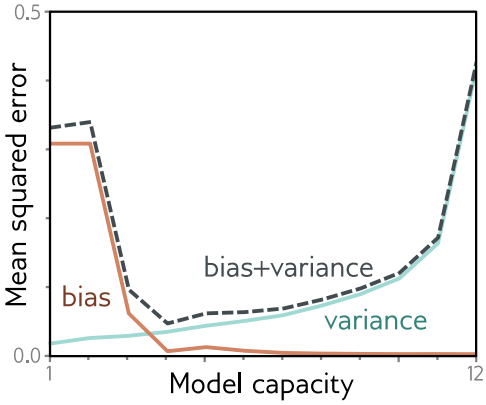


Figure 8.8 Overfitting. a–c) A model with three regions is fit to three different datasets of fifteen points each. The result is similar in all three cases (i.e., the variance is low). d–f) A model with ten regions is fit to the same datasets. The additional flexibility does not necessarily produce better predictions. While these three models each describe the training data better, they are not necessarily closer to the true underlying function (black curve). Instead, they overfit the data and describe the noise, and the variance (difference between fitted curves) is larger.

Figure 8.9 Bias-variance trade-off. The bias and variance terms from equation 8.7 are plotted as a function of the model capacity (number of hidden units / linear regions in range of data) in the simplified model using training data from figure 8.8. As the capacity increases, the bias (solid orange line) decreases, but the variance (solid cyan line) increases. The sum of these two terms (dashed gray line) is minimized when the capacity is four.



training labels. Once more, the training error decreases to zero. This time, there is more randomness, and the model requires almost as many parameters as there are data points to memorize the data. The test error does show the typical bias-variance trade-off as we increase the capacity to the point where the model fits the training data exactly. However, then it does something unexpected; it starts to decrease again. Indeed, if we add enough capacity, the test loss reduces to below the minimal level that we achieved in the first part of the curve.

This phenomenon is known as *double descent*. For some datasets like MNIST, it is present with the original data (figure 8.10c). For others, like MNIST-1D and CIFAR-100 (figure 8.10d), it emerges or becomes more prominent when we add noise to the labels. The first part of the curve is referred to as the *classical* or *under-parameterized regime*, and the second part as the *modern* or *over-parameterized regime*. The central part where the error increases is termed the *critical regime*.

Notebook 8.3
Double descent

8.4.1 Explanation

The discovery of double descent is recent, unexpected, and somewhat puzzling. It results from an interaction of two phenomena. First, the test performance becomes temporarily worse when the model has just enough capacity to memorize the data. Second, the test performance continues to improve with capacity even after the training performance is perfect. The first phenomenon is exactly as predicted by the bias-variance trade-off. The second phenomenon is more confusing; it's unclear why performance should be better in the over-parameterized regime, given that there are now not even enough training data points to constrain the model parameters uniquely.

To understand why performance continues to improve as we add more parameters, note that once the model has enough capacity to drive the training loss to near zero, the model fits the training data almost perfectly. This implies that further capacity cannot help the model fit the training data any better; any change must occur *between* the training points. The tendency of a model to prioritize one solution over another as it extrapolates between data points is known as its *inductive bias*.

Problems 8.4–8.5

The model's behavior between data points is critical because, in high-dimensional space, the training data are extremely sparse. The MNIST-1D dataset has 40 dimensions, and we trained with 10,000 examples. If this seems like plenty of data, consider what would happen if we quantized each input dimension into 10 bins. There would be 10^{40} bins in total, constrained by only 10^4 examples. Even with this coarse quantization, there will only be one data point in every 10^{35} bins! The tendency of the volume of high-dimensional space to overwhelm the number of training points is termed the *curse of dimensionality*.

The implication is that problems in high dimensions might look more like figure 8.11a; there are small regions of the input space where we observe data with significant gaps between them. The putative explanation for double descent is that as we add capacity to the model, it interpolates between the nearest data points increasingly smoothly. In the absence of information about what happens between the training points, assuming smoothness is sensible and will probably generalize reasonably to new data.

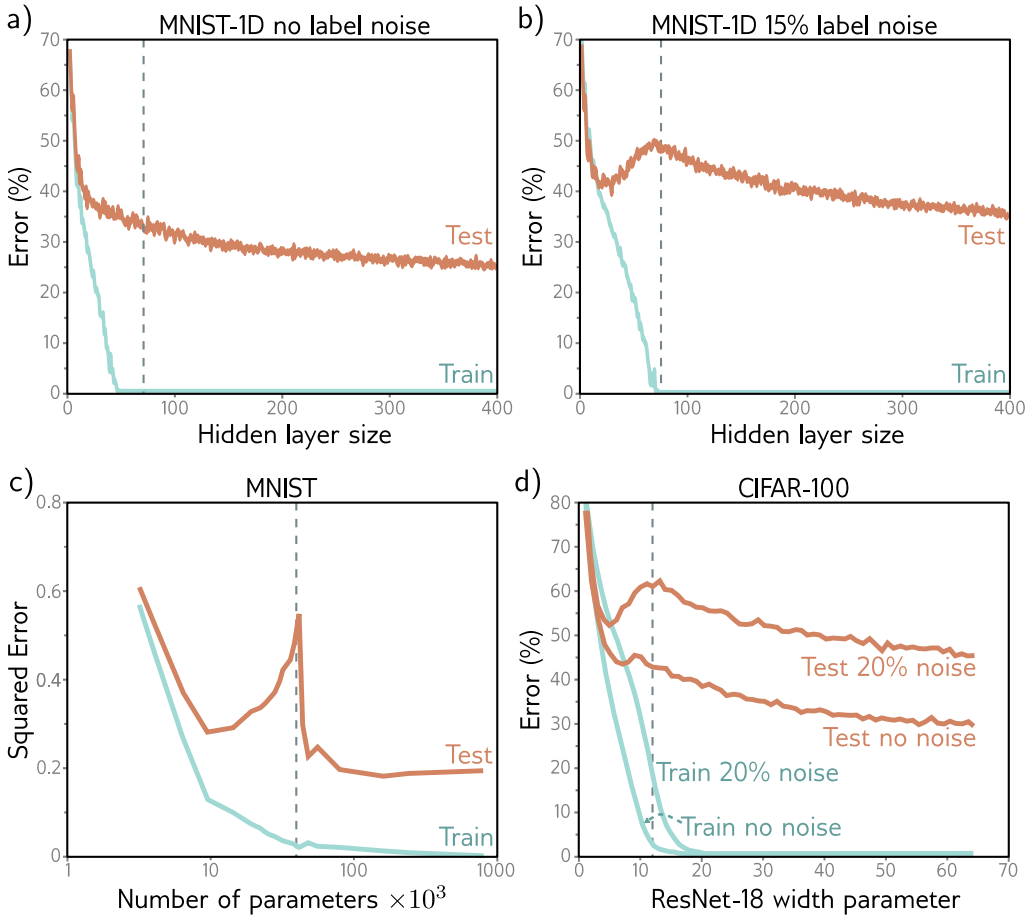


Figure 8.10 Double descent. a) Training and test loss on MNIST-1D for a two-hidden layer network as we increase the number of hidden units (and hence parameters) in each layer. The training loss decreases to zero as the number of parameters approaches the number of training examples (vertical dashed line). The test error does not show the expected bias-variance trade-off but continues to decrease even after the model has memorized the dataset. b) The same experiment is repeated with noisier training data. Again, the training error reduces to zero, although it now takes almost as many parameters as training points to memorize the dataset. The test error shows the predicted bias/variance trade-off; it decreases as the capacity increases but then increases again as we near the point where the training data is exactly memorized. However, it subsequently decreases again and ultimately reaches a better performance level. This is known as double descent. Depending on the loss, the model, and the amount of noise in the data, the double descent pattern can be seen to a greater or lesser degree across many datasets. c) Results on MNIST (without label noise) with shallow neural network from Belkin et al. (2019). d) Results on CIFAR-100 with ResNet18 network (see chapter 11) from Nakkiran et al. (2021). See original papers for details.

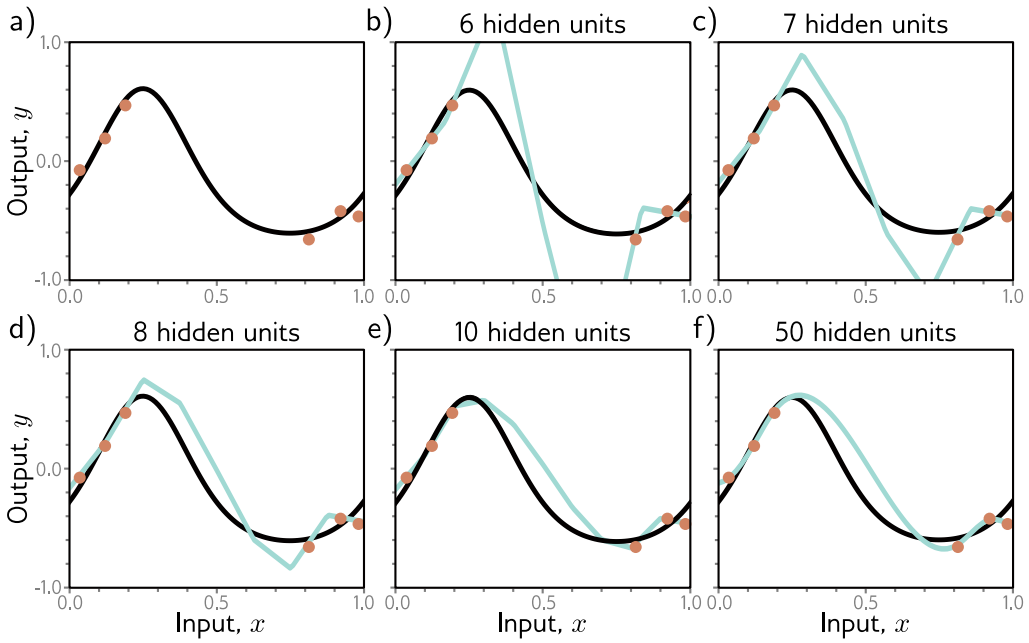


Figure 8.11 Increasing capacity (hidden units) allows smoother interpolation between sparse data points. a) Consider this situation where the training data (orange circles) are sparse; there is a large region in the center with no data examples to constrain the model to mimic the true function (black curve). b) If we fit a model with just enough capacity to fit the training data (cyan curve), then it has to contort itself to pass through the training data, and the output predictions will not be smooth. c–f) However, as we add more hidden units, the model has the *ability* to interpolate between the points more smoothly (smoothest possible curve plotted in each case). However, unlike in this figure, it is not obliged to.

This argument is plausible. It's certainly true that as we add more capacity to the model, it will have the capability to create smoother functions. Figures 8.11b–f show the smoothest possible functions that still pass through the data points as we increase the number of hidden units. When the number of parameters is very close to the number of training data examples (figure 8.11b), the model is forced to contort itself to fit the training data exactly, resulting in erratic predictions. This explains why the peak in the double descent curve is so pronounced. As we add more hidden units, the model has the ability to construct smoother functions that are likely to generalize better to new data.

However, this does not explain *why* over-parameterized models should produce smooth functions. Figure 8.12 shows three functions that can be created by the simplified model with 50 hidden units. In each case, the model fits the data exactly, so the loss is zero. If the modern regime of double descent is explained by increasing smoothness, then what exactly is encouraging this smoothness?

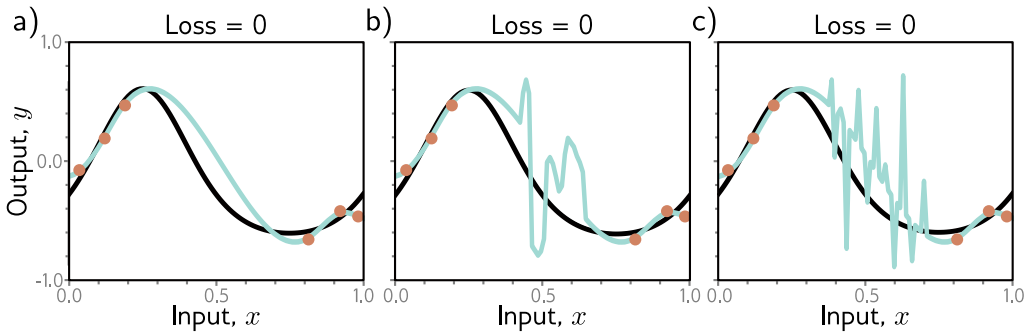


Figure 8.12 Regularization. a–c) Each of the three fitted curves passes through the data points exactly, so the training loss for each is zero. However, we might expect the smooth curve in panel (a) to generalize much better to new data than the erratic curves in panels (b) and (c). Any factor that biases a model toward a subset of the solutions with a similar training loss is known as a regularizer. It is thought that the initialization and/or fitting of neural networks have an implicit regularizing effect. Consequently, in the over-parameterized regime, more reasonable solutions, such as that in panel (a), are encouraged.

The answer to this question is uncertain, but there are two likely possibilities. First, the network initialization may encourage smoothness, and the model never departs from the sub-domain of smooth function during the training process. Second, the training algorithm may somehow "prefer" to converge to smooth functions. Any factor that biases a solution toward a subset of equivalent solutions is known as a *regularizer*, so one possibility is that the training algorithm acts as an implicit regularizer (see section 9.2).

8.5 Choosing hyperparameters

In the previous section, we discussed how test performance changes with model capacity. Unfortunately, in the classical regime, we don't have access to either the bias (which requires knowledge of the true underlying function) or the variance (which requires multiple independently sampled datasets to estimate). In the modern regime, there is no way to tell how much capacity should be added before the test error stops improving. This raises the question of exactly how we should choose model capacity in practice.

For a deep network, the model capacity depends on the numbers of hidden layers and hidden units per layer as well as other aspects of architecture that we have yet to introduce. Furthermore, the choice of learning algorithm and any associated parameters (learning rate, etc.) also affects the test performance. These elements are collectively termed *hyperparameters*. The process of finding the best hyperparameters is termed *hyperparameter search* or (when focused on network structure) *neural architecture search*.

