NVIDIA. DEEP LEARNING INSTITUTE

LEARNING DEEP LEARNING

Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow

MAGNUS EKMAN

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Learning Deep Learning

This page intentionally left blank

Learning Deep Learning

THEORY AND PRACTICE OF NEURAL NETWORKS, COMPUTER VISION, NATURAL LANGUAGE PROCESSING, AND TRANSFORMERS USING TENSORFLOW

MAGNUS EKMAN

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

NVIDIA makes no warranty or representation that the techniques described herein are free from any Intellectual Property claims. The reader assumes all risk of any such claims based on his or her use of these techniques.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021937264

Copyright © 2022 NVIDIA Corporation

Cover image: R.Eva Robot design by Gavriil Klimov and Gregor Kopka

Figures P-4, 8-8, 8-10, 16-1, 16-6, 16-7, B-1, J-1, J-2: Magnus Ekman

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-747035-8
ISBN-10: 0-13-747035-5

ScoutAutomatedPrintCode

For my wife Jennifer, my children Sebastian and Sofia, my dog Babette, and my parents Ingrid and Krister

This page intentionally left blank

Contents

Foreword

Artificial intelligence (AI) has seen impressive progress over the last decade. Humanity's dream of building intelligent machines that can think and act like us, only better and faster, seems to be finally taking off. To enable everyone to be part of this historic revolution requires the democratization of AI knowledge and resources. This book is timely and relevant toward accomplishing these lofty goals.

Learning Deep Learning by Magnus Ekman provides a comprehensive instructional guide for both aspiring and experienced AI engineers. In the book, Magnus shares the rich hands-on knowledge he has garnered at NVIDIA, an established leader in AI. The book does not assume any background in machine learning and is focused on covering significant breakthroughs in deep learning over the last few years. The book strikes a nice balance and covers both important fundamentals such as backpropagation and the latest models in several domains (e.g., GPT for language understanding, Mask R-CNN for image understanding).

AI is a trinity of data, algorithms, and computing infrastructure. The launch of the ImageNet challenge provided a large-scale benchmark dataset needed to train large neural networks. The parallelism of NVIDIA GPUs enabled the training of such large neural networks. We are now in the era of billion, and even trillion, parameter models. Building and maintaining large-scale models will soon be deemed a prerequisite skill for any AI engineer. This book is uniquely placed to teach such skills. It provides in-depth coverage of large-scale models in multiple domains.

The book also covers emerging areas such as neural architecture search, which will likely become more prevalent as we begin to extract the last ounce of accuracy and hardware efficiency out of current AI models. The deep learning revolution has almost entirely occurred in open source. This book provides convenient access to code and datasets and runs through the code examples thoroughly. There is extensive program code available in both TensorFlow and PyTorch, the two most popular frameworks for deep learning.

I do not think any book on AI will be complete without a discussion of ethical issues. I believe that it is the responsibility of every AI engineer to think critically about the societal implications around the deployment of AI. The proliferation of harassment, hate speech, and misinformation in social media has shown how poorly designed algorithms can wreak havoc on our society. Groundbreaking studies such as the Gender Shades project and Stochastic Parrots have shown highly problematic biases in AI models that are commercially deployed at scale. I have advocated for banning the use of AI in sensitive scenarios until appropriate guidelines and testing are in place (e.g., the use of AI-based face recognition by law enforcement). I am glad to see the book cover significant developments such as model cards that improve accountability and transparency in training and maintaining AI models. I am hoping for a bright, inclusive future for the AI community.

—Dr. Anima Anandkumar
Bren Professor, Caltech
Director of ML Research, NVIDIA

Foreword

By training I am an economist. Prior to my work in technical education, I spent years teaching students and professionals well-developed frameworks for understanding our world and how to make decisions within it. The methods and skills you will discover in *Learning Deep Learning* by Magnus Ekman parallel the tools used by economists to make forecasts and predictions in a world full of uncertainty. The power and capabilities of the deep learning techniques taught in this book have brought amazing advances in our ability to make better predictions and inferences from the data in the world around us.

Though their future benefits and importance can sometimes be exaggerated, there is no doubt the world and industry have been greatly affected by deep learning (DL) and its related supersets of machine learning (ML) and artificial intelligence (AI). Applications of these technologies have proven durable and are profound. They are with us everywhere: at home and at work, in our cars, and on our phones. They influence how we travel, how we communicate, how we shop, how we bank, and how we access information. It is very difficult to think of an industry that has not or will not be impacted by these technologies.

The explosion in the use of these technologies has uncovered two important gaps in knowledge and areas of opportunity for those who endeavor to learn. First is the technical skillset required to develop useful applications. And second, importantly, is an understanding of how these applications can address problems and opportunities in the world around us. This book helps to address both gaps. For these reasons, *Learning Deep Learning* has arrived in the right place at the right time.

As NVIDIA's education and training arm, the Deep Learning Institute exists to help individuals and organizations grow their understanding of DL and other computing techniques so they can find creative solutions to challenging problems. *Learning Deep Learning* is the perfect addition to our training library. It is accessible to those with basic skills in statistics and calculus, and it doesn't require the reader to first wade through tangential topics. Instead, Ekman focuses on the building blocks of DL: the perceptron, other artificial neurons, deep neural networks (DNNs), and DL frameworks. Then he gradually layers in additional concepts that build on each other, all the way up to and including modern natural language processing (NLP) architectures such as Transformer, BERT, and GPT.

Importantly, Ekman uses a learning technique that in our experience has proven pivotal to success—asking readers to think about using DL techniques in practice. Simple yet powerful coding examples and exercises are provided throughout the book to help readers apply their understanding. At the same time, explanations of the underlying theory are present, and those interested in deepening their knowledge of relevant concepts and tools without getting into programming code will benefit. Plenty of citations with references for further study of a specific topic are also provided.

For all these reasons, *Learning Deep Learning* is a very good place to start one's journey to understanding the world of DL. Ekman's straightforward approach to helping the reader understand what DL is, how it was developed, and how it can be applied in our ever-changing world is refreshing. He provides a comprehensive yet clear discussion of the technology and an honest assessment of its capabilities and its limitations. And through it all, he permits the reader to dream, just a bit, about where DL may yet take us. That is exciting. It is why this economist finds this book so timely and important, and why I think you will too.

—Dr. Craig Clawson
Director, NVIDIA Deep Learning Institute

Preface

Deep learning (DL) is a quickly evolving field, which has demonstrated amazing results in performing tasks that traditionally have been performed well only by humans. Examples of such tasks are image classification, generating natural language descriptions of images, natural language translation, speech-to-text, and text-to-speech conversion.

Learning Deep Learning (this book, hereafter known as LDL) quickly brings you up to speed on the topic. It teaches how DL works, what it can do, and gives you some practical experience, with the overall objective of giving you a solid foundation for further learning.

In this book, we use green text boxes like this one to highlight concepts that we find extra important. The intent is to ensure that you do not miss key concepts. Let us begin by pointing out that we find **Deep Learning** important.

You will learn about the perceptron and other artificial neurons. They are the fundamental building blocks of deep neural networks that have enabled the DL revolution. You will learn about fully connected feedforward networks and convolutional networks. You will apply these networks to solve practical problems, such as predicting housing prices based on a large number of variables or identifying to which category an image belongs. Figure P-1 shows examples of such categories and images.

You will also learn about ways to represent words from a natural language using an encoding that captures some of the semantics of the encoded words. You will then use these encodings together with a recurrent neural network to create a neural-based natural language translator. This translator can automatically translate simple sentences from English to French or other similar languages, as illustrated in Figure P-2.

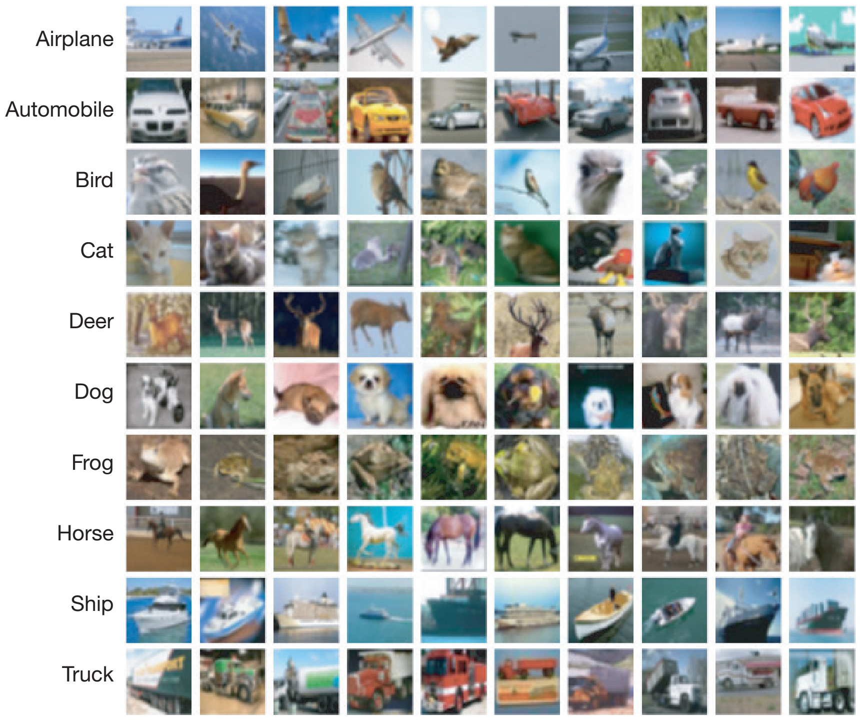

Figure P-1 Categories and example images from the CIFAR-10 dataset (Krizhevsky, 2009). This dataset will be studied in more detail in Chapter 7. (Image source: https://www.cs.toronto.edu/~kriz/cifar.html)

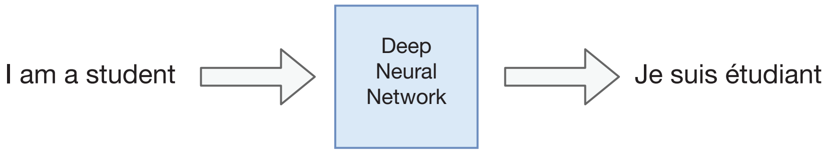

Figure P-2 A neural network translator that takes a sentence in English as input and produces the corresponding sentence in French as output

Finally, you will learn how to build an image-captioning network that combines image and language processing. This network takes an image as an input and automatically generates a natural language description of the image.