Hyperparameters are typically chosen empirically; we train many models with different hyperparameters on the same training set, measure their performance, and retain the best model. However, we do not measure their performance on the test set; this would admit the possibility that these hyperparameters just happen to work well for the test set but don't generalize to further data. Instead, we introduce a third dataset known as a *validation set*. For every choice of hyperparameters, we train the associated model using the training set and evaluate performance on the validation set. Finally, we select the model that worked best on the validation set and measure its performance on the test set. In principle, this should give a reasonable estimate of the true performance.

The hyperparameter space is generally smaller than the parameter space but still too large to try every combination exhaustively. Unfortunately, many hyperparameters are discrete (e.g., the number of hidden layers), and others may be conditional on one another (e.g., we only need to specify the number of hidden units in the tenth hidden layer if there are ten or more layers). Hence, we cannot rely on gradient descent methods as we did for learning the model parameters. Hyperparameter optimization algorithms intelligently sample the space of hyperparameters, contingent on previous results. This procedure is computationally expensive since we must train an entire model and measure the validation performance for each combination of hyperparameters.

8.6 Summary

To measure performance, we use a separate test set. The degree to which performance is maintained on this test set is known as generalization. Test errors can be explained by three factors: noise, bias, and variance. These combine additively in regression problems with least squares losses. Adding training data decreases the variance. When the model capacity is less than the number of training examples, increasing the capacity decreases bias but increases variance. This is known as the bias-variance trade-off, and there is a capacity where the trade-off is optimal.

However, this is balanced against a tendency for performance to improve with capacity, even when the parameters exceed the training examples. Together, these two phenomena create the double descent curve. It is thought that the model interpolates more smoothly between the training data points in the over-parameterized "modern regime," although it is unclear what drives this. To choose the capacity and other model and training algorithm hyperparameters, we fit multiple models and evaluate their performance using a separate validation set.

Notes

Bias-variance trade-off: We showed that the test error for regression problems with least squares loss decomposes into the sum of noise, bias, and variance terms. These factors are all present for models with other losses, but their interaction is typically more complicated (Friedman, 1997; Domingos, 2000). For classification problems, there are some counter-intuitive

predictions; for example, if the model is biased toward selecting the wrong class in a region of the input space, then increasing the variance can improve the classification rate as this pushes some of the predictions over the threshold to be classified correctly.

Cross-validation: We saw that it is typical to divide the data into three parts: training data (which is used to learn the model parameters), validation data (which is used to choose the hyperparameters), and test data (which is used to estimate the final performance). This approach is known as *cross-validation*. However, this division may cause problems where the total number of data examples is limited; if the number of training examples is comparable to the model capacity, then the variance will be large.

One way to mitigate this problem is to use *k-fold cross-validation*. The training and validation data are partitioned into K disjoint subsets. For example, we might divide these data into five parts. We train with four and validate with the fifth for each of the five permutations and choose the hyperparameters based on the average validation performance. The final test performance is assessed using the average of the predictions from the five models with the best hyperparameters on an entirely different test set. There are many variations of this idea, but all share the general goal of using a larger proportion of the data to train the model, thereby reducing variance.

Capacity: We have used the term *capacity* informally to mean the number of parameters or hidden units in the model (and hence indirectly, the ability of the model to fit functions of increasing complexity). The *representational capacity* of a model describes the space of possible functions it can construct when we consider all possible parameter values. When we take into account the fact that an optimization algorithm may not be able to reach all of these solutions, what is left is the *effective capacity*.

The Vapnik-Chervonenkis (VC) dimension (Vapnik & Chervonenkis, 1971) is a more formal measure of capacity. It is the largest number of training examples that a binary classifier can label arbitrarily. Bartlett et al. (2019) derive upper and lower bounds for the VC dimension in terms of the number of layers and weights. An alternative measure of capacity is the Rademacher complexity, which is the expected empirical performance of a classification model (with optimal parameters) for data with random labels. Neyshabur et al. (2017) derive a lower bound on the generalization error in terms of the Rademacher complexity.

Double descent: The term "double descent" was coined by Belkin et al. (2019), who demonstrated that the test error decreases again in the over-parameterized regime for two-layer neural networks and random features. They also claimed that this occurs in decision trees, although Buschjäger & Morik (2021) subsequently provided evidence to the contrary. Nakkiran et al. (2021) show that double descent occurs for various modern datasets (CIFAR-10, CIFAR-100, IWSLT'14 de-en), architectures (CNNs, ResNets, transformers), and optimizers (SGD, Adam). The phenomenon is more pronounced when noise is added to the target labels (Nakkiran et al., 2021) and when some regularization techniques are used (Ishida et al., 2020).

Nakkiran et al. (2021) also provide empirical evidence that test performance depends on *effective model capacity* (the largest number of samples for which a given model and training method can achieve zero training error). At this point, the model starts to devote its efforts to interpolating smoothly. As such, the test performance depends not just on the model but also on the training algorithm and length of training. They observe the same pattern when they study a model with fixed capacity and increase the number of training iterations. They term this *epoch-wise double descent*. This phenomenon has been modeled by Pezeshki et al. (2022) in terms of different features in the model being learned at different speeds.

Double descent makes the rather strange prediction that adding training data can sometimes worsen test performance. Consider an over-parameterized model in the second descending part

of the curve. If we increase the training data to match the model capacity, we will now be in the critical region of the new test error curve, and the test loss may increase.

Bubeck & Sellke (2021) prove that overparameterization is necessary to interpolate data smoothly in high dimensions. They demonstrate a trade-off between the number of parameters and the Lipschitz constant of a model (the fastest the output can change for a small input change). A review of the theory of over-parameterized machine learning can be found in Dar et al. (2021).

Appendix B.1.1 Lipschitz constant

Curse of dimensionality: As dimensionality increases, the volume of space grows so fast that the amount of data needed to densely sample it increases exponentially. This phenomenon is known as the curse of dimensionality. High-dimensional space has many unexpected properties, and caution should be used when trying to reason about it based on low-dimensional examples. This book visualizes many aspects of deep learning in one or two dimensions, but these visualizations should be treated with healthy skepticism.

Surprising properties of high-dimensional spaces include: (i) Two randomly sampled data points from a standard normal distribution are very close to orthogonal to one another (relative to the origin) with high likelihood. (ii) The distance from the origin of samples from a standard normal distribution is roughly constant. (iii) Most of a volume of a high-dimensional sphere (hypersphere) is adjacent to its surface (a common metaphor is that most of the volume of a high-dimensional orange is in the peel, not in the pulp). (iv) If we place a unit-diameter hypersphere inside a hypercube with unit-length sides, then the hypersphere takes up a decreasing proportion of the volume of the cube as the dimension increases. Since the volume of the cube is fixed at size one, this implies that the volume of a high-dimensional hypersphere becomes close to zero. (v) For random points drawn from a uniform distribution in a high-dimensional hypercube, the ratio of the Euclidean distance between the nearest and furthest points becomes close to one. For further information, consult Beyer et al. (1999) and Aggarwal et al. (2001).

Problems 8.6–8.9

Notebook 8.4 High-dimensional spaces

Real-world performance: In this chapter, we argued that model performance could be evaluated using a held-out test set. However, the result won't be indicative of real-world performance if the statistics of the test set don't match those of real-world data. Moreover, the statistics of real-world data may change over time, causing the model to become increasingly stale and performance to decrease. This is known as *data drift* and means that deployed models must be carefully monitored.

There are three main reasons why real-world performance may be worse than the test performance implies. First, the statistics of the input data $\mathbf{x}$ may change; we may now be observing parts of the function that were sparsely sampled or not sampled at all during training. This is known as *covariate shift*. Second, the statistics of the output data $\mathbf{y}$ may change; if some output values are infrequent during training, then the model may learn not to predict these in ambiguous situations and will make mistakes if they are more common in the real world. This is known as *prior shift*. Third, the relationship between input and output may change. This is known as *concept shift*. These issues are discussed in Moreno-Torres et al. (2012).

Hyperparameter search: Finding the best hyperparameters is a challenging optimization task. Testing a single configuration of hyperparameters is expensive; we must train an entire model and measure its performance. We have no easy way to access the derivatives (i.e., how performance changes when we make a small change to a hyperparameter). Moreover, many of the hyperparameters are discrete, so we cannot use gradient descent methods. There are multiple local minima and no way to tell if we are close to the global minimum. The noise level is high since each training/validation cycle uses a stochastic training algorithm; we expect different results if we train a model twice with the same hyperparameters. Finally, some variables are conditional and only exist if others are set. For example, the number of hidden units in the third hidden layer is only relevant if we have at least three hidden layers.

A simple approach is to sample the space randomly (Bergstra & Bengio, 2012). However, for continuous variables, it is better to build a model of performance as a function of the hyperparameters and the uncertainty in this function. This can be exploited to test where the uncertainty is great (explore the space) or home in on regions where performance looks promising (exploit previous knowledge). Bayesian optimization is a framework based on Gaussian processes that does just this, and its application to hyperparameter search is described in Snoek et al. (2012). The Beta-Bernoulli bandit (see Lattimore & Szepesvári, 2020) is a roughly equivalent model for describing uncertainty in results due to discrete variables.

The sequential model-based configuration (SMAC) algorithm (Hutter et al., 2011) can cope with continuous, discrete, and conditional parameters. The basic approach is to use a random forest to model the objective function where the mean of the tree predictions is the best guess about the objective function, and their variance represents the uncertainty. A completely different approach that can also cope with combinations of continuous, discrete, and conditional parameters is Tree-Parzen Estimators (Bergstra et al., 2011). The previous methods modeled the probability of the model performance given the hyperparameters. In contrast, the Tree-Parzen estimator models the probability of the hyperparameters given the model performance.

Hyperband (Li et al., 2017b) is a multi-armed bandit strategy for hyperparameter optimization. It assumes that there are computationally cheap but approximate ways to measure performance (e.g., by not training to completion) and that these can be associated with a budget (e.g., by training for a fixed number of iterations). A number of random configurations are sampled and run until the budget is used up. Then the best fraction η of runs is kept, and the budget is multiplied by $1/\eta$. This is repeated until the maximum budget is reached. This approach has the advantage of efficiency; for bad configurations, it does not need to run the experiment to the end. However, each sample is just chosen randomly, which is inefficient. The BOHB algorithm (Falkner et al., 2018) combines the efficiency of Hyperband with the more sensible choice of hyperparameters from Tree Parzen estimators to construct an even better method.

Problems

Problem 8.1 Will the multiclass cross-entropy training loss in figure 8.2 ever reach zero? Explain your reasoning.

Problem 8.2 What values should we choose for the three weights and biases in the first layer of the model in figure 8.4a so that the hidden unit's responses are as depicted in figures 8.4b–d?

Problem 8.3* Given a training dataset consisting of I input/output pairs $\{x_i, y_i\}$, show how the parameters $\{\beta, \omega_1, \omega_2, \omega_3\}$ for the model in figure 8.4a using the least squares loss function can be found in closed form.

Problem 8.4 Consider the curve in figure 8.10b at the point where we train a model with a hidden layer of size 200, which would have 50,410 parameters. What do you predict will happen to the training and test performance if we increase the number of training examples from 10,000 to 50,410?

Problem 8.5 Consider the case where the model capacity exceeds the number of training data points, and the model is flexible enough to reduce the training loss to zero. What are the implications of this for fitting a heteroscedastic model? Propose a method to resolve any problems that you identify.

Problem 8.6 Show that two random points drawn from a 1000-dimensional standard Gaussian distribution are orthogonal relative to the origin with high probability.

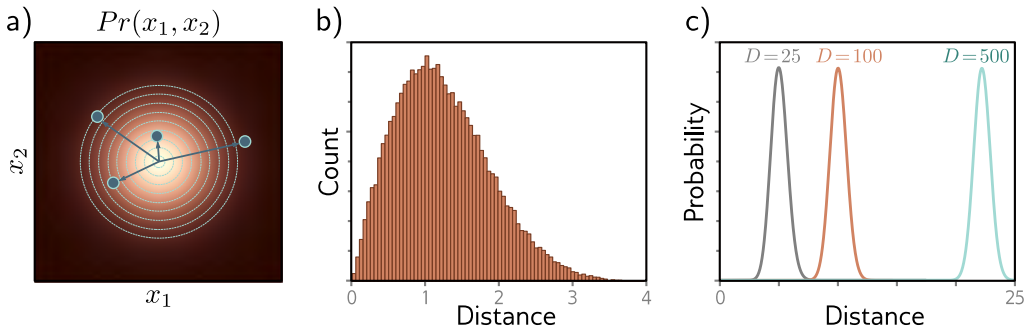


Figure 8.13 Typical sets. a) Standard normal distribution in two dimensions. Circles are four samples from this distribution. As the distance from the center increases, the probability decreases, but the volume of space at that radius (i.e., the area between adjacent evenly spaced circles) increases. b) These factors trade off so that the histogram of distances of samples from the center has a pronounced peak. c) In higher dimensions, this effect becomes more extreme, and the probability of observing a sample close to the mean becomes vanishingly small. Although the most likely point is at the mean of the distribution, the *typical samples* are found in a relatively narrow shell.

Problem 8.7 The volume of a hypersphere with radius r in D dimensions is:

$$\text{Vol}[r] = \frac{r^D \pi^{D/2}}{\Gamma[D/2+1]}, \tag{8.8}$$

where $\Gamma[\bullet]$ is the Gamma function. Show using Stirling's formula that the volume of a hypersphere of diameter one (radius $r=0.5$) becomes zero as the dimension increases.

Appendix B.1.3 Gamma function

Appendix B.1.4 Stirling's formula

Problem 8.8* Consider a hypersphere of radius $r = 1$. Find an expression for the proportion of the total volume that lies in the outermost 1% of the distance from the center (i.e., in the outermost shell of thickness 0.01). Show that this becomes one as the dimension increases.

Problem 8.9 Figure 8.13c shows the distribution of distances of samples of a standard normal distribution as the dimension increases. Empirically verify this finding by sampling from the standard normal distributions in 25, 100, and 500 dimensions and plotting a histogram of the distances from the center. What closed-form probability distribution describes these distances?

Chapter 9

Regularization

Chapter 8 described how to measure model performance and identified that there could be a significant performance gap between the training and test data. Possible reasons for this discrepancy include: (i) the model describes statistical peculiarities of the training data that are not representative of the true mapping from input to output (overfitting), and (ii) the model is unconstrained in areas with no training examples, leading to suboptimal predictions.