What we just described represents the main narrative of LDL. Throughout this journey, you will learn many other details. In addition, we end with a medley of additional important topics. We also provide appendixes that dive deeper into a collection of the discussed topics.

What Is Deep Learning?

We do not know of a crisp definition of what DL is, but one attempt is that *DL is a class of machine learning algorithms that use multiple layers of computational units where each layer learns its own representation of the input data. These representations are combined by later layers in a hierarchical fashion*. This definition is somewhat abstract, especially given that we have not yet described the concept of layers and computational units, but in the first few chapters, we provide many more concrete examples of what this means.

A fundamental part of DL is the deep neural network (DNN), a namesake of the biological neuron, by which it is loosely inspired. There is an ongoing debate about how closely the techniques within DL do mimic activity in a brain, where one camp argues that using the term *neural* network paints the picture that it is more advanced than it is. Along those lines, they recommend using the terms *unit* instead of *artificial neuron* and just *network* instead of *neural network*. No doubt, DL and the larger field of artificial intelligence (AI) have been significantly hyped in mainstream media. At the time of writing this book, it is easy to get the impression that we are close to creating machines that think like humans, although lately, articles that express some doubt are more common. After reading this book, you will have a more accurate view of what kind of problems DL can solve. In this book, we choose to freely use the words *neural network* and *neuron* but recognize that the algorithms presented are more tied to machine capabilities than to how an actual human brain works.

In this book, we use red text boxes like this one when we feel the urge to state something that is somewhat beside the point, a subjective opinion or of similar nature. You can safely ignore these boxes altogether if you do not find them adding any value to your reading experience.

Let us dive into this book by stating the opinion that it is a little bit of a buzz killer to take the stance that our cool DNNs are not similar to the brain. This is especially true for somebody picking up this book after reading about machines with superhuman abilities in the mainstream media. To keep the illusion alive, we sometimes allow ourselves to dream a little bit and make analogies that are not necessarily that well founded, but to avoid misleading you, we try not to dream outside of the red box.

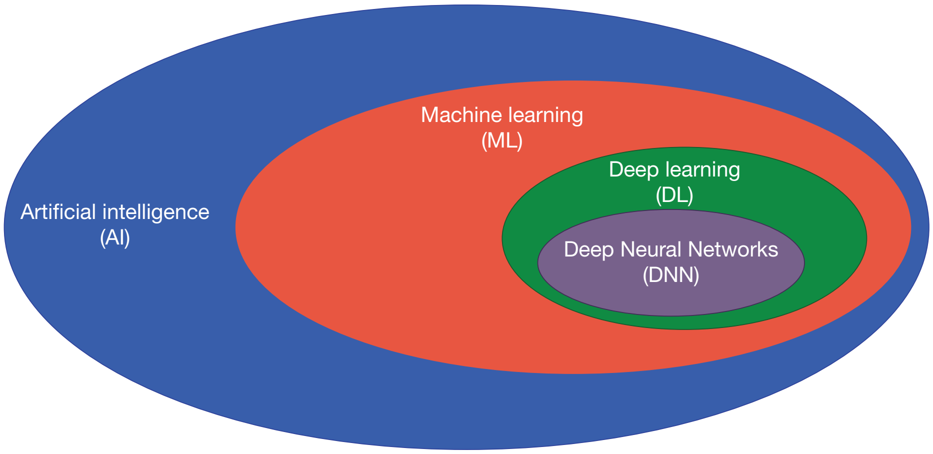

Figure P-3 Relationship between artificial intelligence, machine learning, deep learning, and deep neural networks. The sizes of the different ovals do not represent the relative size of one field compared to another.

To put DL and DNNs into context, Figure P-3 shows how they relate to the machine learning (ML) and AI fields. DNN is a subset of DL. DL in turn is a subset of the field of ML, which in turn is a subset of the greater field of AI.

Deep neural network (DNN) is a subset of DL.

DL is a subset of **machine learning (ML)**, which is a subset of **artificial intelligence (AI).**

In this book, we choose not to focus too much on the exact definition of DL and its boundaries, nor do we go into the details of other areas of ML or AI. Instead, we focus on details of what DNNs are and the types of tasks to which they can be applied.

Brief History of Deep Neural Networks

In the last couple of sections, we loosely referred to networks without describing what a network is. The first few chapters in this book discuss network architectures in detail, but at this point, it is sufficient to think of a network as an opaque system that has inputs and outputs. The usage model is to present something, for example, an image or a text sequence, as inputs to the network, and the network will produce something useful on its outputs, such as an interpretation of what the image contains, as in Figure P-4, or a natural language translation in a different language, as was shown in Figure P-2.

As previously mentioned, a central piece of a neural network is the artificial neuron. The first model of an artificial neuron was introduced in 1943 (McCulloch and Pitts, 1943), which started the first wave of neural network research. The McCulloch and Pitts neuron was followed in 1957 by the Rosenblatt perceptron (Rosenblatt, 1958). A key contribution from the perceptron was its associated automated learning algorithm that demonstrated how a system could learn desired behavior. Details of how the perceptron works are found in Chapter 1. The perceptron has some fundamental limitations, and although it was shown that these limitations can be overcome by combining multiple perceptrons into a multilayer network, the original learning algorithm did not extend to multilayer networks. According to a common narrative, this resulted in neural network research falling out of fashion. This is often referred to as the first AI winter, which was allegedly caused by a book by Minsky and Papert (1969). In this book, they raised the absence of a learning algorithm for multilayer networks as a serious concern.

We note that in the days of Rosenblatt's publications, they were certainly not shy about comparing their work with the human brain. In reading about the Rosenblatt perceptron (Rosenblatt, 1958), we see that the first paper he references is called "Design for a Brain."

This topic and narrative are controversial. Olazaran (1996) has studied whether the statements of Minsky and Papert had been misrepresented. Further, Schmidhuber (2015) pointed out that there did exist a learning algorithm for multilevel networks (Ivakhnenko and Lapa, 1965) four years before the book by Minsky and Papert was published.

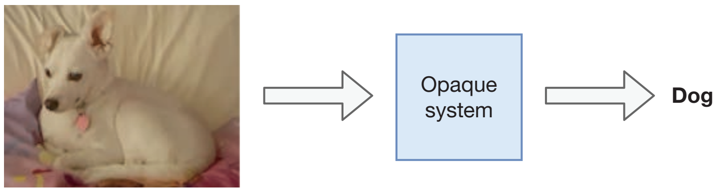

Figure P-4 A deep neural network as an opaque system that can take an image as an input and then output an indication of what type of object is in the image

The second wave of neural network research was initiated in the 1980s. It was heavily influenced by a paper that described the backpropagation algorithm for automatic training of multilayer networks (Rumelhart et al., 1986). Rumelhart and colleagues showed that this algorithm could be used to overcome the limitations of the perceptron. In the study, they explicitly pointed out that they believed this addressed the concerns raised by Minsky and Papert. Rumelhart and colleagues popularized the backpropagation algorithm in the context of neural networks, but it was not the first occurrence of the algorithm in the literature. The algorithm was applied to a similar problem domain in 1970 (Linnainmaa, 1970). Werbos (1981) described it in the context of neural networks in 1981.

Details of how this algorithm works are found in Chapter 3. An important outcome of this second wave of neural network research was the development of LeNet in 1989. It was a convolutional neural network (CNN), which was shown to be able to recognize handwritten zip codes (LeCun et al., 1990). It built on Fukushima's *Neocognitron* (Fukushima, 1980), which we believe is the first published CNN.

An enhanced version of LeNet was later used by major US banks to read handwritten checks, and it thereby became one of the first big commercial applications of neural networks. Convolutional neural networks are described in detail in Chapter 7. Despite the progress, neural networks fell out of fashion yet again, partly because the limited computational capability at the time prevented the networks from scaling to larger problems and partly because other traditional ML approaches were perceived as better alternatives.

The third wave of neural network research was enabled by a combination of algorithmic progress, availability of massive datasets, and the ability to use graphics processing units (GPU) for general purpose computing. From an outsider perspective, all this came together in 2012. At that point, the field had been rebranded as DL and was popularized in large part due to AlexNet (Krizhevsky et al., 2012), which was a CNN that scored significantly higher than any other participant in a computer vision competition known as the ImageNet challenge.

In reality, this third wave was enabled by persistent research groups who had continued to perform neural network research in the 1990s and first decade of the 2000s. These insiders started using the term *deep networks* in 2006. Further, the ImageNet challenge was not the first competition in which neural networks, some of which were GPU accelerated, beat more traditional techniques.

For example, Graves and colleagues (2009) won competitions in handwriting recognition with a neural network in 2009. Similarly, Ciresan and colleagues (2011) used a GPU accelerated network for image classification in 2011.

This work was shortly followed by similar breakthroughs in other fields, which have led to the DL boom that is still ongoing as of the writing of this book. The rest of this book will describe some of these key findings and how they can be applied in practice. For a more detailed description of the history of DL, we recommend Schmidhuber's (2015) overview.

Is This Book for You?

There are already many books on this topic, and different people like to approach subjects in different ways. In this book, we try to cut to the chase while still providing enough background to give you a warm fuzzy feeling that you understand why the techniques work. We decided to *not* start the book with an overall introduction to the field of traditional ML. Although we believe that anybody who wants to get serious about DL needs to also master traditional ML, we do not believe that it is necessary to first learn about traditional ML before learning the basics of DL. We even believe that having to first get through multiple chapters that do not directly discuss DL can be a barrier to entry for many people.

In this book, we use yellow text boxes like this one to highlight things that we otherwise do not discuss or explore in detail but nonetheless think are important for you to learn at some point. We believe that an important part of learning about a new topic is to not only acquire some basic skills but also get some insights into what the next steps are. We use the yellow boxes to signal to you that at this point it is perfectly fine to ignore a certain topic, but it will be important to learn as a next step.

Let us now begin by stating that it is important to know about **traditional ML** if you want to get serious about DL, but you can wait to learn about traditional ML until you have gotten a taste of DL.

Not starting the book with traditional ML techniques is an attempt to avoid one of the buzz killers that we have found in other books. One very logical, and therefore typical, way of introducing DL is to first describe what ML is and, as such, to start with a very simple ML technique, namely, linear regression. It is easy, as an excited beginner, to be a little disheartened when you expect to learn about cool techniques to classify cat images and instead get stuck reading a discussion about fitting a straight line to a set of random data points using mathematics that seem completely unrelated to DL. We instead try to take the quickest, while still logical, path to getting to image classification to provide you with some instant satisfaction, but you will notice that we still sneak in some references and comparisons to linear regression over time.

Apart from deciding whether to include traditional ML as a topic, any author of a book on DL needs to take a position on whether to include code examples and how deeply to dive into the mathematics. Our view is that because DL is an applied field, a book on this topic needs to contain a good mix of theory and practice, so code examples are necessary. We also believe that many topics in DL are inherently mathematical, and it is necessary to include some of the mathematics to provide a good description of how things work. With that background, we try to describe certain concepts from different angles using a good mix of elements:

- Figures
- Natural language (English) descriptions
- Programming code snippets
- Mathematical formulas

Readers who master all of the preceding might find some descriptions redundant, but we believe that this is the best way of making the book accessible to a large audience.

This book does not aim to include details about all the most recent and advanced techniques in the DL field. Instead, we include concepts and techniques that we believe are fundamental to understanding the latest developments in the field. Some of the appendixes describe how some major architectures are built on these concepts, but most likely, even better architectures will emerge. Our goal is to give you enough knowledge to enable you to continue learning by reading more recent research papers. Therefore, we have also decided to sprinkle references throughout the book to enable you to follow up on topics that you find extra interesting. However, it has been our intention to make the book self-contained so that you should never need to look up a reference to be able to follow the explanations in the book. In some cases, we include references to things that we do not explain but mention only in passing. In those cases, we try to make it clear that it is meant as future reading instead of being a central element of the book.

The references in the book are strictly for future reading and should not be necessary to read to be able to understand the main topics of the book.

Is DL Dangerous?

There are plenty of science fiction books and movies that depict AI as a threat against humanity. Machines develop a form of consciousness and perceive humans as a threat and therefore decide to destroy us. There have also been thought experiments about how an AI accidentally destroys the human species as a side effect of trying to deliver on what it is programmed to do. One example is the paperclip maximizer (Bostrom, 2003), which is programmed with the goal of making as many paper clips as possible. In order to do so, it might kill all human beings to free up atoms needed to make paper clips. The risk that these exact scenarios will play out in practice is probably low, but researchers still see future powerful AIs as a significant risk.

More urgently, DL has already been shown to come with serious unintended consequences and malignant use. One example is a study of a commercially available facial recognition system (Buolamwini and Gebru, 2018) used by law enforcement. Although the system achieved 99% accuracy on lighter-skinned men, its accuracy on darker-skinned women was only 65%, thereby putting them at much greater risk of being incorrectly identified and possibly wrongly accused of crimes. An example of malignant use of DL is fake pornography (Dickson, 2019) whereby the technology is used to make it appear as if a person (often a celebrity) is featured in a pornographic video.

DL learns from data created by humans and consequently runs the risk of learning and even amplifying human biases. This underscores the need for taking a responsible approach to DL and AI. Historically, this topic has largely been neglected, but more recently started to receive more attention. A powerful demonstration can be found on the website of the Algorithmic Justice League (Buolamwini, n.d.) with a video showing how a face detection system fails to detect the face of a dark-skinned woman (Buolamwini) until she puts on a white mask.

Another example is the emergence of algorithmic auditing, where researchers identify and report human biases and other observed problems in commercial systems (Raji and Buolamwini, 2019). Researchers have proposed to document known biases and intended use cases of any released system to mitigate these problems. This applies both to the data used to create such systems (Gebru, et al., 2018) and to the released DL model itself (Mitchell et al., 2018). Thomas suggests a checklist of questions to guide DL practitioners throughout the course of a project to avoid ethical problems (Thomas, 2019). We touch on these topics throughout the book. We also provide resources for further reading in Chapter 18.

Choosing a DL Framework

As a practitioner of DL, you will need to decide what DL framework to use. A DL framework provides functionality that handles much of the low-level details when implementing DL models. Just as the DL field is rapidly evolving, so are the different frameworks. To mention a few, Caffe, Theano, MXNet, Torch, TensorFlow, and PyTorch have all been influential throughout the current DL boom. In addition to these full-fledged frameworks, there are specialized frameworks such as Keras and TensorRT. Keras is a high-level API that makes it easier to program for some of these frameworks. TensorRT is an inference optimizer and runtime engine that can be used to run models built and trained by many of the mentioned frameworks.

As of the writing of this book, our impression is that the two most popular full-fledged frameworks are TensorFlow and PyTorch, where TensorFlow nowadays includes native support for the Keras API. Another significant framework is MXNet. Models developed in either of these frameworks can be deployed using the TensorRT inference engine.

Deciding on what DL framework to use can be viewed as a life-changing decision. Some people would say that it is comparable to choosing a text editor or a spouse. We do not share that belief but think that the world is big enough for multiple competing solutions. We decided to provide programming examples in both TensorFlow and PyTorch for this book. The TensorFlow examples are printed in the book itself, but equivalent examples in PyTorch, including detailed descriptions, can be found on the book's website. We suggest that you pick a framework that you like or one that makes it easy to collaborate with people you interact with.

The programming examples in this book are provided in a TensorFlow version using the Keras API (printed in the book) as well as in a PyTorch version (online). Appendix I contains information about how to install TensorFlow and PyTorch, as well as a description of some of the key differences between the two frameworks.

Prerequisites for Learning DL

DL combines techniques from a number of different fields. If you want to get serious about DL, and particularly if you want to do research and publish your findings, over time you will need to acquire advanced knowledge within the scope of many of these skillsets. However, we believe that it is possible to get started with DL with little or partial knowledge in these areas. The sections that follow list the areas we find important, and in each section, we list the minimum set of knowledge that we think you need in order to follow this book.

STATISTICS AND PROBABILITY THEORY

Many DL problems do not have exact answers, so a central theme is probability theory. As an example, if we want to classify objects in an image, there is often uncertainty involved, such as how certain our model is that an object of a specific category, such as a cat, is present in the picture. Further, we might want to classify the type of cat—for example, is it a tiger, lion, jaguar, leopard, or snow leopard? The answer might be that the model is 90% sure that it is a jaguar, but there is a 5% probability that it is a leopard and so on. This book does not require deep knowledge in statistics and probability theory. We do expect you to be able to compute an arithmetic mean and understand the basic concept of probability. It is helpful, although not strictly required, if you know about variance and how to standardize a random variable.

LINEAR ALGEBRA

As you will learn in Chapter 1, the fundamental building block in DL is based on calculating a weighted sum of variables, which implies doing many additions and multiplications. Linear algebra is a field of mathematics that enables us to describe such calculations in a compact manner. This book frequently specifies formulas containing vectors and matrices. Further, calculations involve

- Dot products
- Matrix-vector multiplications
- Matrix-matrix multiplications

If you have not seen these concepts in the past, you will need to learn about them to follow the book. However, Chapter 1 contains a section that goes through these concepts. We suggest that you read that first and then assess whether you need to pick up a book about linear algebra.

CALCULUS

As you will learn in Chapters 2 and 3, the learning part in DL is based on minimizing the value of a function known as a *loss function* or *error function*. The technique used to minimize the loss function builds on the following concepts from calculus:

- Computing the derivative of a function of a single variable
- Computing partial derivatives of a function of multiple variables
- Calculating derivatives using the chain rule of calculus

However, just as we do for linear algebra, we provide sections that go through the basics of these concepts. These sections are found in Chapters 2 and 3.

NUMERICAL METHODS FOR CONSTRAINED AND UNCONSTRAINED OPTIMIZATION

In DL, it is typically not feasible to find an analytical solution when trying to minimize the loss function. Instead, we rely on numerical optimization methods. The most prevalent method is an iterative method known as *gradient descent*. It is helpful if you already know something about iterative methods and finding extreme points in continuous functions. However, we do not require prior knowledge of gradient descent, and we describe how it works before using it in Chapter 3.

PYTHON PROGRAMMING

It is hard to do anything except specific DL applications without some knowledge about programming in general. Further, given that the most popular DL frameworks are based on Python, it is highly recommended to acquire at least basic Python skills to enable trying out and modifying code examples. There are many good books on the topic of programming, and if you have basic programming skills, it should be relatively simple to get started with Python by just following tutorials at python.org. It is possible for nonprogrammers to read this book and just skip the coding sections, but if you intend to apply your DL skills in practice, you should learn the basics of Python programming.

You do not need to learn everything about Python to get started with DL. Many DL applications use only a small subset of the Python language, extended with heavy use of domain-specific DL frameworks and libraries. In particular, many introductory examples make little or no use of object-oriented programming constructs. A specific module that is used frequently is the *NumPy* (numerical Python) module that, among other things, provides data types for vectors and matrices. It is also common to use *pandas* (Python Data Manipulation Library) to manipulate multidimensional data, but we do not make use of pandas in this book.

The following Python constructs are frequent in most of the code examples in the book:

- Integer and floating point datatypes
- Lists and dictionaries
- Importing and using external packages
- NumPy arrays
- NumPy functions
- If-statements, for-loops, and while-loops
- Defining and calling functions
- Printing strings and numerical datatypes

- Plotting data with matplotlib
- Reading from and writing to files

In addition, many of the programming examples rely on constructs provided by a DL framework (TensorFlow in the book and PyTorch provided online). There is no need to know about these frameworks up front. The functionality is gradually introduced in the descriptions of the code examples. The code examples become progressively harder throughout the book, so if you are a beginner to coding, you will need to be prepared to spend some time honing your coding skills in parallel with reading the book.

DATA REPRESENTATION

Much of the DL mechanics are handled by highly optimized ML frameworks. However, your input data first needs to be converted into suitable formats that can be consumed by these frameworks. As such, you need to know something about the format of the data that you will use and, when applicable, how to convert it into a more suitable format. For example, for images, it is helpful to know the basics about RGB (red, green, blue) representation. Similarly, for the cases that use text as input data, it is helpful to know something about how characters are represented by a computer. In general, it is good to have some insight into how raw input data is often of low quality and needs to be cleaned. You will often find missing or duplicated data entries, timestamps from different time zones, and typos originating from manual processing. For the examples in this book, this is typically not a problem, but it is something you need to be aware of in a production setting.

About the Code Examples

You will find much overlap between the code examples in this book and code examples found in online tutorials as well as in other DL books (e.g., Chollet 2018; Glassner, 2018). Many of these examples have evolved from various published research papers in combination with publicly available datasets. (Datasets are described in more detail in Chapter 4.) In other words, we want to stress that we have not made up these examples from scratch, but they are heavily inspired by previously published work. However, we have done the actual implementation of these examples, and we have put our own touch on them to follow the organization of this book.