This chapter discusses *regularization* techniques. These are a family of methods that reduce the generalization gap between training and test performance. Strictly speaking, regularization involves adding explicit terms to the loss function that favor certain parameter choices. However, in machine learning, this term is commonly used to refer to any strategy that improves generalization.

We start by considering regularization in its strictest sense. Then we show how the stochastic gradient descent algorithm itself favors certain solutions. This is known as implicit regularization. Following this, we consider a set of heuristic methods that improve test performance. These include early stopping, ensembling, dropout, label smoothing, and transfer learning.

9.1 Explicit regularization

Consider fitting a model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ using a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek the minimum of the loss function $L[\boldsymbol{\phi}]$:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\big[L[\boldsymbol{\phi}]\big] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, \mathbf{y}_i]\right], \end{aligned} \tag{9.1}$$

where the individual terms $\ell_i[\mathbf{x}_i, \mathbf{y}_i]$ measure the mismatch between the network predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and output targets $\mathbf{y}_i$ for each training pair. To bias this minimization

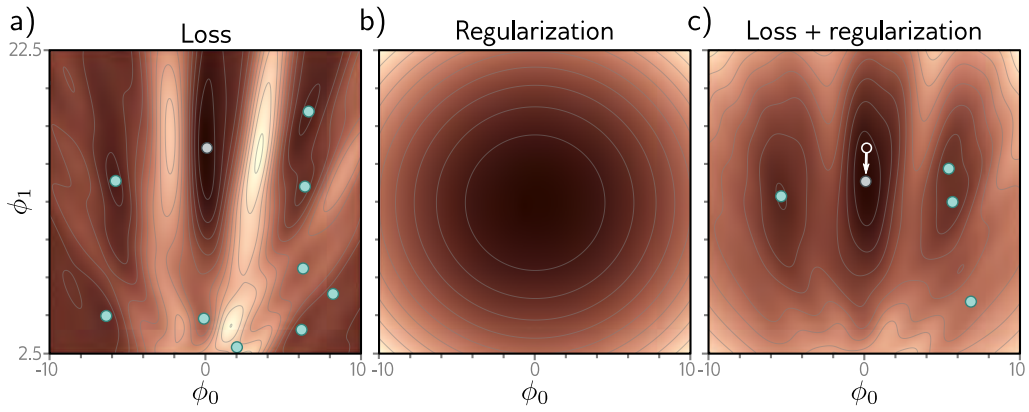


Figure 9.1 Explicit regularization. a) Loss function for Gabor model (see section 6.1.2). Cyan circles represent local minima. Gray circle represents the global minimum. b) The regularization term favors parameters close to the center of the plot by adding an increasing penalty as we move away from this point. c) The final loss function is the sum of the original loss function plus the regularization term. This surface has fewer local minima, and the global minimum has moved to a different position (arrow shows change).

toward certain solutions, we include an additional term:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, \mathbf{y}_i] + \lambda \cdot \mathrm{g}[\boldsymbol{\phi}] \right], \tag{9.2}$$

where $\mathrm{g}[\boldsymbol{\phi}]$ is a function that returns a scalar that takes a larger value when the parameters are less preferred. The term λ is a positive scalar that controls the relative contribution of the original loss function and the regularization term. The minima of the regularized loss function usually differ from those in the original, so the training procedure converges to different parameter values (figure 9.1).

9.1.1 Probabilistic interpretation

Regularization can be viewed from a probabilistic perspective. Section 5.1 shows how loss functions are constructed from the maximum likelihood criterion:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) \right]. \tag{9.3}$$

The regularization term can be considered as a *prior* $Pr(\boldsymbol{\phi})$ that represents knowledge about the parameters before we observe the data and we now have the *maximum a posteriori* or *MAP* criterion:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}) \right]. \tag{9.4}$$

Moving back to the negative log-likelihood loss function by taking the log and multiplying by minus one, we see that $\lambda \cdot \mathrm{g}[\boldsymbol{\phi}] = -\log[Pr(\boldsymbol{\phi})]$.

9.1.2 L2 regularization

This discussion has sidestepped the question of *which* solutions the regularization term should penalize (or equivalently that the prior should favor). Since neural networks are used in an extremely broad range of applications, these can only be very generic preferences. The most commonly used regularization term is the *L2 norm*, which penalizes the sum of the squares of the parameter values:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, \mathbf{y}_i] + \lambda \sum_{j} \phi_j^2 \right], \tag{9.5}$$

Problems 9.1–9.2

where j indexes the parameters. This is also referred to as *Tikhonov regularization* or *ridge regression*, or (when applied to matrices) *Frobenius norm regularization*.

For neural networks, L2 regularization is usually applied to the weights but not the biases and is hence referred to as a *weight decay* term. The effect is to encourage smaller weights, so the output function is smoother. To see this, consider that the output prediction is a weighted sum of the activations at the last hidden layer. If the weights have a smaller magnitude, the output will vary less. The same logic applies to the computation of the pre-activations at the last hidden layer and so on, progressing backward through the network. In the limit, if we forced all the weights to be zero, the network would produce a constant output determined by the final bias parameter.

Notebook 9.1
L2 regularization

Figure 9.2 shows the effect of fitting the simplified network from figure 8.4 with weight decay and different values of the regularization coefficient λ. When λ is small, it has little effect. However, as λ increases, the fit to the data becomes less accurate, and the function becomes smoother. This might improve the test performance for two reasons:

- If the network is overfitting, then adding the regularization term means that the network must trade off slavish adherence to the data against the desire to be smooth. One way to think about this is that the error due to variance reduces (the model no longer needs to pass through every data point) at the cost of increased bias (the model can only describe smooth functions).
- When the network is over-parameterized, some of the extra model capacity describes areas with no training data. Here, the regularization term will favor functions that smoothly interpolate between the nearby points. This is reasonable behavior in the absence of knowledge about the true function.

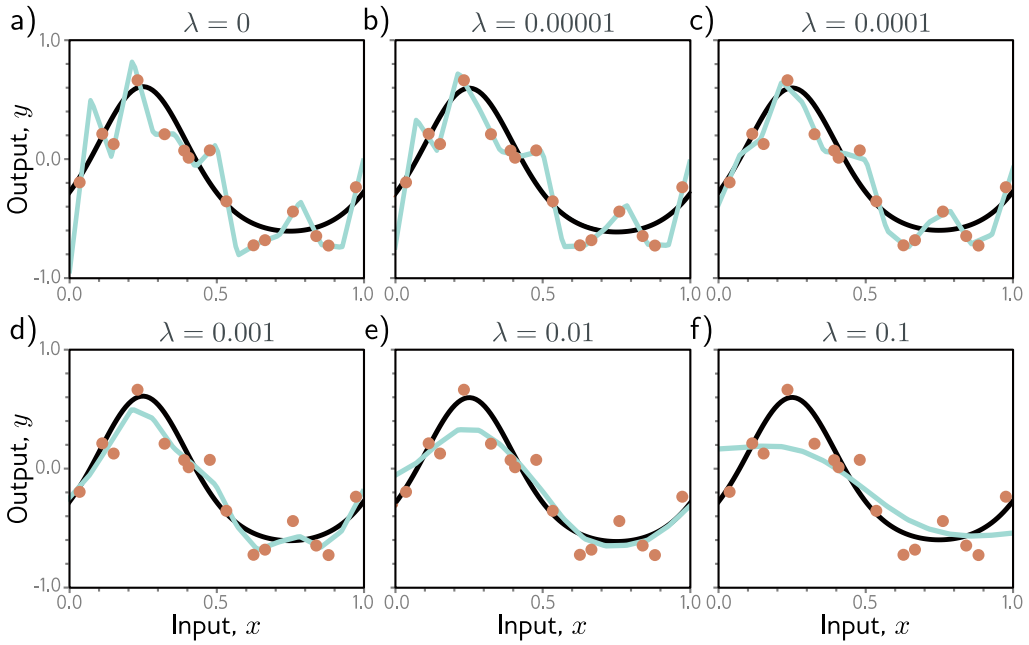


Figure 9.2 L2 regularization in simplified network (see figure 8.4). a–f) Fitted functions as we increase the regularization coefficient λ. The black curve is the true function, the orange circles are the noisy training data, and the cyan curve is the fitted model. For small λ (panels a–b), the fitted function passes exactly through the data points. For intermediate λ (panels c–d), the function is smoother and more similar to the ground truth. For large λ (panels e–f), the fitted function is smoother than the ground truth, so the fit is worse.

9.2 Implicit regularization

An intriguing recent finding is that neither gradient descent nor stochastic gradient descent moves neutrally to the minimum of the loss function; each exhibits a preference for some solutions over others. This is known as *implicit regularization*.

9.2.1 Implicit regularization in gradient descent

Consider a continuous version of gradient descent where the step size is infinitesimal. The change in parameters $\boldsymbol{\phi}$ will be governed by the differential equation:

$$\frac{d\boldsymbol{\phi}}{dt} = -\frac{\partial L}{\partial \boldsymbol{\phi}}. \tag{9.6}$$

Gradient descent approximates this process with a series of discrete steps of size α:

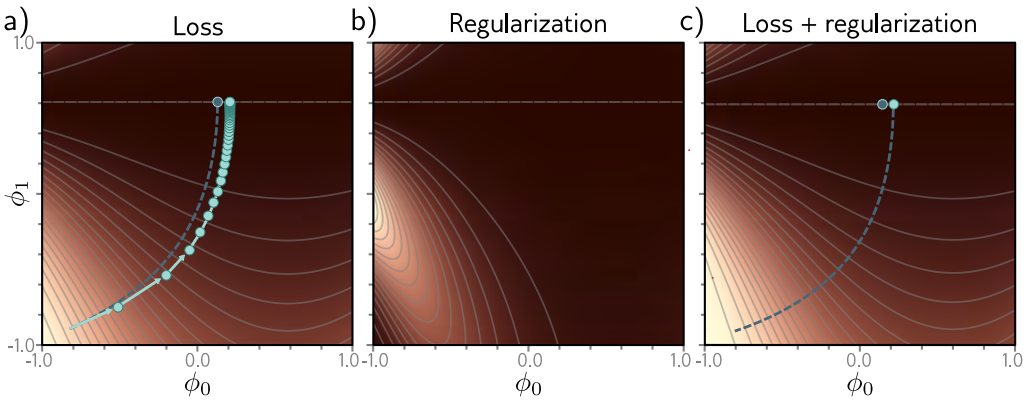


Figure 9.3 Implicit regularization in gradient descent. a) Loss function with family of global minima on horizontal line $\phi_1 = 0.61$. Dashed blue line shows continuous gradient descent path starting in bottom-left. Cyan trajectory shows discrete gradient descent with step size 0.1 (first few steps shown explicitly as arrows). The finite step size causes the paths to diverge and reach a different final position. b) This disparity can be approximated by adding a regularization term to the continuous gradient descent loss function that penalizes the squared gradient magnitude. c) After adding this term, the continuous gradient descent path converges to the same place that the discrete one did on the original function.

$$\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}_t - \alpha \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{9.7}$$

The discretization causes a deviation from the continuous path (figure 9.3).

This deviation can be understood by deriving a modified loss term $\tilde{L}$ for the continuous case that arrives at the same place as the discretized version on the original loss L. It can be shown (see end of chapter) that this modified loss is:

$$\tilde{L}_{GD}[\boldsymbol{\phi}] = L[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2. \tag{9.8}$$

In other words, the discrete trajectory is repelled from places where the gradient norm is large (the surface is steep). This doesn't change the position of the minima where the gradients are zero anyway. However, it changes the effective loss function elsewhere and modifies the optimization trajectory, which potentially converges to a different minimum. Implicit regularization due to gradient descent may be responsible for the observation that full batch gradient descent generalizes better with larger step sizes (figure 9.5a).

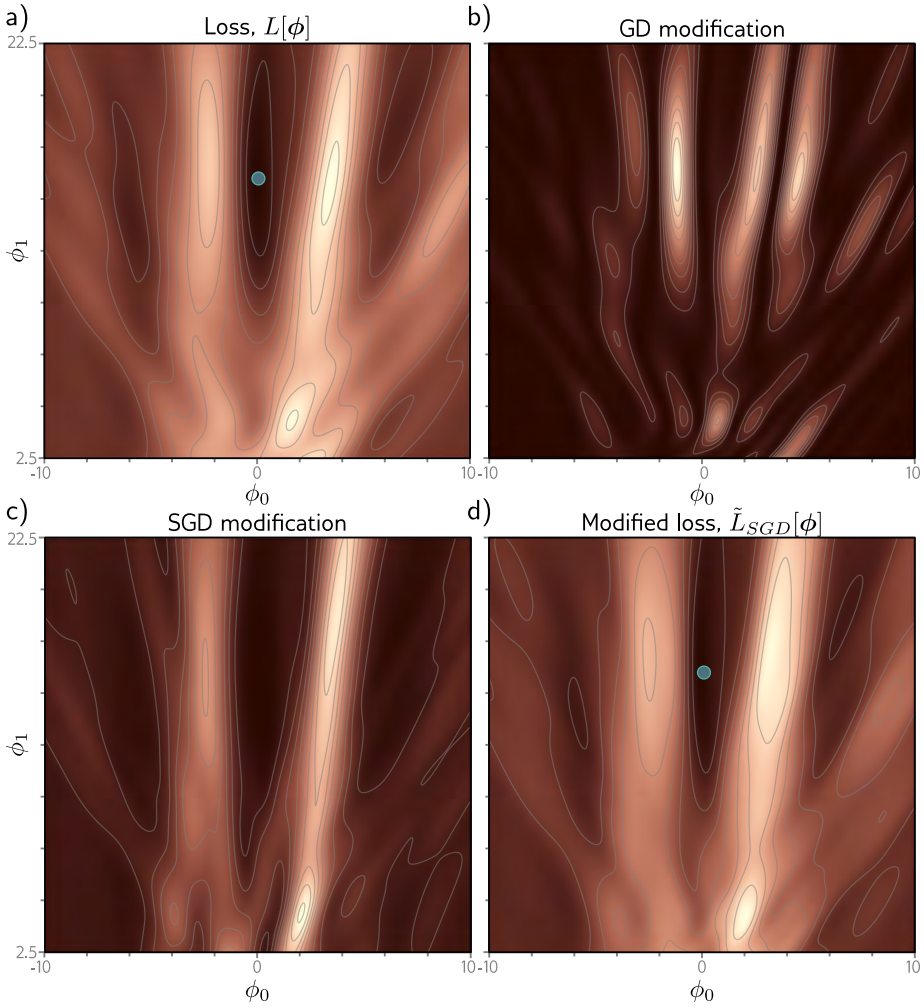


Figure 9.4 Implicit regularization for stochastic gradient descent. a) Original loss function for Gabor model (section 6.1.2). Blue point represents global minimum. b) Implicit regularization term from gradient descent penalizes the squared gradient magnitude. c) Additional implicit regularization from stochastic gradient descent penalizes the variance of the batch gradients. d) Modified loss function (sum of original loss plus two implicit regularization components). Blue point represents global minimum which may now be in a different place from panel (a).

9.2.2 Implicit regularization in stochastic gradient descent

A similar analysis can be applied to stochastic gradient descent. Now we seek a modified loss function such that the continuous version reaches the same place as the average of the possible random SGD updates. This can be shown to be:

$$\begin{aligned}\tilde{L}_{SGD}[\boldsymbol{\phi}] &= \tilde{L}_{GD}[\boldsymbol{\phi}] + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 \\ &= L[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2. \end{aligned} \tag{9.9}$$

Here, L_b is the loss for the b^{th} of the B batches in an epoch, and both L and L_b now represent the means of the I individual losses in the full dataset and the $|\mathcal{B}|$ individual losses in the batch, respectively:

$$L = \frac{1}{I}\sum_{i=1}^{I}\ell_i[\mathbf{x}_i, y_i] \qquad \text{and} \qquad L_b = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_b}\ell_i[\mathbf{x}_i, y_i]. \tag{9.10}$$

Equation 9.9 reveals an extra regularization term, which corresponds to the variance of the gradients of the batch losses L_b. In other words, SGD implicitly favors places where the gradients are stable (where all the batches agree on the slope). Once more, this modifies the trajectory of the optimization process (figure 9.4) but does not necessarily change the position of the global minimum; if the model is over-parameterized, then it may fit all the training data exactly, so *all* of these gradient terms will all be zero at the global minimum.

SGD generalizes better than gradient descent, and smaller batch sizes generally perform better than larger ones (figure 9.5b). One possible explanation is that the inherent randomness allows the algorithm to reach different parts of the loss function. However, it's also possible that some or all of this performance increase is due to implicit regularization; this encourages solutions where all the data fits well (so the batch variance is small) rather than solutions where some of the data fit extremely well and other data less well (perhaps with the same overall loss, but with larger batch variance). The former solutions are likely to generalize better.

Notebook 9.2 Implicit regularization

9.3 Heuristics to improve performance

We've seen that adding explicit regularization terms encourages the training algorithm to find a good solution by adding extra terms to the loss function. This also occurs implicitly as an unintended (but seemingly helpful) byproduct of stochastic gradient descent. This section describes other heuristic methods used to improve generalization.

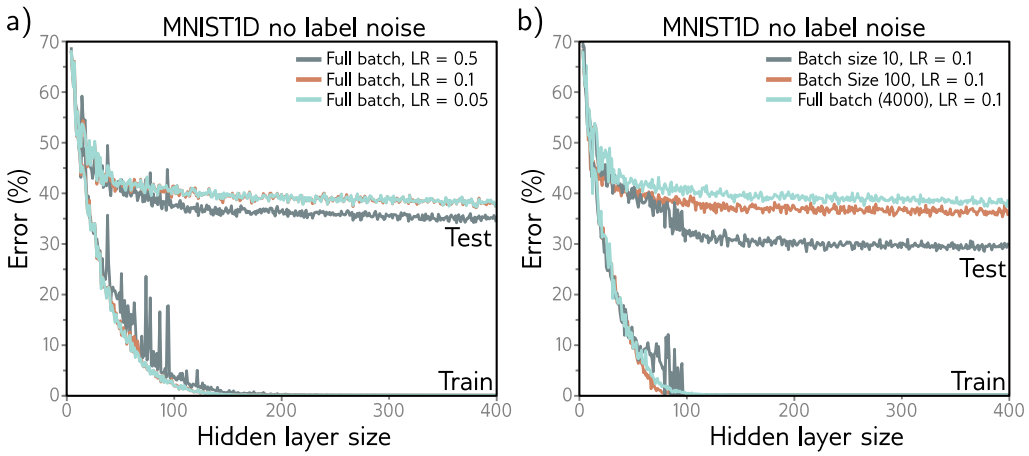


Figure 9.5 Effect of learning rate (LR) and batch size for 4000 training and 4000 test examples from MNIST-1D (see figure 8.1) for a neural network with two hidden layers. a) Performance is better for large learning rates than for intermediate or small ones. In each case, the number of iterations is 6000/LR, so each solution has the opportunity to move the same distance. b) Performance is superior for smaller batch sizes. In each case, the number of iterations was chosen so that the training data were memorized at roughly the same model capacity.

9.3.1 Early stopping

Early stopping refers to stopping the training procedure before it has fully converged. This can reduce overfitting if the model has already captured the coarse shape of the underlying function but has not yet had time to overfit to the noise (figure 9.6). One way of thinking about this is that since the weights are initialized to small values (see section 7.5), they simply don't have time to become large, so early stopping has a similar effect to explicit L2 regularization. A different view is that early stopping reduces the effective model complexity. Hence, we move back down the bias/variance trade-off curve from the critical region, and performance improves (see figures 8.9 and 8.10).

Early stopping has a single hyperparameter, the number of steps after which learning is terminated. As usual, this is chosen empirically using a validation set (section 8.5). However, for early stopping, the hyperparameter can be selected without the need to train multiple models. The model is trained once, the performance on the validation set is monitored every T iterations, and the associated models are stored. The stored model where the validation performance was best is selected.

9.3.2 Ensembling

Another approach to reducing the generalization gap between training and test data is to build several models and average their predictions. A group of such models is known

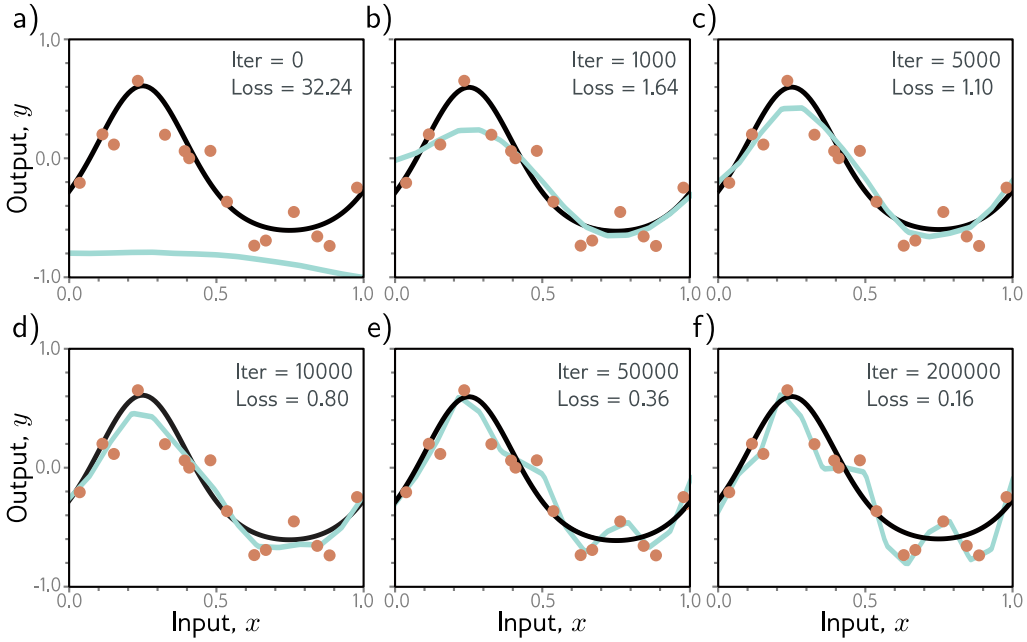


Figure 9.6 Early stopping. a) Simplified shallow network model with 14 linear regions (figure 8.4) is initialized randomly (cyan curve) and trained with SGD using a batch size of five and a learning rate of 0.05. b–d) As training proceeds, the function first captures the coarse structure of the true function (black curve) before e–f) overfitting to the noisy training data (orange points). Although the training loss continues to decrease throughout this process, the learned models in panels (c) and (d) are closest to the true underlying function. They will generalize better on average to test data than those in panels (e) or (f).

as an *ensemble*. This technique reliably improves test performance at the cost of training and storing multiple models and performing inference multiple times.

The models can be combined by taking the mean of the outputs (for regression problems) or the mean of the pre-softmax activations (for classification problems). The assumption is that model errors are independent and will cancel out. Alternatively, we can take the median of the outputs (for regression problems) or the most frequent predicted class (for classification problems) to make the predictions more robust.

Notebook 9.3
Ensembling

One way to train different models is just to use different random initializations. This may help in regions of input space far from the training data. Here, the fitted function is relatively unconstrained, and different models may produce different predictions, so the average of several models may generalize better than any single model.

A second approach is to generate several different datasets by re-sampling the training data with replacement and training a different model from each. This is known as *bootstrap aggregating* or *bagging* for short (figure 9.7). It has the effect of smoothing out the data; if a data point is not present in one training set, the model will interpo-

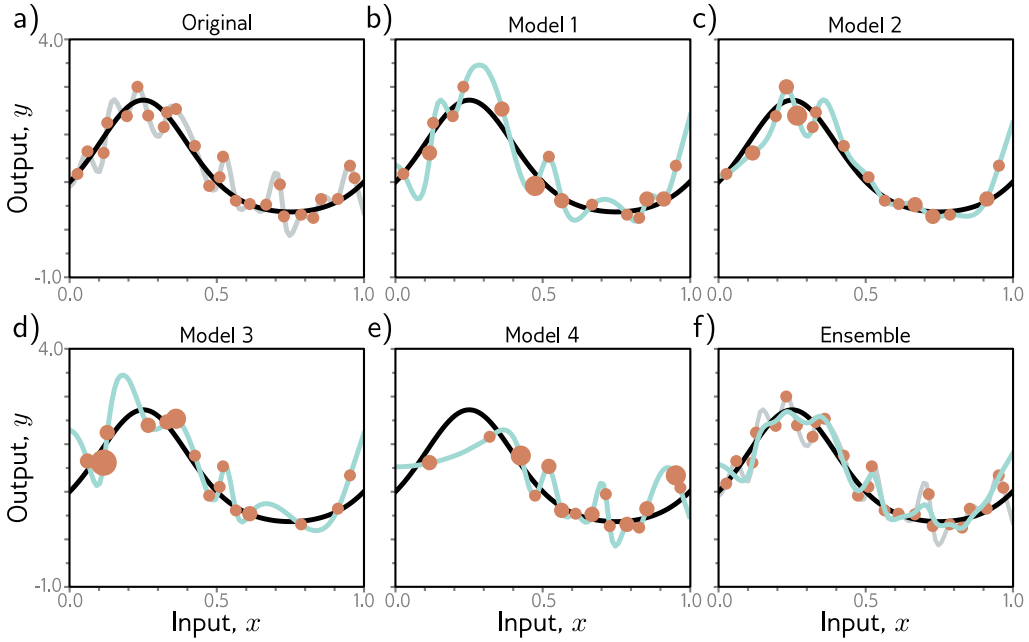


Figure 9.7 Ensemble methods. a) Fitting a single model (gray curve) to the entire dataset (orange points). b–e) Four models created by re-sampling the data with replacement (bagging) four times (size of orange point indicates number of times the data point was re-sampled). f) When we average the predictions of this ensemble, the result (cyan curve) is smoother than the result from panel (a) for the full dataset (gray curve) and will probably generalize better.

late from nearby points; hence, if that point was an outlier, the fitted function will be more moderate in this region. Other approaches include training models with different hyperparameters or training completely different families of models.

9.3.3 Dropout

Dropout randomly clamps a subset (typically 50%) of hidden units to zero at each iteration of SGD (figure 9.8). This makes the network less dependent on any given hidden unit and encourages the weights to have smaller magnitudes so that the change in the function due to the presence or absence of the hidden unit is reduced.

This technique has the positive benefit that it can eliminate undesirable "kinks" in the function that are far from the training data and don't affect the loss. For example, consider three hidden units that become active sequentially as we move along the curve (figure 9.9a). The first hidden unit causes a large increase in the slope. A second hidden

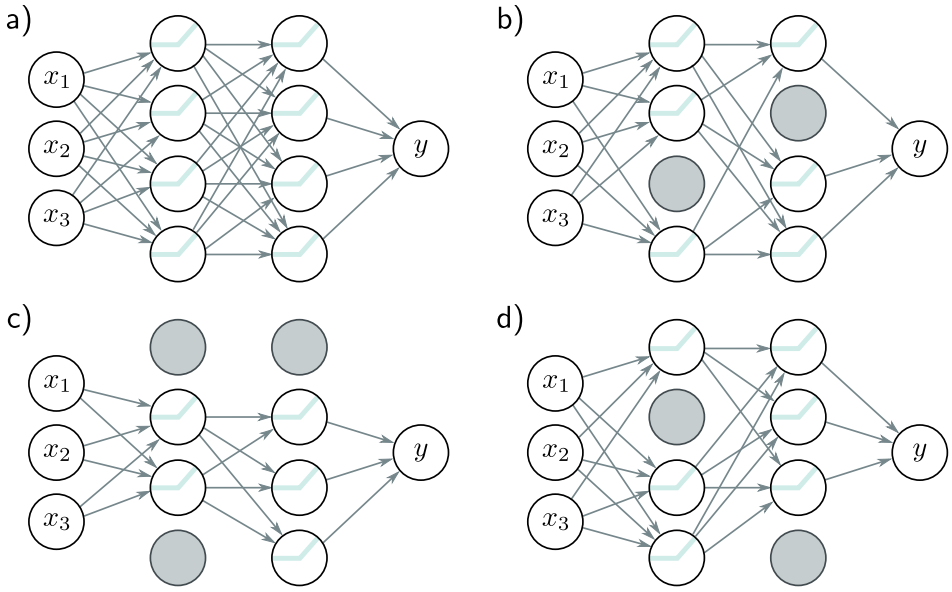


Figure 9.8 Dropout. a) Original network. b–d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes). The result is that the incoming and outgoing weights from these units have no effect, so we are training with a slightly different network each time.

unit decreases the slope, so the function goes back down. Finally, the third unit cancels out this decrease and returns the curve to its original trajectory. These three units conspire to make an undesirable local change in the function. This will not change the training loss but is unlikely to generalize well.