The longer code examples are broken up into smaller pieces and presented step by step interspersed throughout the text in the book. You should be able to just copy/paste or type each code snippet into a Python interpreter, but it is probably better to just put all code snippets for a specific code example in a single file and execute in a noninteractive manner. The code examples are also available for download both as regular Python files and as Jupyter notebooks at https://github.com/NVDLI/LDL/. See Appendix I for more details.

We were tempted to not provide downloadable versions of the code examples but instead force you to type them in yourself. After all, that is what we had to do in the 1980s when typing in a code listing from a computer magazine was a perfectly reasonable way of obtaining a new game. The youth of today with their app stores simply do not know how lucky they are.

In most chapters, we first present a basic version of a code example, and then we present results for variations of the program. We do not provide the full listings for all variations, but we try to provide all the necessary constructs in the book to enable you to do these variations yourself.

Modifying the code is left as an exercise for the reader. Hah, we finally got to say that!

Seriously, we do believe that modifying existing code is a good way of getting your hands dirty. However, there is no need to exactly recreate the variations we did. If you are new to programming, you can start with just tweaking existing parameter values instead of adding new code. If you already have more advanced coding skills, you can consider defining your own experiments based on what you find extra interesting.

DL algorithms are based on stochastic optimization techniques. As such, the results from an experiment may vary from time to time. That is, when you run a code example, you should not expect to get exactly the same result that is shown in the book. However, the overall behavior should be the same.

Another thing to note is that the chosen format, where we intersperse code throughout the book and explain each snippet, results in certain restrictions, such as minimizing the length of each program, and we have also tried to maintain a linear flow and to not heavily modularize the code into classes and functions in most cases. Thus, instead of using sound coding practices to make the code examples easy to extend and maintain, focus is on keeping the examples small and readable.

That is a lame excuse for writing ugly code, but whatever works. . .

Another thing to consider is what kind of development environment is needed to follow this book. In our opinion, anybody who wants to do serious work in DL will need to get access to a hardware platform that provides specific acceleration for DL—for example, a suitable graphics processing unit (GPU). However, if you do not have access to a GPU-based platform just yet, the code examples in the first few chapters are small enough to be run on a somewhat modern central processing unit (CPU) without too much pain. That is, you can start with a vanilla setup using the CPU for the first few chapters and then spend the resources needed to get access to a GPU-accelerated platform[1] when you are getting to Chapter 7.

Medium term, you should get access to a GPU accelerated platform, but you can live with a standard CPU for the beginning of the book.

Instructions on how to set up a machine with the necessary development environment can be found in Appendix I, which also contains links to the code examples and datasets used in this book.

How to Read This Book

This book is written in a linear fashion and is meant to be read from beginning to end. We introduce new concepts in each chapter and frequently build on and refer back to discussions in previous chapters. It is often the case that we try to avoid introducing too many new concepts at once. This sometimes results in logically similar concepts being introduced in different chapters. However, we do sometimes take a step back and try to summarize a group of related techniques once they have all been introduced. You will see this for hidden units in Chapter 5,

1. Nothing prevents you from running all programming examples on a CPU, but in some cases, you might need to do it overnight.

output units in Chapter 6, and techniques to address vanishing and exploding gradients in Chapter 10.

Readers who are complete beginners to neural networks and DL (the core target audience of the book) will likely find the first four chapters more challenging to get through than the remainder of the book. We introduce many new concepts. There is a fair amount of mathematical content, and we implement a neural network from scratch in Python. We encourage you to still try to get through these four chapters, but we also think it is perfectly fine to skim through some of the mathematical equations if you find them challenging. In Chapter 5, we move on to using a DL framework, and you will find that it will handle many of the details under the hood, and you can almost forget about them.

APPENDIXES

This book ends with a number of appendixes. Appendixes A through D could have been included as regular chapters in the book. However, we wanted to avoid information overload for first-time readers. Therefore, we decided to put some of the material in appendixes instead because we simply do not think that you need to learn those concepts in order to follow the narrative of the book. Our recommendation if you are a complete beginner to ML and DL is to read these appendixes last.

If you feel that you already know the basics about ML or DL, then it can make sense for you to read the first four appendixes interspersed among other chapters during your first pass through the book. Appendix A can be read after Chapter 3. Appendix B logically follows Chapter 8. Appendix C naturally falls after Chapter 13. Finally, Appendix D extends topics presented in Chapter 15.

Alternatively, even if you are a beginner but want to learn more details about a specific topic, then do go ahead and read the appendix that relates to that topic in the order just presented.

Appendixes E through H are shorter and focus on providing background or additional detail on some very specific topics. Appendix I describes how to set up a development environment and how to access the programming examples. Appendix J contains cheat sheets that summarize many of the concepts described throughout the book.[2]

2. Larger versions of these cheat sheets can be downloaded from http://informit.com/title/9780137470358.

GUIDANCE FOR READERS WHO DO NOT WANT TO READ ALL OF THIS BOOK

We recognize that some readers want to read this book in a more selective manner. This can be the case if you feel that you already have some of the basic skills or if you just want to learn about a specific topic. In this section, we provide some pointers for such readers, but this also means that we use some terminology that has not yet been introduced. If you are not interested in cherry picking chapters to read, then feel free to skip this section.

Figure P-5 illustrates three different envisioned tracks to follow depending on your interests. The leftmost track is what we just described, namely, to read the book from beginning to end.

If you are very interested in working with images and computer vision, we suggest that you read Appendix B about object detection, semantic segmentation, and instance segmentation. Further, the last few chapters of the book focus on natural language processing, and if that does not interest you, then we suggest that you skip Chapters 12 through 17. You should still skim Chapters 9 through 11 about recurrent neural networks. This track is shown in the middle of the figure.

If you want to focus mostly on language processing, then you can select the rightmost track. We suggest that you just skim Chapter 8 but do pay attention to the description of skip connections because it is referenced in later chapters. Then read Chapters 9 through 13, followed by Appendix C, then Chapters 14 and 15, and conclude with Appendix D. These appendixes contain additional content about word embeddings and describe GPT and BERT, which are important network architectures for language processing tasks.

Overview of Each Chapter and Appendix

This section contains a brief overview of each chapter. It can safely be skipped if you just want to cut to the chase and get started with LDL!

CHAPTER 1 – THE ROSENBLATT PERCEPTRON

The perceptron, a fundamental building block of a neural network, is introduced. You will learn limitations of the perceptron, and we show how to overcome

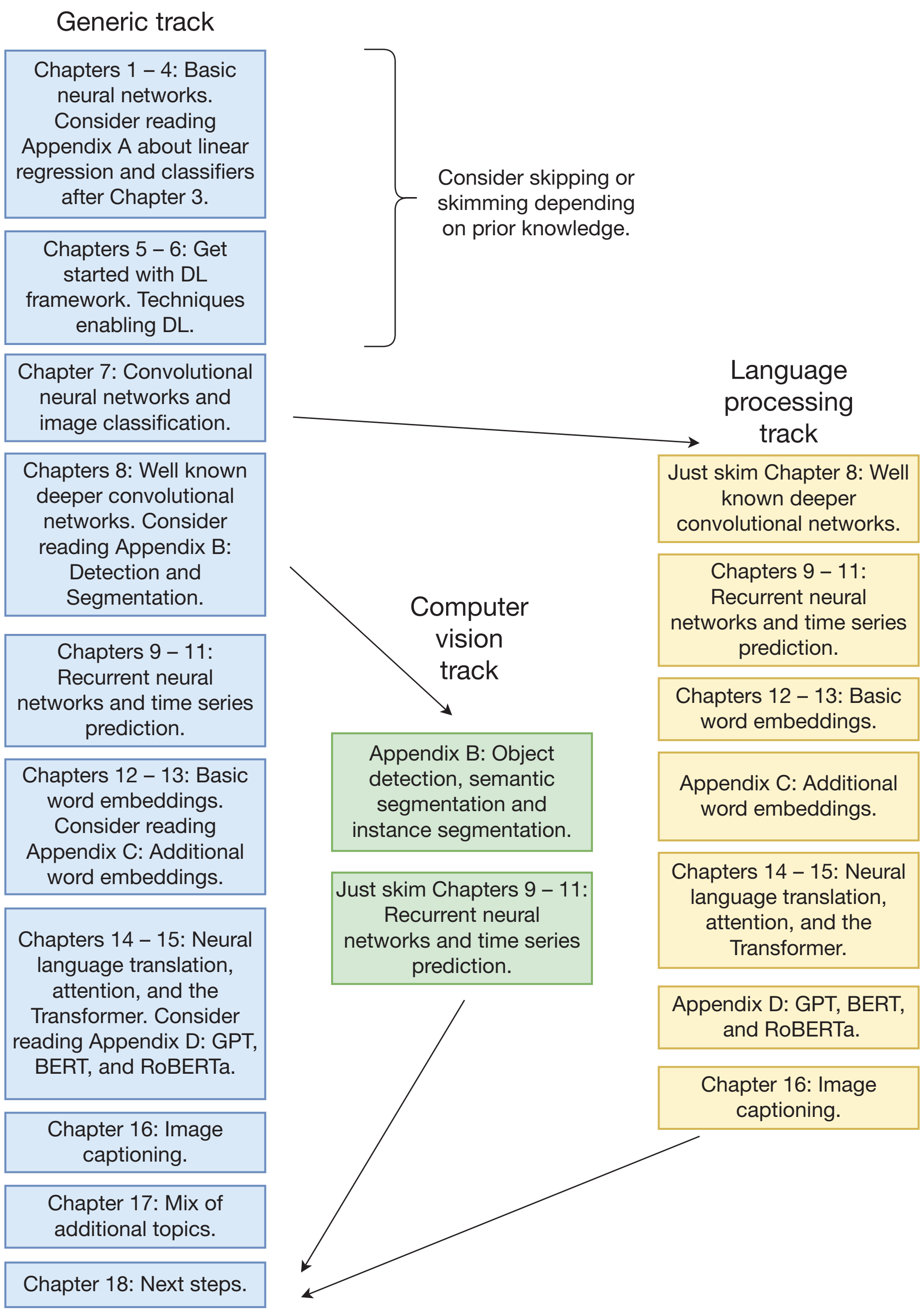
Generic track
Chapters 1 – 4: Basic neural networks. Consider reading Appendix A about linear regression and classifiers after Chapter 3.
Chapters 5 – 6: Get started with DL framework. Techniques enabling DL.
Consider skipping or skimming depending on prior knowledge.
Chapter 7: Convolutional neural networks and image classification.
Chapters 8: Well known deeper convolutional networks. Consider reading Appendix B: Detection and Segmentation.
Chapters 9 – 11: Recurrent neural networks and time series prediction.
Chapters 12 – 13: Basic word embeddings. Consider reading Appendix C: Additional word embeddings.
Chapters 14 – 15: Neural language translation, attention, and the Transformer. Consider reading Appendix D: GPT, BERT, and RoBERTa.
Chapter 16: Image captioning.
Chapter 17: Mix of additional topics.
Chapter 18: Next steps.
Computer vision track
Appendix B: Object detection, semantic segmentation and instance segmentation.
Just skim Chapters 9 – 11: Recurrent neural networks and time series prediction.
Language processing track
Just skim Chapter 8: Well known deeper convolutional networks.
Chapters 9 – 11: Recurrent neural networks and time series prediction.
Chapters 12 – 13: Basic word embeddings.
Appendix C: Additional word embeddings.
Chapters 14 – 15: Neural language translation, attention, and the Transformer.
Appendix D: GPT, BERT, and RoBERTa.
Chapter 16: Image captioning.