When several units conspire in this way, eliminating one (as would happen in dropout) causes a considerable change to the output function that is propagated to the half-space where that unit was active (figure 9.9b). A subsequent gradient descent step will attempt to compensate for the change that this induces, and such dependencies will be eliminated over time. The overall effect is that large unnecessary changes between training data points are gradually removed even though they contribute nothing to the loss (figure 9.9).

At test time, we can run the network as usual with all the hidden units active; however, the network now has more hidden units than it was trained with at any given iteration, so we multiply the weights by one minus the dropout probability to compensate. This is known as the *weight scaling inference rule*. A different approach to inference is to use *Monte Carlo dropout*, in which we run the network multiple times with different random subsets of units clamped to zero (as in training) and combine the results. This is closely related to ensembling in that every random version of the network is a different model; however, we do not have to train or store multiple networks here.

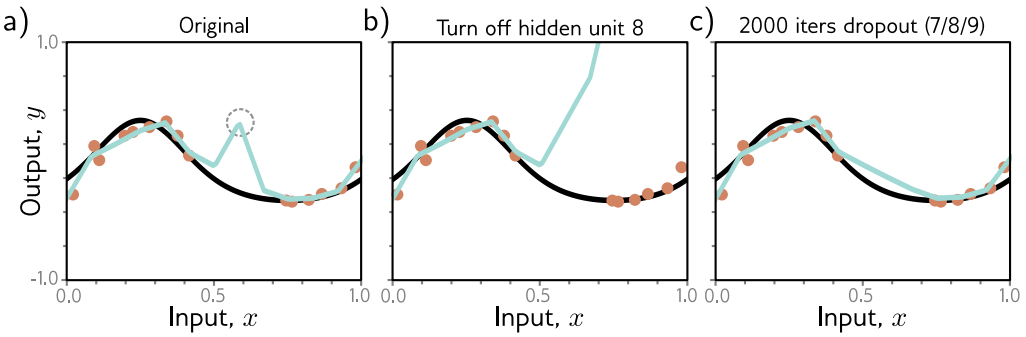


Figure 9.9 Dropout mechanism. a) An undesirable kink in the curve is caused by a sequential increase in the slope, decrease in the slope (at circled joint), and then another increase to return the curve to its original trajectory. Here we are using full-batch gradient descent, and the model (from figure 8.4) fits the data as well as possible, so further training won't remove the kink. b) Consider what happens if we remove the eighth hidden unit that produced the circled joint in panel (a), as might happen using dropout. Without the decrease in the slope, the right-hand side of the function takes an upwards trajectory, and a subsequent gradient descent step will aim to compensate for this change. c) Curve after 2000 iterations of (i) randomly removing one of the three hidden units that cause the kink and (ii) performing a gradient descent step. The kink does not affect the loss but is nonetheless removed by this approximation of the dropout mechanism.

9.3.4 Applying noise

Dropout can be interpreted as applying multiplicative Bernoulli noise to the network activations. This leads to the idea of applying noise to other parts of the network during training to make the final model more robust.

One option is to add noise to the input data; this smooths out the learned function (figure 9.10). For regression problems, it can be shown to be equivalent to adding a regularizing term that penalizes the derivatives of the network's output with respect to its input. An extreme variant is *adversarial training*, in which the optimization algorithm actively searches for small perturbations of the input that cause large changes to the output. These can be thought of as worst-case additive noise vectors.

Problem 9.3

A second possibility is to add noise to the weights. This encourages the network to make sensible predictions even for small perturbations of the weights. The result is that the training converges to local minima in the middle of wide, flat regions, where changing the individual weights does not matter much.

Finally, we can perturb the labels. The maximum-likelihood criterion for multiclass classification aims to predict the correct class with absolute certainty (equation 5.24). To this end, the final network activations (i.e., before the softmax function) are pushed to very large values for the correct class and very small values for the wrong classes.

We could discourage this overconfident behavior by assuming that a proportion ρ of

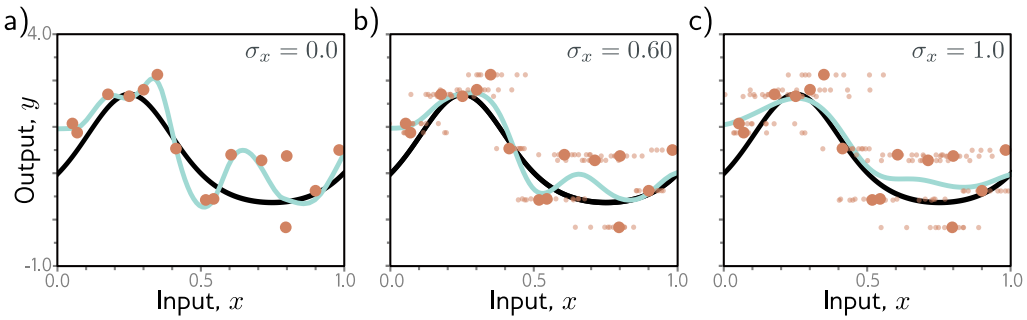


Figure 9.10 Adding noise to inputs. At each step of SGD, random noise with variance σ_x^2 is added to the batch data. a–c) Fitted model with different noise levels (small dots represent ten samples). Adding more noise smooths out the fitted function (cyan line).

the training labels are incorrect and belong with equal probability to the other classes. This could be done by randomly changing the labels at each training iteration. However, the same end can be achieved by changing the loss function to minimize the cross-entropy between the predicted distribution and a distribution where the true label has probability $1-\rho$, and the other classes have equal probability. This is known as *label smoothing* and improves generalization in diverse scenarios.

Problem 9.4

9.3.5 Bayesian inference

The maximum likelihood approach is generally overconfident; in the training phase, it selects the most likely parameters and bases its predictions on the model defined by these. However, many parameter values may be broadly compatible with the data and only slightly less likely. The Bayesian approach treats the parameters as unknown variables and computes a distribution $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ over these parameters $\boldsymbol{\phi}$ conditioned on the training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using Bayes' rule:

Appendix C.1.4 Bayes' rule

$$Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) = \frac{\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi})}{\int \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}) d\boldsymbol{\phi}}, \tag{9.11}$$

where $Pr(\boldsymbol{\phi})$ is the prior probability of the parameters, and the denominator is a normalizing term. Hence, every parameter choice is assigned a probability (figure 9.11).

The prediction $\mathbf{y}$ for new input $\mathbf{x}$ is an infinite weighted sum (i.e., an integral) of the predictions for each parameter set, where the weights are the associated probabilities:

$$Pr(\mathbf{y}|\mathbf{x}, \{\mathbf{x}_i, \mathbf{y}_i\}) = \int Pr(\mathbf{y}|\mathbf{x}, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) d\boldsymbol{\phi}. \tag{9.12}$$

This is effectively an infinite weighted ensemble, where the weight depends on (i) the prior probability of the parameters and (ii) their agreement with the data.

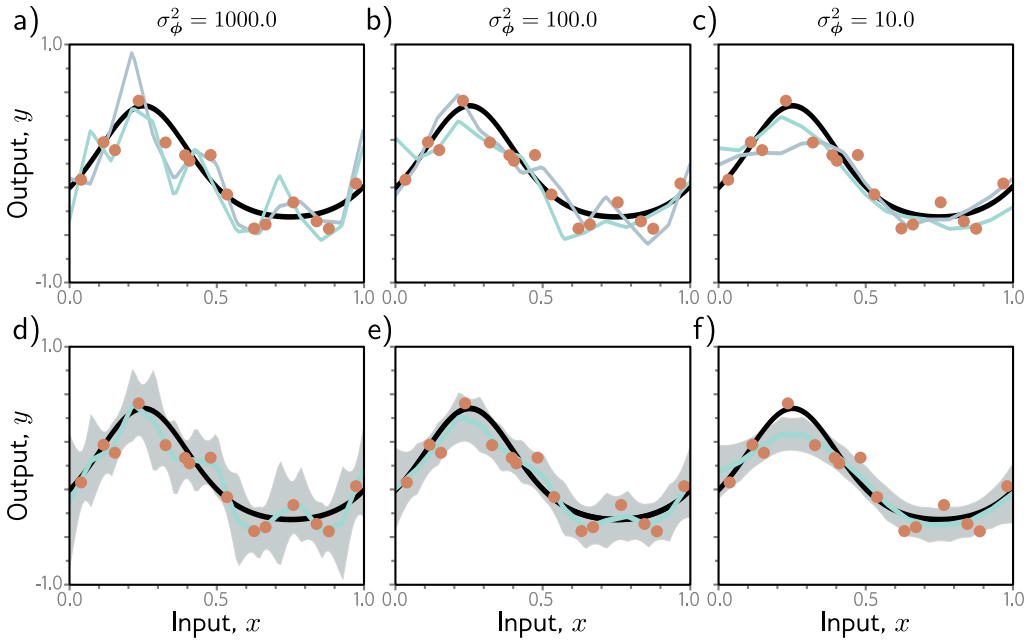


Figure 9.11 Bayesian approach for simplified network model (see figure 8.4). The parameters are treated as uncertain. The posterior probability $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ for a set of parameters is determined by their compatibility with the data $\{\mathbf{x}_i, \mathbf{y}_i\}$ and a prior distribution $Pr(\boldsymbol{\phi})$. a–c) Two sets of parameters (cyan and gray curves) sampled from the posterior using normally distributed priors with mean zero and three variances. When the prior variance σ_ϕ^2 is small, the parameters also tend to be small, and the functions smoother. d–f) Inference proceeds by taking a weighted sum over all possible parameter values where the weights are the posterior probabilities. This produces both a prediction of the mean (cyan curves) and the associated uncertainty (gray region is two standard deviations).

The Bayesian approach is elegant and can provide more robust predictions than those that derive from maximum likelihood. Unfortunately, for complex models like neural networks, there is no practical way to represent the full probability distribution over the parameters or to integrate over it during the inference phase. Consequently, all current methods of this type make approximations of some kind, and typically these add considerable complexity to learning and inference.

Notebook 9.4 Bayesian approach

9.3.6 Transfer learning and multi-task learning

When training data are limited, other datasets can be exploited to improve performance. In *transfer learning* (figure 9.12a), the network is *pre-trained* to perform a related sec-

ondary task for which data are more plentiful. The resulting model is then adapted to the original task. This is typically done by removing the last layer and adding one or more layers that produce a suitable output. The main model may be fixed, and the new layers trained for the original task, or we may *fine-tune* the entire model.

The principle is that the network will build a good internal representation of the data from the secondary task, which can subsequently be exploited for the original task. Equivalently, transfer learning can be viewed as initializing most of the parameters of the final network in a sensible part of the space that is likely to produce a good solution.

Multi-task learning (figure 9.12b) is a related technique in which the network is trained to solve several problems concurrently. For example, the network might take an image and simultaneously learn to segment the scene, estimate the pixel-wise depth, and predict a caption describing the image. All of these tasks require some understanding of the image and, when learned simultaneously, the model performance for each may improve.

9.3.7 Self-supervised learning

The above discussion assumes that we have plentiful data for a secondary task or data for multiple tasks to be learned concurrently. If not, we can create large amounts of "free" labeled data using *self-supervised* learning and use this for transfer learning. There are two families of methods for self-supervised learning: *generative* and *contrastive*.

In *generative self-supervised learning*, part of each data example is masked, and the secondary task is to predict the missing part (figure 9.12c). For example, we might use a corpus of unlabeled images and a secondary task that aims to *inpaint* (fill in) missing parts of the image (figure 9.12c). Similarly, we might use a large corpus of text and mask some words. We train the network to predict the missing words and then fine-tune it for the actual language task we are interested in (see chapter 12).

In *contrastive self-supervised learning*, pairs of examples with commonalities are compared to unrelated pairs. For images, the secondary task might be to identify whether a pair of images are transformed versions of one another or are unconnected. For text, the secondary task might be to determine whether two sentences followed one another in the original document. Sometimes, the precise relationship between a connected pair must be identified (e.g., finding the relative position of two patches from the same image).

9.3.8 Augmentation

Transfer learning improves performance by exploiting a different dataset. Multi-task learning improves performance using additional labels. A third option is to expand the dataset. We can often transform each input data example in such a way that the label stays the same. For example, we might aim to determine if there is a bird in an image (figure 9.13). Here, we could rotate, flip, blur, or manipulate the color balance of the image, and the label "bird" remains valid. Similarly, for tasks where the input is text, we can substitute synonyms or translate to another language and back again. For tasks where the input is audio, we can amplify or attenuate different frequency bands.

Notebook 9.5 Augmentation

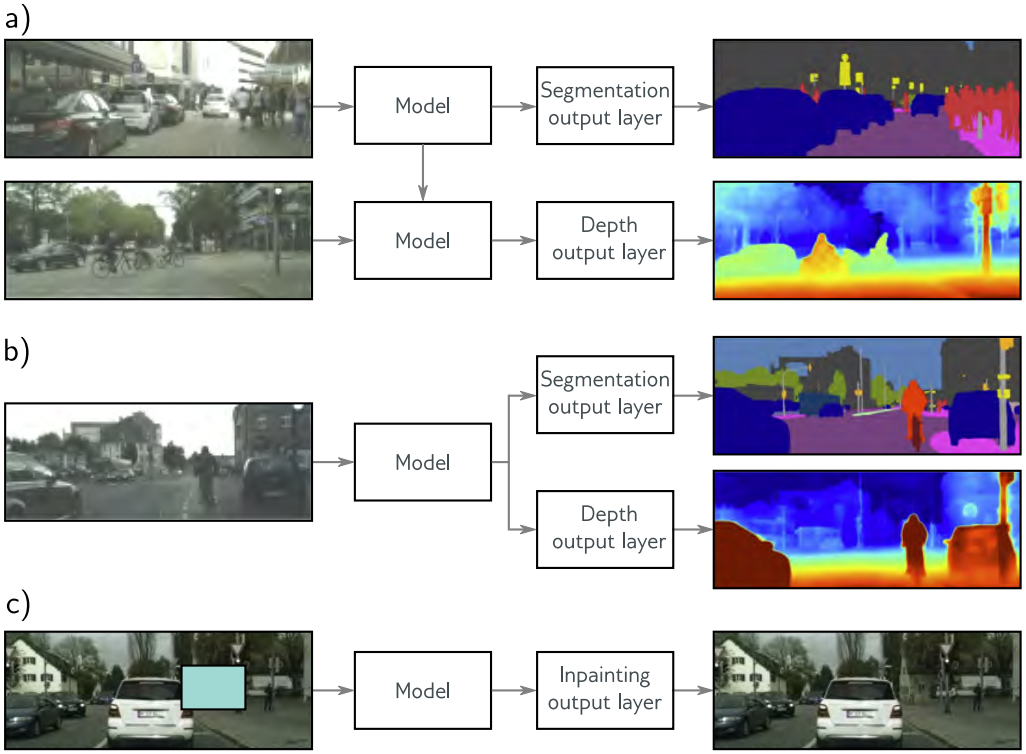


Figure 9.12 Transfer, multi-task, and self-supervised learning. a) Transfer learning is used when we have limited labeled data for the primary task (here depth estimation) but plentiful data for a secondary task (here segmentation). We train a model for the secondary task, remove the final layers, and replace them with new layers appropriate to the primary task. We then train only the new layers or fine-tune the entire network for the primary task. The network learns a good internal representation from the secondary task that is then exploited for the primary task. b) In multi-task learning, we train a model to perform multiple tasks simultaneously, hoping that performance on each will improve. c) In generative self-supervised learning, we remove part of the data and train the network to complete the missing information. Here, the task is to fill in (inpaint) a masked portion of the image. This permits transfer learning when no labels are available. Images from Cordts et al. (2016).

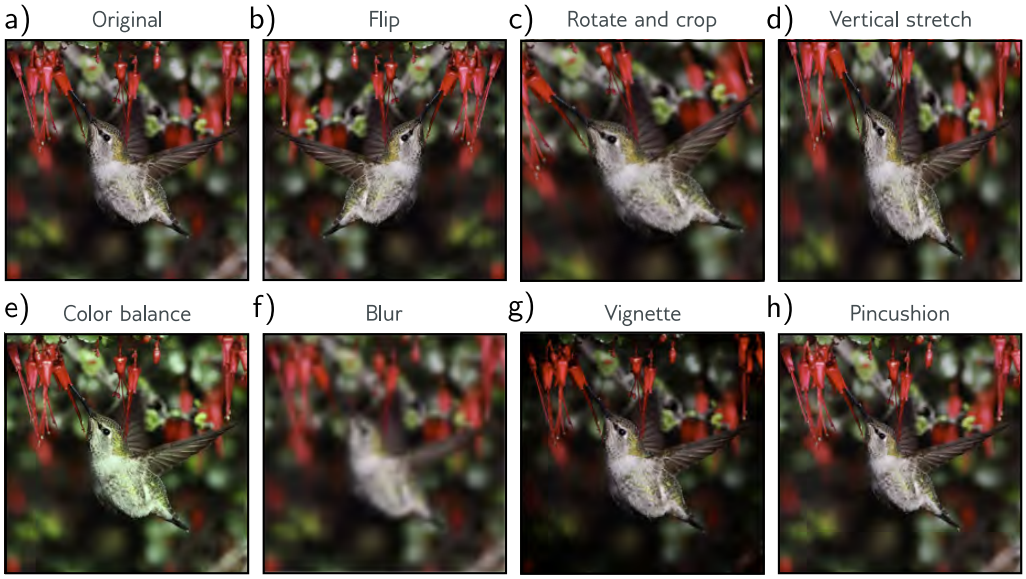


Figure 9.13 Data augmentation. For some problems, each data example can be transformed to augment the dataset. a) Original image. b–h) Various geometric and photometric transformations of this image. For image classification, all these images still have the same label, "bird." Adapted from Wu et al. (2015a).

Generating extra training data in this way is known as *data augmentation*. The aim is to teach the model to be indifferent to these irrelevant data transformations.

9.4 Summary

Explicit regularization involves adding an extra term to the loss function that changes the position of the minimum. The term can be interpreted as a prior probability over the parameters. Stochastic gradient descent with a finite step size does not neutrally descend to the minimum of the loss function. This bias can be interpreted as adding additional terms to the loss function, and this is known as implicit regularization.

There are also many heuristics for improving generalization, including early stopping, dropout, ensembling, the Bayesian approach, adding noise, transfer learning, multi-task learning, and data augmentation. There are four main principles behind these methods (figure 9.14). We can (i) encourage the function to be smoother (e.g., L2 regularization), (ii) increase the amount of data (e.g., data augmentation), (iii) combine models (e.g., ensembling), or (iv) search for wider minima (e.g., applying noise to network weights).

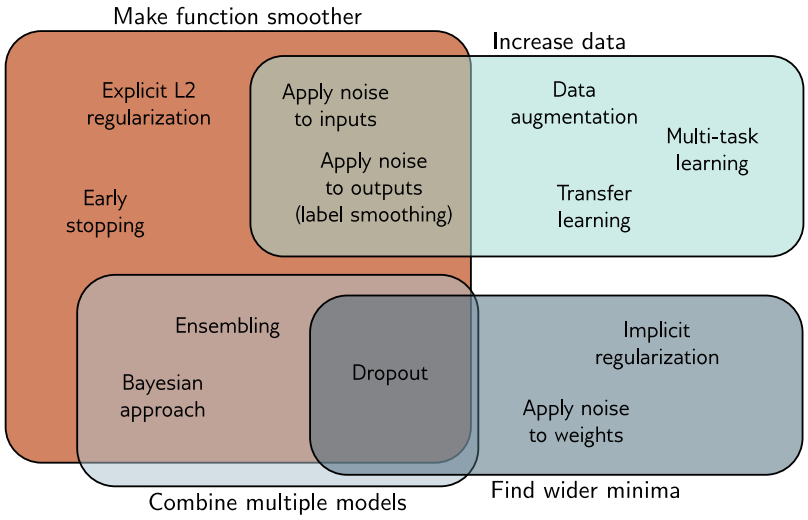


Figure 9.14 Regularization methods. The regularization methods discussed in this chapter aim to improve generalization by one of four mechanisms. Some methods aim to make the modeled function smoother. Other methods increase the effective amount of data. The third group of methods combine multiple models and hence mitigate against uncertainty in the fitting process. Finally, the fourth group of methods encourages the training process to converge to a wide minimum where small errors in the estimated parameters are less important (see also figure 20.11).

Another way to improve generalization is to choose the model architecture to suit the task. For example, in image segmentation, we can share parameters within the model, so we don't need to independently learn what a tree looks like at every image location. Chapters 10–13 consider architectural variations designed for different tasks.

Notes

An overview and taxonomy of regularization techniques in deep learning can be found in Kukačka et al. (2017). Notably missing from the discussion in this chapter is BatchNorm (Szegedy et al., 2016) at its variants, which are described in chapter 11.

Regularization: L2 regularization penalizes the sum of squares of the network weights. This encourages the output function to change slowly (i.e., become smoother) and is the most used regularization term. It is sometimes referred to as Frobenius norm regularization as it penalizes the Frobenius norms of the weight matrices. It is often also mistakenly referred to as "weight decay," although this is a separate technique devised by Hanson & Pratt (1988) in which the parameters $\boldsymbol{\phi}$ are updated as:

$$\boldsymbol{\phi} \longleftarrow (1-\lambda')\boldsymbol{\phi} - \alpha\frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.13}$$

where, as usual, α is the learning rate, and L is the loss. This is identical to gradient descent, except that the weights are reduced by a factor of $1-\lambda'$ before the gradient update. For standard SGD, weight decay is equivalent to L2 regularization (equation 9.5) with coefficient $\lambda = \lambda'/2\alpha$. However, for Adam, the learning rate α is different for each parameter, so L2 regularization and weight decay differ. Loshchilov & Hutter (2019) present AdamW, which modifies Adam to implement weight decay correctly and show that this improves performance.

Problem 9.5

Other choices of vector norm encourage sparsity in the weights. The L0 regularization term applies a fixed penalty for every non-zero weight. The effect is to "prune" the network. L0 regularization can also be used to encourage group sparsity; this might apply a fixed penalty if any of the weights contributing to a given hidden unit are non-zero. If they are all zero, we can remove the unit, decreasing the model size and making inference faster.

Appendix B.3.2 Vector norms

Unfortunately, L0 regularization is challenging to implement since the derivative of the regularization term is not smooth, and more sophisticated fitting methods are required (see Louizos et al., 2018). Somewhere between L2 and L0 regularization is L1 regularization or *LASSO* (least absolute shrinkage and selection operator), which imposes a penalty on the absolute values of the weights. L2 regularization somewhat discourages sparsity in that the derivative of the squared penalty decreases as the weight becomes smaller, lowering the pressure to make it smaller still. L1 regularization does not have this disadvantage, as the derivative of the penalty is constant. This can produce sparser solutions than L2 regularization but is much easier to optimize than L0 regularization. Sometimes both L1 and L2 regularization terms are used, which is termed an *elastic net* penalty (Zou & Hastie, 2005).

Problem 9.6

A different approach to regularization is to modify the gradients of the learning algorithm without ever explicitly formulating a new loss function (e.g., equation 9.13). This approach has been used to promote sparsity during backpropagation (Schwarz et al., 2021).

The evidence on the effectiveness of explicit regularization is mixed. Zhang et al. (2017a) showed that L2 regularization contributes little to generalization. It has been proven that the Lipschitz constant of the network (how fast the function can change as we modify the input) bounds the generalization error (Bartlett et al., 2017; Neyshabur et al., 2018). However, the Lipschitz constant depends on the product of the spectral norms of the weight matrices $\boldsymbol{\Omega}_k$, which are only indirectly dependent on the magnitudes of the individual weights. Bartlett et al. (2017), Neyshabur et al. (2018), and Yoshida & Miyato (2017) all add terms that indirectly encourage the spectral norms to be smaller. Gouk et al. (2021) take a different approach and develop an algorithm that constrains the Lipschitz constant of the network to be below a particular value.

Appendix B.1.1 Lipschitz constant

Appendix B.3.7 Spectral norm

Implicit regularization in gradient descent: The gradient descent step is:

$$\boldsymbol{\phi}_1 = \boldsymbol{\phi}_0 + \alpha \cdot \mathbf{g}[\boldsymbol{\phi}_0], \tag{9.14}$$

where $\mathbf{g}[\boldsymbol{\phi}_0]$ is the negative of the gradient of the loss function, and α is the step size. As $\alpha \to 0$, the gradient descent process can be described by a differential equation:

$$\frac{d\boldsymbol{\phi}}{dt} = \mathbf{g}[\boldsymbol{\phi}]. \tag{9.15}$$

For typical step sizes α, the discrete and continuous versions converge to different solutions. We can use *backward error analysis* to find a correction $\mathbf{g}_1[\boldsymbol{\phi}]$ to the continuous version:

$$\frac{d\boldsymbol{\phi}}{dt} \approx \mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}] + \ldots, \tag{9.16}$$

so that it gives the same result as the discrete version.

Consider the first two terms of a Taylor expansion of the modified continuous solution $\boldsymbol{\phi}$ around initial position $\boldsymbol{\phi}_0$:

$$\begin{aligned}
\boldsymbol{\phi}[\alpha] &\approx \boldsymbol{\phi} + \alpha\frac{d\boldsymbol{\phi}}{dt} + \frac{\alpha^2}{2}\frac{d^2\boldsymbol{\phi}}{dt^2}\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha\left(\mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2}\left(\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\frac{d\boldsymbol{\phi}}{dt} + \alpha\frac{\partial\mathbf{g}_1[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\frac{d\boldsymbol{\phi}}{dt}\right)\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&= \boldsymbol{\phi} + \alpha\left(\mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2}\left(\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}] + \alpha\frac{\partial\mathbf{g}_1[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]\right)\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha\mathbf{g}[\boldsymbol{\phi}] + \alpha^2\left(\mathbf{g}_1[\boldsymbol{\phi}] + \frac{1}{2}\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]\right)\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0}, \qquad (9.17)
\end{aligned}$$

where in the second line, we have introduced the correction term (equation 9.16), and in the final line, we have removed terms of greater order than α^2.

Note that the first two terms on the right-hand side $\boldsymbol{\phi}_0 + \alpha\mathbf{g}[\boldsymbol{\phi}_0]$ are the same as the discrete update (equation 9.14). Hence, to make the continuous and discrete versions arrive at the same place, the third term on the right-hand side must equal zero, allowing us to solve for $\mathbf{g}_1[\boldsymbol{\phi}]$:

$$\mathbf{g}_1[\boldsymbol{\phi}] = -\frac{1}{2}\frac{\partial\mathbf{g}[\boldsymbol{\phi}]}{\partial\boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]. \qquad (9.18)$$

During training, the evolution function $\mathbf{g}[\boldsymbol{\phi}]$ is the negative of the gradient of the loss:

$$\begin{aligned}
\frac{d\boldsymbol{\phi}}{dt} &\approx \mathbf{g}[\boldsymbol{\phi}] + \alpha\mathbf{g}_1[\boldsymbol{\phi}] \\
&= -\frac{\partial L}{\partial\boldsymbol{\phi}} - \frac{\alpha}{2}\left(\frac{\partial^2 L}{\partial\boldsymbol{\phi}^2}\right)\frac{\partial L}{\partial\boldsymbol{\phi}}. \qquad (9.19)
\end{aligned}$$

This is equivalent to performing continuous gradient descent on the loss function:

$$L_{GD}[\boldsymbol{\phi}] = L[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial\boldsymbol{\phi}}\right\|^2, \qquad (9.20)$$

because the right-hand side of equation 9.19 is the derivative of that in equation 9.20.

This formulation of implicit regularization was developed by Barrett & Dherin (2021) and extended to stochastic gradient descent by Smith et al. (2021). Smith et al. (2020) and others have shown that stochastic gradient descent with small or moderate batch sizes outperforms full batch gradient descent on the test set, and this may in part be due to implicit regularization.