Figure P-5 Three different tracks to follow when reading this book these limitations by combining multiple perceptrons into a network. The chapter contains some programming examples of how to implement a perceptron and its learning algorithm.

CHAPTER 2 – GRADIENT-BASED LEARNING

We describe an optimization algorithm known as *gradient descent* and the theory behind the perceptron learning algorithm. This is used as a stepping-stone in the subsequent chapter that describes the learning algorithm for multilevel networks.

CHAPTER 3 – SIGMOID NEURONS AND BACKPROPAGATION

We introduce the backpropagation algorithm that is used for automatic learning in DNNs. This is both described in mathematical terms and implemented as a programming example used to do binary classification.

CHAPTER 4 – FULLY CONNECTED NETWORKS APPLIED TO MULTICLASS CLASSIFICATION

This chapter describes the concept of datasets and how they can be divided into a training set and a test set. It also touches on a network's ability to generalize. We extend the neural network architecture to handle multiclass classification, and the programming example then applies this to the task of classifying handwritten digits. This programming example is heavily inspired by an example created by Nielsen (2015).

CHAPTER 5 – TOWARD DL: FRAMEWORKS AND NETWORK TWEAKS

The example from the previous chapter is reimplemented using a DL framework. We show how this framework vastly simplifies the code and enables us to model many variations on our network. Chapter 5 also introduces many techniques that are needed to enable training of deeper networks.

CHAPTER 6 – FULLY CONNECTED NETWORKS APPLIED TO REGRESSION

In this chapter, we study how a network can be used to predict a numerical value instead of classification problems studied in previous chapters. We do this with a programming example in which we apply the network to a regression problem where we are trying to predict sales prices of houses based on a number of variables.

CHAPTER 7 – CONVOLUTIONAL NEURAL NETWORKS APPLIED TO IMAGE CLASSIFICATION

You will learn about the one type of network that initiated the DL boom in 2012, namely, the convolutional neural network, or just convolutional network. A CNN can be used in multiple problem domains, but it has been shown to be especially effective when applied to image classification/analysis. We explain how it works and walk through a programming example that uses a CNN to classify a more complex image dataset. In this example, instead of just distinguishing between different handwritten digits, we identify more complex object classes such as airplanes, automobiles, birds, and cats.

CHAPTER 8 – DEEPER CNNs AND PRETRAINED MODELS

Here we describe deeper CNNs such as GoogLeNet, VGG, and ResNet. As a programming example, we show how to download a pretrained ResNet implementation and how to use it to classify your own images.

CHAPTER 9 – PREDICTING TIME SEQUENCES WITH RECURRENT NEURAL NETWORKS

One limitation of the networks described in the previous chapters is that they are not well suited to handle data of different input lengths. Important problem domains such as text and speech often consist of sequences of varying lengths. This chapter introduces the recurrent neural network (RNN) architecture, which is well suited to handle such tasks. We use a programming example to explore how this network architecture can be used to predict the next data point in a time series.

CHAPTER 10 – LONG SHORT-TERM MEMORY

We discuss problems that prevent RNNs from learning long-term dependencies. We describe the long short-term memory (LSTM) technique that enables better handling of long sequences.

CHAPTER 11 – TEXT AUTOCOMPLETION WITH LSTM AND BEAM SEARCH

In this chapter, we explore how to use LSTM-based RNNs for longer-term prediction and introduce a concept known as *beam search*. We illustrate it with a programming example in which we build a network that can be used for autocompletion of text. This is a simple example of natural language generation (NLG), which is a subset of the greater field of natural language processing (NLP).

CHAPTER 12 – NEURAL LANGUAGE MODELS AND WORD EMBEDDINGS

The example in the previous chapter is based on individual characters instead of words. In many cases, it is more powerful to work with words and their semantics instead of working with individual characters. Chapter 12 introduces the concepts language models and word encodings in a vector space (also known as *embedding space*) that can be used to capture some important relationships between words. As code examples, we extend our autocompletion example to work with words instead of characters and explore how to create word vectors in an embedding space. We also discuss how to build a model that can do sentiment analysis on text. This is an example of natural language understanding (NLU), which is yet another subfield of NLP.

CHAPTER 13 – WORD EMBEDDINGS FROM word2vec AND GloVe

In this chapter, we discuss two popular techniques for creating word embeddings. We download a set of existing embeddings and show how they capture various semantic relationships between words.

CHAPTER 14 – SEQUENCE-TO-SEQUENCE NETWORKS AND NATURAL LANGUAGE TRANSLATION

At this point, we introduce a network known as a sequence-to-sequence network, which is a combination of two recurrent neural networks. A key property of such a network is that its output sequence can be of a different length than the input sequence. We combine this type of network with the word encodings studied in the previous chapter. We build a natural language translator that takes a word sequence in one language (e.g., French) as an input and outputs a word sequence in a different language (e.g., English). Further, the output might be a different number of words and in a different word order than the input word sequence. The sequence-to-sequence model is an example of an architecture known as *encoder-decoder architecture*.

CHAPTER 15 – ATTENTION AND THE TRANSFORMER

In this chapter, we describe a technique known as *attention*, which can improve the accuracy of encoder-decoder architectures. We describe how it can be used to improve the neural machine translator from the previous chapter. We also describe the attention-based Transformer architecture. It is a key building block in many NLP applications.

CHAPTER 16 – ONE-TO-MANY NETWORK FOR IMAGE CAPTIONING

We describe in this chapter how a one-to-many network can be used to create textual descriptions of images and how to extend such a network with attention. A programming example implements this image-captioning network and demonstrates how it can be used to generate textual descriptions of a set of pictures.

CHAPTER 17 – MEDLEY OF ADDITIONAL TOPICS

Up until this point, we have organized topics so that they build on each other. In this chapter, we introduce a handful of topics that we did not find a good way of including in the previous chapters. Examples of such topics are autoencoders, multimodal learning, multitask learning, and neural architecture search.

CHAPTER 18 – SUMMARY AND NEXT STEPS

In the final chapter, we organize and summarize the topics discussed in earlier chapters to give you a chance to confirm that you have captured the key concepts described in the book. In addition to the summary, we provide some guidance to future reading tailored according to the direction you want to take—for example, highly theoretical versus more practical. We also discuss the topics of ethical AI and data ethics.

APPENDIX A – LINEAR REGRESSION AND LINEAR CLASSIFIERS

The focus of this book is DL. Our approach to the topic is to jump straight into DL without first describing traditional ML techniques. However, this appendix does describe very basic ML topics so you can get an idea of how some of the presented DL concepts relate to more traditional ML techniques. This appendix logically follows Chapter 3.

APPENDIX B – OBJECT DETECTION AND SEGMENTATION

In this appendix, we describe techniques to detect and classify multiple objects in a single image. It includes both coarse-grained techniques that draw bounding boxes around the objects and fine-grained techniques that pinpoint the individual pixels in an image that correspond to a certain object. This appendix logically follows Chapter 8.

APPENDIX C – WORD EMBEDDINGS BEYOND word2vec AND GloVe

In this appendix, we describe some more elaborate techniques for word embeddings. In particular, these techniques can handle words that did not exist in the training dataset. Further, we describe a technique that can handle cases in which a word has a different meaning depending on its context. This appendix logically follows Chapter 13.

APPENDIX D – GPT, BERT, AND RoBERTa

This appendix describes architectures that build on the Transformer. These network architectures have resulted in significant improvements in many NLP tasks. This appendix logically follows Chapter 15.

APPENDIX E – NEWTON-RAPHSON VERSUS GRADIENT DESCENT

In Chapter 2, we introduce a mathematical concept technique known as *gradient descent*. This appendix describes a different method, known as *Newton-Raphson,* and how it relates to gradient descent.

APPENDIX F – MATRIX IMPLEMENTATION OF DIGIT CLASSIFICATION NETWORK

In Chapter 4, we include a programming example implementing a neural network in Python code. This appendix describes two different optimized variations of that programming example.

APPENDIX G – RELATING CONVOLUTIONAL LAYERS TO MATHEMATICAL CONVOLUTION

In Chapter 7, we describe convolutional neural networks. They are based on, and named after, a mathematical operation known as *convolution.* This appendix describes this connection in more detail.

APPENDIX H – GATED RECURRENT UNITS

In Chapter 10, we describe a network unit known as *long short-term memory* (LSTM). In this appendix, we describe a simplified version of this unit known as *gated recurrent unit* (GRU).

APPENDIX I – SETTING UP A DEVELOPMENT ENVIRONMENT

This appendix contains information about how to set up a development environment. This includes how to install a deep learning framework and where to find the code examples. It also contains a brief section about key differences between TensorFlow and PyTorch, which are the two DL frameworks used for the code examples in this book.

APPENDIX J – CHEAT SHEETS

This appendix contains a set of cheat sheets that summarize much of the content in this book. They are also available for download in a different form factor: http://informit.com/title/9780137470358.