Relatedly, Jastrzębski et al. (2021) and Cohen et al. (2021) both show that using a large learning rate reduces the tendency of typical optimization trajectories to move to "sharper" parts of the loss function (i.e., where at least one direction has high curvature). This implicit regularization effect of large learning rates can be approximated by penalizing the trace of the Fisher Information Matrix, which is closely related to penalizing the gradient norm in equation 9.20 (Jastrzębski et al., 2021).

Early stopping: Bishop (1995) and Sjöberg & Ljung (1995) argued that early stopping limits the effective solution space that the training procedure can explore; given that the weights are initialized to small values, this leads to the idea that early stopping helps prevent the weights from getting too large. Goodfellow et al. (2016) show that under a quadratic approximation of the loss function with parameters initialized to zero, early stopping is equivalent to L2 regularization in gradient descent. The effective regularization weight λ is approximately $1/(\tau\alpha)$ where α is the learning rate, and τ is the early stopping time.

Ensembling: Ensembles can be trained using different random seeds (Lakshminarayanan et al., 2017), hyperparameters (Wenzel et al., 2020b), or even entirely different families of models. The models can be combined by averaging their predictions, weighting the predictions, or *stacking* (Wolpert, 1992), in which the results are combined using another machine learning model. Lakshminarayanan et al. (2017) showed that averaging the output of independently trained networks can improve accuracy, calibration, and robustness. Conversely, Frankle et al. (2020) showed that if we average together the weights to make one model, the network fails. Fort et al. (2019) compared ensembling solutions that resulted from different initializations with ensembling solutions that were generated from the same original model. For example, in the latter case, they consider exploring around the solution in a limited subspace to find other good nearby points. They found that both techniques provide complementary benefits but that genuine ensembling from different random starting points provides a bigger improvement.

Appendix B.3.6 Subspaces

An efficient way of ensembling is to combine models from intermediate stages of training. To this end, Izmailov et al. (2018) introduce *stochastic weight averaging*, in which the model weights are sampled at different time steps and averaged together. As the name suggests, *snapshot ensembles* (Huang et al., 2017a) also store the models from different time steps and average their predictions. The diversity of these models can be improved by cyclically increasing and decreasing the learning rate. Garipov et al. (2018) observed that different minima of the loss function are often connected by a low-energy path (i.e., a path with a low loss everywhere along it). Motivated by this observation, they developed a method that explores low-energy regions around an initial solution to provide diverse models without retraining. This is known as *fast geometric ensembling*. A review of ensembling methods can be found in Ganaie et al. (2022).

Dropout: Dropout was first introduced by Hinton et al. (2012b) and Srivastava et al. (2014). Dropout is applied at the level of hidden units. Dropping a hidden unit has the same effect as temporarily setting all the incoming and outgoing weights and the bias to zero. Wan et al. (2013) generalized dropout by randomly setting individual weights to zero. Gal & Ghahramani (2016) and Kendall & Gal (2017) proposed Monte Carlo dropout, in which inference is computed with several dropout patterns, and the results are averaged together. Gal & Ghahramani (2016) argued that this could be interpreted as approximating Bayesian inference.

Dropout is equivalent to applying multiplicative Bernoulli noise to the hidden units. Similar benefits derive from using other distributions, including the normal (Srivastava et al., 2014; Shen et al., 2017), uniform (Shen et al., 2017), and beta distributions (Liu et al., 2019b).

Adding noise: Bishop (1995) and An (1996) added Gaussian noise to the network inputs to improve performance. Bishop (1995) showed that this is equivalent to weight decay. An (1996) also investigated adding noise to the weights. DeVries & Taylor (2017a) added Gaussian noise to the hidden units. The *randomized ReLU* (Xu et al., 2015) applies noise in a different way by making the activation functions stochastic.

Label smoothing: Label smoothing was introduced by Szegedy et al. (2016) for image classification but has since been shown to be helpful in speech recognition (Chorowski & Jaitly, 2017), machine translation (Vaswani et al., 2017), and language modeling (Pereyra et al., 2017). The precise mechanism by which label smoothing improves test performance isn't well understood, although Müller et al. (2019a) show that it improves the calibration of the predicted output probabilities. A closely related technique is *DisturbLabel* (Xie et al., 2016), in which a certain percentage of the labels in each batch are randomly switched at each training iteration.

Finding wider minima: It is thought that wider minima generalize better (see figure 20.11). Here, the exact values of the weights are less important, so performance should be robust to errors in their estimates. One of the reasons that applying noise to parts of the network during training is effective is that it encourages the network to be indifferent to their exact values.

Chaudhari et al. (2019) developed a variant of SGD that biases the optimization toward flat minima, which they call *entropy SGD*. The idea is to incorporate local entropy as a term in the loss function. In practice, this takes the form of one SGD-like update within another. Keskar

et al. (2017) showed that SGD finds wider minima as the batch size is reduced. This may be because of the batch variance term that results from implicit regularization by SGD.

Ishida et al. (2020) use a technique named *flooding*, in which they intentionally prevent the training loss from becoming zero. This encourages the solution to perform a random walk over the loss landscape and drift into a flatter area with better generalization.

Bayesian approaches: For some models, including the simplified neural network model in figure 9.11, the Bayesian predictive distribution can be computed in closed form (see Bishop, 2006; Prince, 2012). For neural networks, the posterior distribution over the parameters cannot be represented in closed form and must be approximated. The two main approaches are variational Bayes (Hinton & van Camp, 1993; MacKay, 1995; Barber & Bishop, 1997; Blundell et al., 2015), in which the posterior is approximated by a simpler tractable distribution, and Markov Chain Monte Carlo (MCMC) methods, which approximate the distribution by drawing a set of samples (Neal, 1995; Welling & Teh, 2011; Chen et al., 2014; Ma et al., 2015; Li et al., 2016a). The generation of samples can be integrated into SGD, and this is known as stochastic gradient MCMC (see Ma et al., 2015). It has recently been discovered that "cooling" the posterior distribution over the parameters (making it sharper) improves predictions from these models (Wenzel et al., 2020a), but this is not currently fully understood (see Noci et al., 2021).

Transfer learning: Transfer learning for visual tasks works extremely well (Sharif Razavian et al., 2014) and has supported rapid progress in computer vision, including the original AlexNet results (Krizhevsky et al., 2012). Transfer learning has also impacted natural language processing (NLP), where many models are based on pre-trained features from the BERT model (Devlin et al., 2019). More information can be found in Zhuang et al. (2020) and Yang et al. (2020b).

Self-supervised learning: Self-supervised learning techniques for images have included inpainting masked image regions (Pathak et al., 2016), predicting the relative position of patches in an image (Doersch et al., 2015), re-arranging permuted image tiles back into their original configuration (Noroozi & Favaro, 2016), colorizing grayscale images (Zhang et al., 2016b), and transforming rotated images back to their original orientation (Gidaris et al., 2018). In SimCLR (Chen et al., 2020c), a network is learned that maps versions of the same image that have been photometrically and geometrically transformed to the same representation while repelling versions of different images, with the goal of becoming indifferent to irrelevant image transformations. Jing & Tian (2020) present a survey of self-supervised learning in images.

Self-supervised learning in NLP can be based on predicting masked words(Devlin et al., 2019), predicting the next word in a sentence (Radford et al., 2019; Brown et al., 2020), or predicting whether two sentences follow one another (Devlin et al., 2019). In automatic speech recognition, the Wav2Vec model (Schneider et al., 2019) aims to distinguish an original audio sample from one where 10ms of audio has been swapped out from elsewhere in the clip. Self-supervision has also been applied to graph neural networks (chapter 13). Tasks include recovering masked features (You et al., 2020) and recovering the adjacency structure of the graph (Kipf & Welling, 2016). Liu et al. (2023a) review self-supervised learning for graph models.

Data augmentation: Data augmentation for images dates back to at least LeCun et al. (1998) and contributed to the success of AlexNet (Krizhevsky et al., 2012), in which the dataset was increased by a factor of 2048. Image augmentation approaches include geometric transformations, changing or manipulating the color space, noise injection, and applying spatial filters. More elaborate techniques include randomly mixing images (Inoue, 2018; Summers & Dinneen, 2019), randomly erasing parts of the image (Zhong et al., 2020), style transfer (Jackson et al., 2019), and randomly swapping image patches (Kang et al., 2017). In addition, many studies have used generative adversarial networks or GANs (see chapter 15) to produce novel but plausible data examples (e.g., Calimeri et al., 2017). In other cases, the data have been augmented with adversarial examples (Goodfellow et al., 2015a), which are minor perturbations of the training data that cause the example to be misclassified. A review of data augmentation for images can be found in Shorten & Khoshgoftaar (2019).

Augmentation methods for acoustic data include pitch shifting, time stretching, dynamic range compression, and adding random noise (e.g., Abeßer et al., 2017; Salamon & Bello, 2017; Xu et al., 2015; Lasseck, 2018), as well as mixing data pairs (Zhang et al., 2017c; Yun et al., 2019), masking features (Park et al., 2019), and using GANs to generate new data (Mun et al., 2017). Augmentation for speech data includes vocal tract length perturbation (Jaitly & Hinton, 2013; Kanda et al., 2013), style transfer (Gales, 1998; Ye & Young, 2004), adding noise (Hannun et al., 2014), and synthesizing speech (Gales et al., 2009).

Augmentation methods for text include adding noise at a character level by switching, deleting, and inserting letters (Belinkov & Bisk, 2018; Feng et al., 2020), or by generating adversarial examples (Ebrahimi et al., 2018), using common spelling mistakes (Coulombe, 2018), randomly swapping or deleting words (Wei & Zou, 2019), using synonyms (Kolomiyets et al., 2011), altering adjectives (Li et al., 2017c), passivization (Min et al., 2020), using generative models to create new data (Qiu et al., 2020), and round-trip translation to another language and back (Aiken & Park, 2010). Augmentation methods for text are reviewed by Bayer et al. (2022).

Problems

Problem 9.1 Consider a model where the prior distribution over the parameters is a normal distribution with mean zero and variance $\sigma_{\boldsymbol{\phi}}^2$ so that

$$Pr(\boldsymbol{\phi}) = \prod_{j=1}^{J} \text{Norm}_{\phi_j}[0, \sigma_{\boldsymbol{\phi}}^2], \tag{9.21}$$

where j indexes the model parameters. We now maximize $\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi})Pr(\boldsymbol{\phi})$. Show that the associated loss function of this model is equivalent to L2 regularization.

Problem 9.2 How do the gradients of the loss function change when L2 regularization (equation 9.5) is added?

Problem 9.3* Consider a linear regression model $y = \phi_0 + \phi_1 x$ with input x, output y, and parameters ϕ_0 and ϕ_1. Assume we have I training examples $\{x_i, y_i\}$ and use a least squares loss. Consider adding Gaussian noise with mean zero and variance σ_x^2 to the inputs x_i at each training iteration. What is the expected gradient update?

Problem 9.4* Derive the loss function for multiclass classification when we use label smoothing so that the target probability distribution has 0.9 at the correct class and the remaining probability mass of 0.1 is divided between the remaining $D_o - 1$ classes.

Problem 9.5 Show that the weight decay parameter update with decay rate λ:

$$\boldsymbol{\phi} \longleftarrow (1-\lambda)\boldsymbol{\phi} - \alpha\frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.22}$$

on the original loss function $L[\boldsymbol{\phi}]$ is equivalent to a standard gradient update using L2 regularization so that the modified loss function $\tilde{L}[\boldsymbol{\phi}]$ is:

$$\tilde{L}[\boldsymbol{\phi}] = L[\boldsymbol{\phi}] + \frac{\lambda}{2\alpha}\sum_{k}\phi_k^2, \tag{9.23}$$

where $\boldsymbol{\phi}$ are the parameters, and α is the learning rate.

Problem 9.6 Consider a model with parameters $\boldsymbol{\phi} = [\boldsymbol{\phi}_0, \boldsymbol{\phi}_1]^T$. Draw the L0, $L\frac{1}{2}$, and L1 regularization terms in a similar form to figure 9.1b. The LP regularization term is $\sum_{d=1}^{D}|\phi_d|^P$.

Chapter 10

Convolutional networks

Chapters 2–9 introduced the supervised learning pipeline for deep neural networks. However, these chapters only considered fully connected networks with a single path from input to output. Chapters 10–13 introduce more specialized network components with sparser connections, shared weights, and parallel processing paths. This chapter describes *convolutional layers*, which are mainly used for processing image data.

Images have three properties that suggest the need for specialized model architecture. First, they are high-dimensional. A typical image for a classification task contains 224×224 RGB values (i.e., 150,528 input dimensions). Hidden layers in fully connected networks are generally larger than the input size, so even for a shallow network, the number of weights would exceed $150,528^2$, or 22 billion. This poses obvious practical problems in terms of the required training data, memory, and computation.

Second, nearby image pixels are statistically related. However, fully connected networks have no notion of "nearby" and treat the relationship between every input equally. If the pixels of the training and test images were randomly permuted in the same way, the network could still be trained with no practical difference. Third, the interpretation of an image is stable under geometric transformations. An image of a tree is still an image of a tree if we shift it leftwards by a few pixels. However, this shift changes every input to the network. Hence, a fully connected model must learn the patterns of pixels that signify a tree separately at every position, which is clearly inefficient.

Convolutional layers process each local image region independently, using parameters shared across the whole image. They use fewer parameters than fully connected layers, exploit the spatial relationships between nearby pixels, and don't have to re-learn the interpretation of the pixels at every position. A network predominantly consisting of convolutional layers is known as a *convolutional neural network* or *CNN*.

10.1 Invariance and equivariance

We argued above that some properties of images (e.g., tree texture) are stable under transformations. In this section, we make this idea more mathematically precise. A

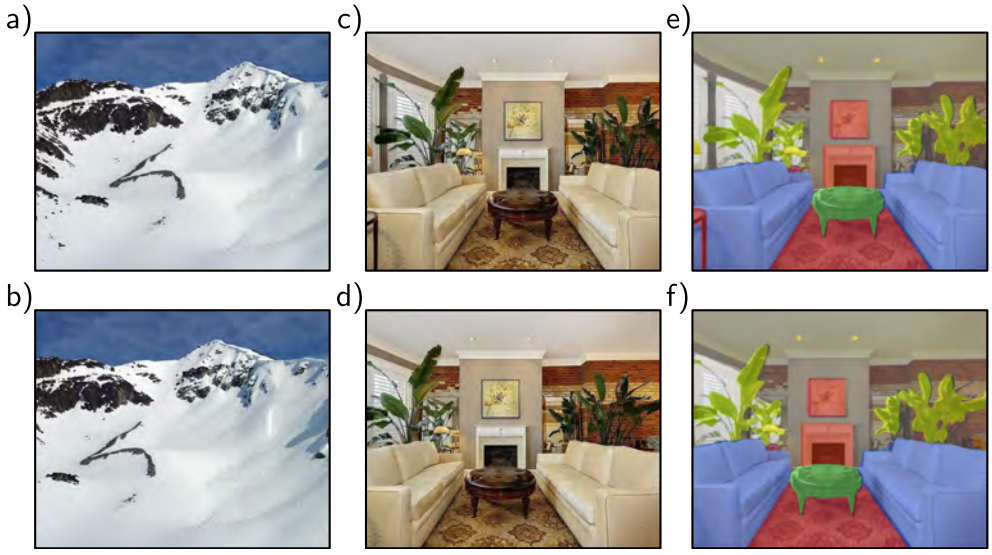


Figure 10.1 Invariance and equivariance for translation. a–b) In image classification, the goal is to categorize both images as "mountain" regardless of the horizontal shift that has occurred. In other words, we require the network prediction to be invariant to translation. c,e) The goal of semantic segmentation is to associate a label with each pixel. d,f) When the input image is translated, we want the output (colored overlay) to translate in the same way. In other words, we require the output to be equivariant with respect to translation. Panels c–f) adapted from Bousselham et al. (2021).

function $\mathbf{f}[\mathbf{x}]$ of an image $\mathbf{x}$ is *invariant* to a transformation $\mathbf{t}[\mathbf{x}]$ if:

$$\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{f}[\mathbf{x}]. \tag{10.1}$$

In other words, the output of the function $\mathbf{f}[\mathbf{x}]$ is the same regardless of the transformation $\mathbf{t}[\mathbf{x}]$. Networks for image classification should be invariant to geometric transformations of the image (figure 10.1a–b). The network $\mathbf{f}[\mathbf{x}]$ should identify an image as containing the same object, even if it has been translated, rotated, flipped, or warped.

A function $\mathbf{f}[\mathbf{x}]$ of an image $\mathbf{x}$ is *equivariant* or *covariant* to a transformation $\mathbf{t}[\mathbf{x}]$ if:

$$\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{t}\big[\mathbf{f}[\mathbf{x}]\big]. \tag{10.2}$$

In other words, $\mathbf{f}[\mathbf{x}]$ is equivariant to the transformation $\mathbf{t}[\mathbf{x}]$ if its output changes in the same way under the transformation as the input. Networks for per-pixel image segmentation should be equivariant to transformations (figure 10.1c–f); if the image is translated, rotated, or flipped, the network $\mathbf{f}[\mathbf{x}]$ should return a segmentation that has been transformed in the same way.

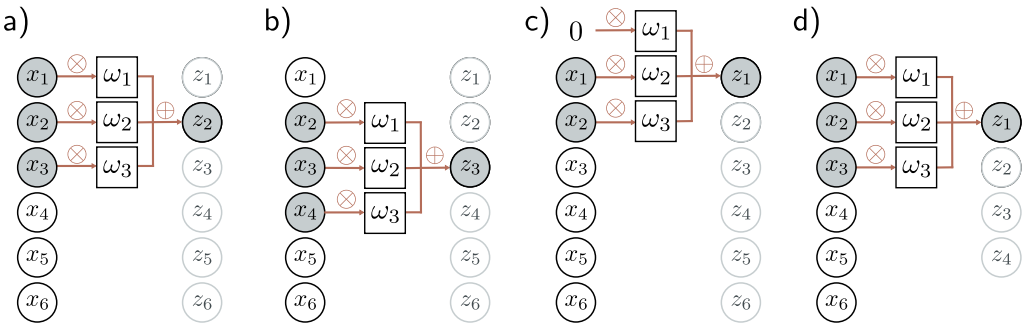


Figure 10.2 1D convolution with kernel size three. Each output z_i is a weighted sum of the nearest three inputs x_{i-1}, x_i, and x_{i+1}, where the weights are $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]$. a) Output z_2 is computed as $z_2 = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$. b) Output z_3 is computed as $z_3 = \omega_1 x_2 + \omega_2 x_3 + \omega_3 x_4$. c) At position z_1, the kernel extends beyond the first input x_1. This can be handled by zero padding, in which we assume values outside the input are zero. The final output is treated similarly. d) Alternatively, we could only compute outputs where the kernel fits within the input range ("valid" convolution); now, the output will be smaller than the input.

10.2 Convolutional networks for 1D inputs

Convolutional networks consist of a series of convolutional layers, each of which is equivariant to translation. They also typically include pooling mechanisms that induce partial invariance to translation. For clarity of exposition, we first consider convolutional networks for 1D data, which are easier to visualize. In section 10.3, we progress to 2D convolution, which can be applied to image data.

10.2.1 1D convolution operation

Convolutional layers are network layers based on the *convolution* operation. In 1D, a convolution transforms an input vector $\mathbf{x}$ into an output vector $\mathbf{z}$ so that each output z_i is a weighted sum of nearby inputs. The same weights are used at every position and are collectively called the *convolution kernel* or *filter*. The size of the region over which inputs are combined is termed the *kernel size*. For a kernel size of three, we have:

$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}, \tag{10.3}$$

where $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]^T$ is the kernel (figure 10.2).[1] Notice that the convolution operation is equivariant with respect to translation. If we translate the input x, then the corresponding output z is translated in the same way.

Problem 10.1

[1]Strictly speaking, this is a cross-correlation and not a convolution, in which the weights would be flipped relative to the input (so we would switch x_{i-1} with x_{i+1}). Regardless, this (incorrect) definition is the usual convention in machine learning.

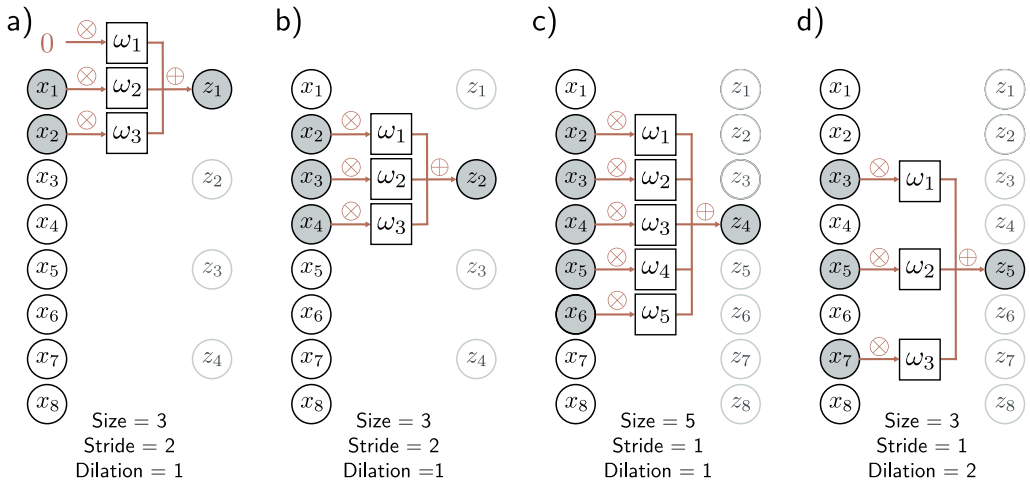


Figure 10.3 Stride, kernel size, and dilation. a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can also be changed. With a kernel size of five, we take a weighted sum of the nearest five inputs. d) In dilated or atrous convolution (from the French "à trous" – with holes), we intersperse zeros in the weight vector to allow us to combine information over a large area using fewer weights.

10.2.2 Padding

Equation 10.3 shows that each output is computed by taking a weighted sum of the previous, current, and subsequent positions in the input. This begs the question of how to deal with the first output (where there is no previous input) and the final output (where there is no subsequent input).

There are two common approaches. The first is to pad the edges of the inputs with new values and proceed as usual. *Zero padding* assumes the input is zero outside its valid range (figure 10.2c). Other possibilities include treating the input as circular or reflecting it at the boundaries. The second approach is to discard the output positions where the kernel exceeds the range of input positions. These *valid convolutions* have the advantage of introducing no extra information at the edges of the input. However, they have the disadvantage that the representation decreases in size.

10.2.3 Stride, kernel size, and dilation

In the example above, each output was a sum of the nearest three inputs. However, this is just one of a larger family of convolution operations, the members of which are

distinguished by their *stride*, *kernel size*, and *dilation rate*. When we evaluate the output at every position, we term this a *stride* of one. However, it is also possible to shift the kernel by a stride greater than one. If we have a stride of two, we create roughly half the number of outputs (figure 10.3a–b).

The *kernel size* can be increased to integrate over a larger area (figure 10.3c). However, it typically remains an odd number so that it can be centered around the current position. Increasing the kernel size has the disadvantage of requiring more weights. This leads to the idea of *dilated* or *atrous* convolutions, in which the kernel values are interspersed with zeros. For example, we can turn a kernel of size five into a dilated kernel of size three by setting the second and fourth elements to zero. We still integrate information from a larger input region but only require three weights to do this (figure 10.3d). The number of zeros we intersperse between the weights determines the *dilation rate*.

Problems 10.2–10.4

10.2.4 Convolutional layers

A convolutional layer computes its output by convolving the input, adding a bias β, and passing each result through an activation function $\mathrm{a}[\bullet]$. With kernel size three, stride one, and dilation rate one, the i^{th} hidden unit h_i would be computed as:

$$\begin{aligned} h_i &= \mathrm{a}\left[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}\right] \\ &= \mathrm{a}\left[\beta + \sum_{j=1}^{3} \omega_j x_{i+j-2}\right], \end{aligned} \tag{10.4}$$

where the bias β and kernel weights $\omega_1, \omega_2, \omega_3$ are trainable parameters, and (with zero padding) we treat the input x as zero when it is out of the valid range. This is a special case of a fully connected layer that computes the i^{th} hidden unit as:

$$h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right]. \tag{10.5}$$

If there are D inputs $x_\bullet$ and D hidden units $h_\bullet$, this fully connected layer would have D^2 weights $\omega_{\bullet\bullet}$ and D biases $\beta_\bullet$. The convolutional layer only uses three weights and one bias. A fully connected layer can reproduce this exactly if most weights are set to zero and others are constrained to be identical (figure 10.4).

Problem 10.5

10.2.5 Channels

If we only apply a single convolution, information will inevitably be lost; we are averaging nearby inputs, and the ReLU activation function clips results that are less than zero. Hence, it is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, termed a *feature map* or *channel*.

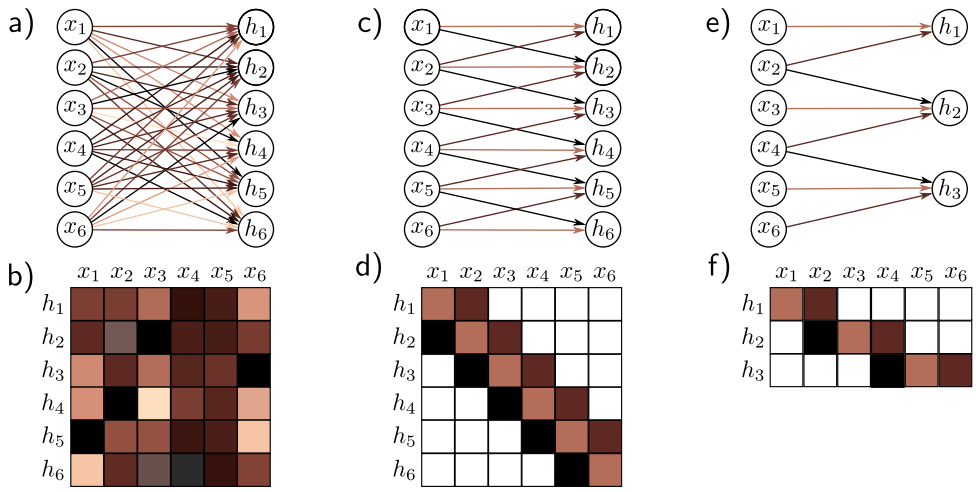


Figure 10.4 Fully connected vs. convolutional layers. a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown). b) Hence, the associated weight matrix $\mathbf{\Omega}$ contains 36 weights relating the six inputs to the six hidden units. c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown). d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate same value, white indicates zero weight). e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position. f) This is also a special case of a fully connected network with a different sparse weight structure.

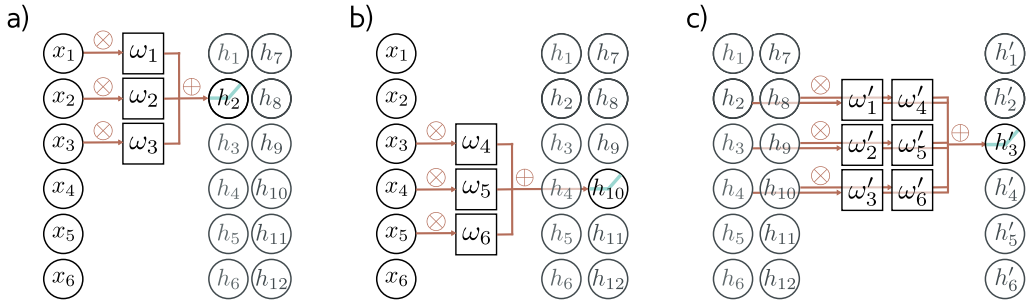


Figure 10.5 Channels. Typically, multiple convolutions are applied to the input $\mathbf{x}$ and stored in channels. a) A convolution is applied to create hidden units h_1 to h_6, which form the first channel. b) A second convolution operation is applied to create hidden units h_7 to h_{12}, which form the second channel. The channels are stored in a 2D array $\mathbf{H}_1$ that contains all the hidden units in the first hidden layer. c) If we add a further convolutional layer, there are now two channels at each input position. Here, the 1D convolution defines a weighted sum over both input channels at the three closest positions to create each new output channel.