Register your copy of *Learning Deep Learning* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780137470358) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

Acknowledgments

I am incredibly grateful for all help I have received in the process of writing this book. I would like to extend my warmest gratitude to all of you:

- Eric Haines for reading this book front to back and providing guidance and feedback throughout the whole process. Having had you as a sounding board and discussion partner was invaluable.
- Ankit Patel and Amanda Lam for believing in me. Without any prior interaction with me, you worked extra hours to figure out how to make this book happen. Thank you for finding the perfect publisher, and thanks to Jenny Chen for working out the agreement. Being represented by a professional team allowed me to focus solely on the content of the book.
- Nick Cohron, Orazio Gallo, Boris Ginsburg, Samuli Laine, Ryan Prenger, Raul Puri, Kevin Shih, and Sophie Tabac for providing expert feedback on the material. All of your comments greatly improved the book.
- Aaron Beddes and Torbjörn Ekman for reading an early manuscript and providing valuable feedback, which gave me confidence to engage all the people above.
- Anders Landin, Feihui Li, Niklas Lindström, Jatin Mitra, Clint Olsen, Sebastian Sylvan, and Johan Överby for pointing out various issues, both in the manuscript and in the code examples.
- Andy Cook for your vision of how to tie the book to efforts within the NVIDIA Deep Learning Institute, as well as your involvement in the cover art proposal, together with Sandra Froehlich and Chris Strach. Sandra and Chris also contributed with other aspects of style and branding. The original image of the R.Eva Robot was designed by Gavriil Klimov and Gregor Kopka.
- Anima Anandkumar and Craig Clawson for writing the forewords.

- All people who Pearson involved in the publishing process, in particular Debra Williams Cauley, Carol Lallier, Julie Nahil, Chuti Prasertsith, and Chris Zahn.

- Darrell Boggs for providing your support when I first introduced the idea of starting this project. Further, the following NVIDIA colleagues all played a role in making this happen by supporting the project or connecting me with the right people: Tomas Akenine-Möller, Anima Anandkumar, Jonathan Cohen, Greg Estes, Sanja Fidler, David Hass, Brian Kelleher, Will Ramey, and Mohammad Shoeybi.

- The research community and other authors. This book does not contain original ideas. Its focus is on describing published work from multiple sources in a common framework. This would not have been possible without the original publications as well as multiple books on the topic. I have done my best to list all these sources in the bibliography.

Finally, I am grateful to my wife, Jennifer, and my children, Sebastian and Sofia, for being understanding and enabling me to spend the time required to write this book. I also want to give credit to our dog, Babette, and late cat, Stella, because I used their pictures in various object classification examples.

About the Author

Magnus Ekman, PhD, is a Director of Architecture at NVIDIA Corporation. His doctorate is in computer engineering, and he holds multiple patents. He was first exposed to artificial neural networks in the late 1990s in his native country, Sweden. After some dabbling in evolutionary computation, he focused on computer architecture and relocated to Silicon Valley, where he lives with his wife, Jennifer, children, Sebastian and Sofia, and dog, Babette. He has previously worked with processor design and R&D at Sun Microsystems and Samsung Research America and has been involved in starting two companies, one of which (Skout) was later acquired by The Meet Group, Inc. In his current role at NVIDIA, he leads an engineering team working on CPU performance and power efficiency for chips targeting markets ranging from autonomous vehicles to data centers for artificial intelligence (AI).

As the deep learning (DL) field exploded in the past few years, fueled by NVIDIA's GPU technology and CUDA, Dr. Ekman found himself in the midst of a company expanding beyond computer graphics and becoming a DL powerhouse. As a part of that journey, he challenged himself to stay up to date with the most recent developments in the field. He considers himself an educator, and in the process of writing *Learning Deep Learning* (LDL) he partnered with the NVIDIA Deep Learning Institute (DLI), which offers hands-on training in AI, accelerated computing, and accelerated data science. He is thrilled about DLI's plans to add LDL to its existing portfolio of self-paced online courses; live, instructor-led workshops; educator programs; and teaching kits.

This page intentionally left blank

Chapter 5

Toward DL: Frameworks and Network Tweaks

An obvious next step would be to see if adding more layers to our neural networks results in even better accuracy. However, it turns out getting deeper networks to learn well is a major obstacle. A number of innovations were needed to overcome these obstacles and enable deep learning (DL). We introduce the most important ones later in this chapter, but before doing so, we explain how to use a DL framework. The benefit of using a DL framework is that we do not need to implement all these new techniques from scratch in our neural network. The downside is that you will not deal with the details in as much depth as in previous chapters. You now have a solid enough foundation to build on. Now we switch gears a little and focus on the big picture of solving real-world problems using a DL framework. The emergence of DL frameworks played a significant role in making DL practical to adopt in the industry as well as in boosting productivity of academic research.

Programming Example: Moving to a DL Framework

In this programming example, we show how to implement the handwritten digit classification from Chapter 4, “Fully Connected Networks Applied to Multiclass Classification,” using a DL framework. In this book, we have chosen to use the two frameworks TensorFlow and PyTorch. Both of these frameworks are popular and flexible. The TensorFlow versions of the code examples are interspersed throughout the book, and the PyTorch versions are available online on the book Web site.

TensorFlow provides a number of different constructs and enables you to work at different abstraction levels using different application programming interfaces (APIs). In general, to keep things simple, you want to do your work at the highest abstraction level possible because that means that you do not need to implement the low-level details. For the examples we will study, the Keras API is a suitable abstraction level. Keras started as a stand-alone library. It was not tied to TensorFlow and could be used with multiple DL frameworks. However, at this point, Keras is fully supported inside of TensorFlow itself. See Appendix I for information about how to install TensorFlow and what version to use.

Appendix I also contains information about how to install PyTorch if that is your framework of choice. Almost all programming constructs in this book exist both in TensorFlow and in PyTorch. The section “Key Differences between PyTorch and TensorFlow” in Appendix I describes some key differences between the two frameworks. You will find it helpful if you do not want to pick a single framework but want to master both of them.

The frameworks are implemented as Python libraries. That is, we still write our program as a Python program and we just import the framework of choice as a library. We can then use DL functions from the famework in our program. The initialization code for our TensorFlow example is shown in Code Snippet 5-1.

Code Snippet 5-1 Import Statements for Our TensorFlow/Keras Example

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
import numpy as np
import logging
```

```
tf.get_logger().setLevel(logging.ERROR)
tf.random.set_seed(7)

EPOCHS = 20
BATCH_SIZE = 1
```

As you can see in the code, TensorFlow has its own random seed that needs to be set if we want reproducible results. However, this still does not guarantee that repeated runs produce identical results for all types of networks, so for the remainder of this book, we will not worry about setting the random seeds. The preceding code snippet also sets the logging level to only print out errors while suppressing warnings.

We then load and prepare our MNIST dataset. Because MNIST is a common dataset, it is included in Keras. We can access it by a call to `keras.datasets.mnist` and `load_data`. The variables `train_images` and `test_images` will contain the input values, and the variables `train_labels` and `test_labels` will contain the ground truth (Code Snippet 5-2).

Code Snippet 5-2 Load and Prepare the Training and Test Datasets

```
# Load training and test datasets.
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images,
                               test_labels) = mnist.load_data()

# Standardize the data.
mean = np.mean(train_images)
stddev = np.std(train_images)
train_images = (train_images - mean) / stddev
test_images = (test_images - mean) / stddev

# One-hot encode labels.
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)
```

Just as before, we need to standardize the input data and one-hot encode the labels. We use the function `to_categorical` to one-hot encode our labels instead of doing it manually, as we did in our previous example. This serves as an example of how the framework provides functionality to simplify our implementation of common tasks.

If you are not so familiar with Python, it is worth pointing out that functions can be defined with optional arguments, and to avoid having to pass the arguments in a specific order, optional arguments can be passed by first naming which argument we are trying to set. An example is the **num_classes** argument in the **to_categorical** function.

We are now ready to create our network. There is no need to define variables for individual neurons because the framework provides functionality to instantiate entire layers of neurons at once. We do need to decide how to initialize the weights, which we do by creating an initializer object, as shown in Code Snippet 5-3. This might seem somewhat convoluted but will come in handy when we want to experiment with different initialization values.

Code Snippet 5-3 Create the Network

```
# Object used to initialize weights.
initializer = keras.initializers.RandomUniform(
    minval=-0.1, maxval=0.1)

# Create a Sequential model.
# 784 inputs.
# Two Dense (fully connected) layers with 25 and 10 neurons.
# tanh as activation function for hidden layer.
# Logistic (sigmoid) as activation function for output layer.
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(25, activation='tanh',
                       kernel_initializer=initializer,
                       bias_initializer='zeros'),
    keras.layers.Dense(10, activation='sigmoid',
                       kernel_initializer=initializer,
                       bias_initializer='zeros')])
```

The network is created by instantiating a `keras.Sequential` object, which implies that we are using the Keras Sequential API. (This is the simplest API, and we use it for the next few chapters until we start creating networks that require a more advanced API.) We pass a list of layers as an argument to the `Sequential` class. The first layer is a `Flatten` layer, which does not do computations but only changes the organization of the input. In our case, the inputs are changed from a 28×28 array into an array of 784 elements. If the data had already been organized into a 1D-array, we could have skipped the `Flatten` layer and simply declared the two `Dense` layers. If we had done it that way, then we would have needed to pass an `input_shape` parameter to the first `Dense` layer because we always have to declare the size of the inputs to the first layer in the network.

The second and third layers are both `Dense` layers, which means they are fully connected. The first argument tells how many neurons each layer should have, and the `activation` argument tells the type of activation function; we choose `tanh` and `sigmoid`, where `sigmoid` means the *logistic sigmoid function*. We pass our `initializer` object to initialize the regular weights using the `kernel_initializer` argument. The bias weights are initialized to 0 using the `bias_initializer` argument.

One thing that might seem odd is that we are not saying anything about the number of inputs and outputs for the second and third layers. If you think about it, the number of inputs is fully defined by saying that both layers are fully connected and the fact that we have specified the number of neurons in each layer along with the number of inputs to the first layer of the network. This discussion highlights that using the DL framework enables us to work at a higher abstraction level. In particular, we use layers instead of individual neurons as building blocks, and we need not worry about the details of how individual neurons are connected to each other. This is often reflected in our figures as well, where we work with individual neurons only when we need to explain alternative network topologies. On that note, Figure 5-1 illustrates our digit recognition network at this higher abstraction level. We use rectangular boxes with rounded corners to depict a layer of neurons, as opposed to circles that represent individual neurons.

We are now ready to train the network, which is done by Code Snippet 5-4. We first create a `keras.optimizer.`SGD object. This means that we want to use stochastic gradient descent (SGD) when training the network. Just as with the initializer, this might seem somewhat convoluted, but it provides flexibility to adjust parameters for the learning process, which we explore soon. For now, we just set the learning rate to 0.01 to match what we did in our plain Python example. We then prepare the model for training by calling the model's `compile`

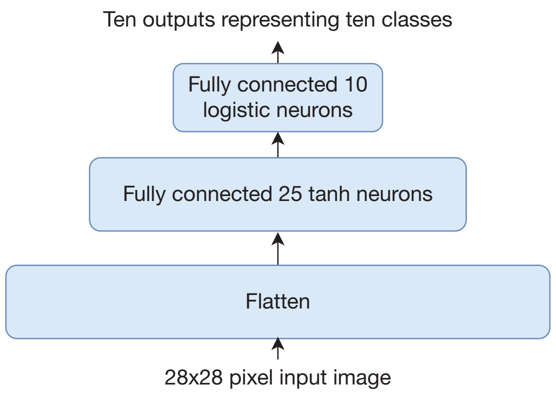
Ten outputs representing ten classes
Fully connected 10 logistic neurons
Fully connected 25 tanh neurons
Flatten
28x28 pixel input image

Figure 5-1 Digit classification network using layers as building blocks

function. We provide parameters to specify which `loss` function to use (where we use `mean_squared_error` as before), the optimizer that we just created and that we are interested in looking at the `accuracy` metric during training.

Code Snippet 5-4 Train the Network

```
# Use stochastic gradient descent (SGD) with
# learning rate of 0.01 and no other bells and whistles.
# MSE as loss function and report accuracy during training.
opt = keras.optimizers.SGD(learning_rate=0.01)

model.compile(loss='mean_squared_error', optimizer = opt,
              metrics =['accuracy'])

# Train the model for 20 epochs.
# Shuffle (randomize) order.
# Update weights after each example (batch_size=1).
history = model.fit(train_images, train_labels,
                    validation_data=(test_images, test_labels),
                    epochs=EPOCHS, batch_size=BATCH_SIZE,
                    verbose=2, shuffle=True)
```

We finally call the `fit` function for the model, which starts the training process. As the function name indicates, it fits the model to the data. The first two arguments specify the training dataset. The parameter `validation_data` is the test dataset. Our variables EPOCHS and BATCH_SIZE from the initialization code determine how many epochs to train for and what batch size we use. We had set BATCH_SIZE to 1, which means that we update the weight after a single training example, as we did in our plain Python example. We set `verbose=2` to get a reasonable amount of information printed during the training process and set `shuffle` to `True` to indicate that we want the order of the training data to be randomized during the training process. All in all, these parameters match what we did in our plain Python example.

Depending on what TensorFlow version you run, you might get a fair number of printouts about opening libraries, detecting the graphics processing unit (GPU), and other issues as the program starts. If you want it less verbose, you can set the environment variable TF_CPP_MIN_LOG_LEVEL to 2. If you are using bash, you can do that with the following command line:

```
export TF_CPP_MIN_LOG_LEVEL=2
```

Another option is to add the following code snippet at the top of your program.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

The printouts for the first few training epochs are shown here. We stripped out some timestamps to make it more readable.

```
Epoch 1/20

loss: 0.0535 - acc: 0.6624 - val_loss: 0.0276 - val_acc: 0.8893

Epoch 2/20

loss: 0.0216 - acc: 0.8997 - val_loss: 0.0172 - val_acc: 0.9132

Epoch 3/20

loss: 0.0162 - acc: 0.9155 - val_loss: 0.0145 - val_acc: 0.9249

Epoch 4/20

loss: 0.0142 - acc: 0.9227 - val_loss: 0.0131 - val_acc: 0.9307
```

```
Epoch 5/20

loss: 0.0131 - acc: 0.9274 - val_loss: 0.0125 - val_acc: 0.9309

Epoch 6/20

loss: 0.0123 - acc: 0.9313 - val_loss: 0.0121 - val_acc: 0.9329
```

In the printouts, `loss` represents the mean squared error (MSE) of the training data, `acc` represents the prediction accuracy on the training data, `val_loss` represents the MSE of the test data, and `val_acc` represents the prediction accuracy of the test data. It is worth noting that we do not get exactly the same learning behavior as was observed in our plain Python model. It is hard to know why without diving into the details of how TensorFlow is implemented. Most likely, it could be subtle issues related to how initial parameters are randomized and the random order in which training examples are picked. Another thing worth noting is how simple it was to implement our digit classification application using TensorFlow. Using the TensorFlow framework enables us to study more advanced techniques while still keeping the code size at a manageable level.

We now move on to describing some techniques needed to enable learning in deeper networks. After that, we can finally do our first DL experiment in the next chapter.

The Problem of Saturated Neurons and Vanishing Gradients

In our experiments, we made some seemingly arbitrary changes to the learning rate parameter as well as to the range with which we initialized the weights. For our perceptron learning example and the XOR network, we used a learning rate of 0.1, and for the digit classification, we used 0.01. Similarly, for the weights, we used the range −1.0 to +1.0 for the XOR example, whereas we used −0.1 to +0.1 for the digit example. A reasonable question is whether there is some method to the madness. Our dirty little secret is that we changed the values simply because our networks did not learn well without these changes. In this section, we discuss the reasons for this and explore some guidelines that can be used when selecting these seemingly random parameters.

To understand why it is sometimes challenging to get networks to learn, we need to look in more detail at our activation function. Figure 5-2 shows our two S-shaped functions. It is the same chart that we showed in Figure 3-4 in Chapter 3, "Sigmoid Neurons and Backpropagation."

One thing to note is that both functions are uninteresting outside of the shown *z*-interval (which is why we showed only this *z*-interval in the first place). Both functions are more or less straight horizontal lines outside of this range.

Now consider how our learning process works. We compute the derivative of the error function and use that to determine which weights to adjust and in what direction. Intuitively, what we do is tweak the input to the activation function (*z* in the chart in Fig. 5-2) slightly and see if it affects the output. If the *z*-value is within the small range shown in the chart, then this will change the output (the *y*-value in the chart). Now consider the case when the *z*-value is a large positive or negative number. Changing the input by a small amount (or even a large amount) will not affect the output because the output is a horizontal line in those regions. We say that the neuron is *saturated*.

Saturated neurons can cause learning to stop completely. As you remember, when we compute the gradient with the backpropagation algorithm, we propagate the error backward through the network, and part of that process is to multiply the derivative of the loss function by the derivative of the activation function. Consider

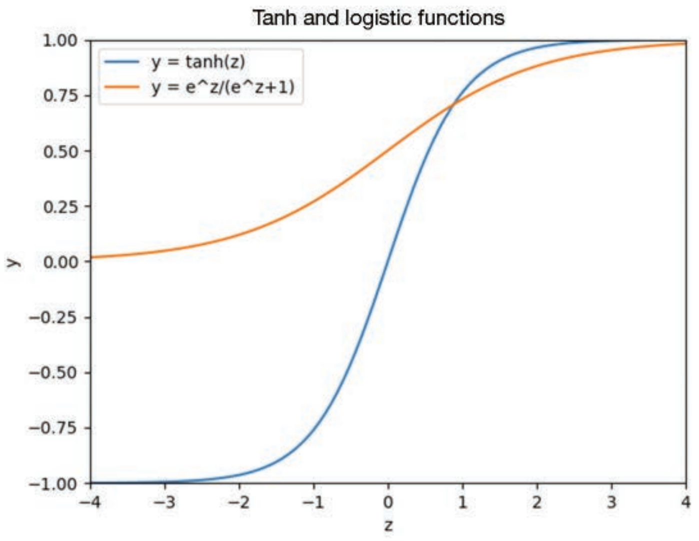

Figure 5-2 The two S-shaped functions tanh and logistic sigmoid

what the derivatives of the two activation functions above are for z-values of significant magnitude (positive or negative). The derivative is 0! In other words, no error will propagate backward, and no adjustments will be done to the weights. Similarly, even if the neuron is not fully saturated, the derivative is less than 0. Doing a series of multiplications (one per layer) where each number is less than 0 results in the gradient approaching 0. This problem is known as the *vanishing gradient problem*. Saturated neurons are not the only reason for vanishing gradients, as we will see later in the book.

Saturated neurons are insensitive to input changes because their derivative is 0 in the saturated region. This is one cause of the **vanishing gradient** problem where the backpropagated error is 0 and the weights are not adjusted.

Initialization and Normalization Techniques to Avoid Saturated Neurons

We now explore how we can prevent or address the problem of saturated neurons. Three techniques that are commonly used—and often combined—are weight initialization, input standardization, and batch normalization.

WEIGHT INITIALIZATION

The first step in avoiding saturated neurons is to ensure that our neurons are not saturated to begin with, and this is where weight initialization is important. It is worth noting that, although we use the same type of neurons in our different examples, the actual parameters for the neurons that we have shown are much different. In the XOR example, the neurons in the hidden layer had three inputs including the bias, whereas for the digit classification example, the neurons in the hidden layer had 785 inputs. With that many inputs, it is not hard to imagine that the weighted sum can swing far in either the negative or positive direction if there is just a little imbalance in the number of negative versus positive inputs if the weights are large. From that perspective, it kind of makes sense that if a neuron has a large number of inputs, then we want to initialize the weights to a smaller value to have a reasonable probability of still keeping the input to the activation function close to 0 to avoid saturation. Two popular weight initialization strategies are Glorot initialization (Glorot and Bengio, 2010) and He initialization (He et al., 2015b). Glorot initialization is recommended for tanh- and sigmoid-based neurons, and He initialization is recommended for ReLU-based neurons (described later). Both of these take the number of inputs into account, and Glorot initialization also takes the number of outputs into account. Both Glorot and He initialization exist in two flavors, one that is based on a uniform random distribution and one that is based on a normal random distribution.

We do not go into the formulas for **Glorot** and **He initialization**, but they are good topics well worth considering for further reading (Glorot and Bengio, 2010; He et al., 2015b).

We have previously seen how we can initialize the weights from a uniform random distribution in TensorFlow by using an initializer, as was done in Code Snippet 5-4. We can choose a different initializer by declaring any one of the supported initializers in Keras. In particular, we can declare a Glorot and a He initializer in the following way:

```
initializer = keras.initializers.glorot_uniform()
initializer = keras.initializers.he_normal()
```

Parameters to control these initializers can be passed to the initializer constructor. In addition, both the Glorot and He initializers come in the two flavors `uniform` and `normal`. We picked uniform for Glorot and normal for He because that is what was described in the publications where they were introduced.

If you do not feel the need to tweak any of the parameters, then there is no need to declare an initializer object at all, but you can just pass the name of the initializer as a string to the function where you create the layer. This is shown in Code Snippet 5-5, where the `kernel_initializer` argument is set to *`'glorot_uniform'`*.

Code Snippet 5-5 Setting an Initializer by Passing Its Name as a String

```
model = keras.Sequential([
        keras.layers.Flatten(input_shape=(28, 28)),
        keras.layers.Dense(25, activation='tanh',
                           kernel_initializer='glorot_uniform',
                           bias_initializer='zeros'),
        keras.layers.Dense(10, activation='sigmoid',
                           kernel_initializer='glorot_uniform',
                           bias_initializer='zeros')])
```

We can separately set `bias_initializer` to any suitable initializer, but as previously stated, a good starting recommendation is to just initialize the bias weights to 0, which is what the `'zeros'` initializer does.

INPUT STANDARDIZATION

In addition to initializing the weights properly, it is important to preprocess the input data. In particular, standardizing the input data to be centered around 0 and with most values close to 0 will reduce the risk of saturating neurons from the start. We have already used this in our implementation; let us discuss it in a little bit more detail. As stated earlier, each pixel in the MNIST dataset is represented by an integer between 0 and 255, where 0 represents the blank paper and a higher value represents pixels where the digit was written.[1] Most of the pixels will be either 0 or a value close to 255, where only the edges of the digits are somewhere in between. Further, a majority of the pixels will be 0 because a digit is sparse and does not cover the entire 28×28 image. If we compute the average pixel value for the entire dataset, then it turns out that it is about 33. Clearly, if we used the raw pixel values as inputs to our neurons, then there would be a big risk that the neurons would be far into the saturation region. By subtracting the mean and dividing by the standard deviation, we ensure that the neurons get presented with input data that is in the region that does not lead to saturation.

BATCH NORMALIZATION

Normalizing the inputs does not necessarily prevent saturation of neurons for hidden layers, and to address that problem Ioffe and Szegedy (2015) introduced batch normalization. The idea is to normalize values inside of the network as well and thereby prevent hidden neurons from becoming saturated. This may sound somewhat counterintuitive. If we normalize the output of a neuron, does that not result in undoing the work of that neuron? That would be the case if it truly was just normalizing the values, but the batch normalization function also contains parameters to counteract this effect. These parameters are adjusted during the learning process. Noteworthy is that after the initial idea was published, subsequent work indicated that the reason batch normalization works is different than the initial explanation (Santurkar et al., 2018).

Batch normalization (Ioffe and Szegedy, 2015) is a good topic for further reading.

1. This might seem odd because a value of 0 typically represents black and a value of 255 typically represents white for a grayscale image. However, that is not the case for this dataset.

There are two main ways to apply batch normalization. In the original paper, the suggestion was to apply the normalization on the input to the activation function (after the weighted sum). This is shown to the left in Figure 5-3.

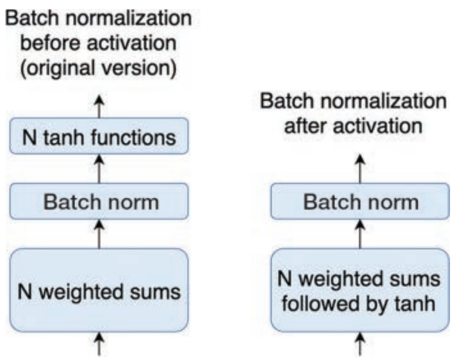

Figure 5-3 Left: Batch normalization as presented by Ioffe and Szegedy (2015). The layer of neurons is broken up into two parts. The first part is the weighted sums for all neurons. Batch normalization is applied to these weighted sums. The activation function (tanh) is applied to the output of the batch normalization operation. Right: Batch normalization is applied to the output of the activation functions.

This can be implemented in Keras by instantiating a layer without an activation function, followed by a `BatchNormalization` layer, and then apply an activation function without any new neurons, using the `Activation` layer. This is shown in Code Snippet 5-6.

Code Snippet 5-6 Batch Normalization before Activation Function

```
keras.layers.Dense(64),
keras.layers.BatchNormalization(),
keras.layers.Activation('tanh'),
```

However, it turns out that batch normalization also works well if done after the activation function, as shown to the right in Figure 5-3. This alternative implementation is shown in Code Snippet 5-7.

Code Snippet 5-7 Batch Normalization after Activation Function

```
keras.layers.Dense(64, activation='tanh'),
keras.layers.BatchNormalization(),
```

Cross-Entropy Loss Function to Mitigate Effect of Saturated Output Neurons

One reason for saturation is that we are trying to make the output neuron get to a value of 0 or 1, which itself drives it to saturation. A simple trick introduced by LeCun, Bottou, Orr, and Müller (1998) is to instead set the desired output to 0.1 or 0.9, which restricts the neuron from being pushed far into the saturation region. We mention this technique for historical reasons, but a more mathematically sound technique is recommended today.

We start by looking at the first couple of factors in the backpropagation algorithm; see Chapter 3, Equation 3-1(1) for more context. The formulas for the MSE loss function, the logistic sigmoid function, and their derivatives for a single training example are restated here:2

$$MSE\ loss: e(\hat{y}) = \frac{(y-\hat{y})^2}{2}, \qquad e'(\hat{y}) = -(y-\hat{y})$$

$$Logistic: S(z_f) = \frac{1}{1-e^{-z_f}}, \qquad S'(z_f) = S(z_f)\cdot(1-S(z_f))$$

We then start backpropagation by using the chain rule to compute the derivative of the loss function and multiply by the derivative of the logistic sigmoid function to arrive at the following as the error term for the output neuron:

$$Output\ neuron\ error\ term: \frac{\partial e}{\partial z_f} = \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_f} = -(y-\hat{y})\cdot S'(z_f)$$

We chose to not expand $S'(z_f)$ in the expression because it makes the formula unnecessarily cluttered. The formula reiterates what we stated in one of the previous sections: that if $S'(z_f)$ is close to 0, then no error will backpropagate through the network. We show this visually in Figure 5-4. We simply plot the derivative of the loss function and the derivative of the logistic sigmoid function as well as the product of the two. The chart shows these entities as functions of the output value *y* (horizontal axis) of the output neuron. The chart assumes that the desired output value (ground truth) is 0. That is, at the very left in the chart, the output value matches the ground truth, and no weight adjustment is needed.

2. In the equations in Chapter 3, we referred to the output of the last neuron as *f* to avoid confusing it with the output of the other neuron, *g*. In this chapter, we use a more standard notation and refer to predicted value (the output of the network) as $\hat{y}$.

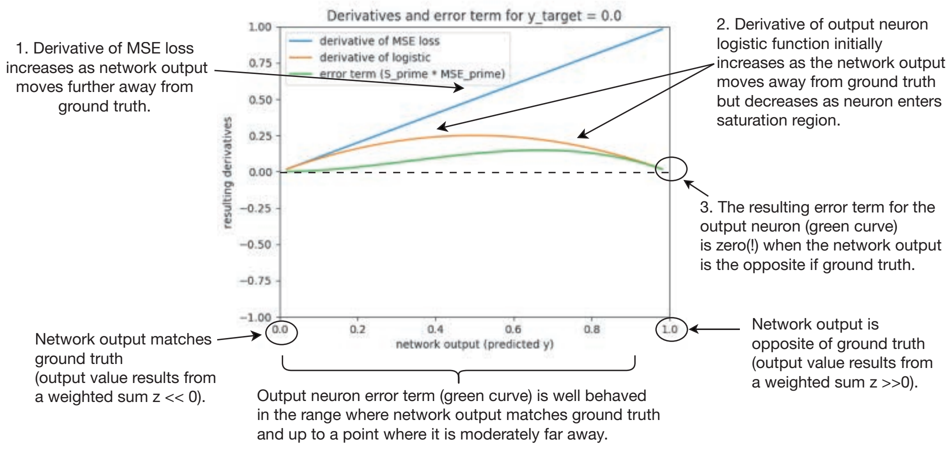

Figure 5-4 Derivatives and error term as function of neuron output when ground truth y (denoted y_target in the figure) is 0

As we move to the right in the chart, the output is further away from the ground truth, and the weights need to be adjusted. Looking at the figure, we see that the derivative of the loss function (blue) is 0 if the output value is 0, and as the output value increases, the derivative increases. This makes sense in that the further away from the true value the output is, the larger the derivative will be, which will cause a larger error to backpropagate through the network. Now look at the derivative of the logistic sigmoid function. It also starts at 0 and increases as the output starts deviating from 0. However, as the output gets closer to 1, the derivative is decreasing again and starts approaching 0 as the neuron enters its saturation region. The green curve shows the resulting product of the two derivatives (the error term for the output neuron), and it also approaches 0 as the output approaches 1 (i.e., the error term becomes 0 when the neuron saturates).

Looking at the charts, we see that the problem arises from the combination of the derivative of the activation function approaching 0, whereas the derivative of the loss function never increases beyond 1, and multiplying the two will therefore approach 0. One potential solution to this problem is to use a different loss function whose derivative can take on much higher values than 1. Without further rationale at this point, we introduce the function in Equation 5-1 that is known as the *cross-entropy loss function:*

$$\textit{Cross entropy loss}: \; e(\hat{y}) = -\left(y \cdot \ln(\hat{y}) + (1-y) \cdot \ln(1-\hat{y})\right)$$

Equation 5-1 Cross-entropy loss function

Substituting the cross-entropy loss function into our expression for the error term of the output neuron yields Equation 5-2:

$$\frac{\partial e}{\partial z_f} = \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_f} = -\left(\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot S'(z_f) = \hat{y} - y$$

Equation 5-2 Derivative of cross-entropy loss function and derivative of logistic output unit combined into a single expression

We spare you from the algebra needed to arrive at this result, but if you squint your eyes a little bit and remember that the logistic sigmoid function has some e^x terms, and we know that $ln(e^x) = x$ and the derivative of $ln(x) = x^{-1}$, then it does not seem farfetched that our seemingly complicated formulas might end up as something as simple as that. Figure 5-5 shows the equivalent plot for these functions. The *y*-range is increased compared to Figure 5-4 to capture more of the range of the new loss function. Just as discussed, the derivative of the cross-entropy loss function does increase significantly at the right end of the chart, and the resulting product (the green line) now approaches 1 in the case where the neuron is saturated. That is, the backpropagated error is no longer 0, and the weight adjustments will no longer be suppressed.

Although the chart seems promising, you might feel a bit uncomfortable to just start using Equation 5-2 without further explanation. We used the MSE loss function in the first place, you may recall, on the assumption that your likely familiarity with linear regression would make the concept clearer. We even stated that using MSE together with the logistic sigmoid function is not a good choice.

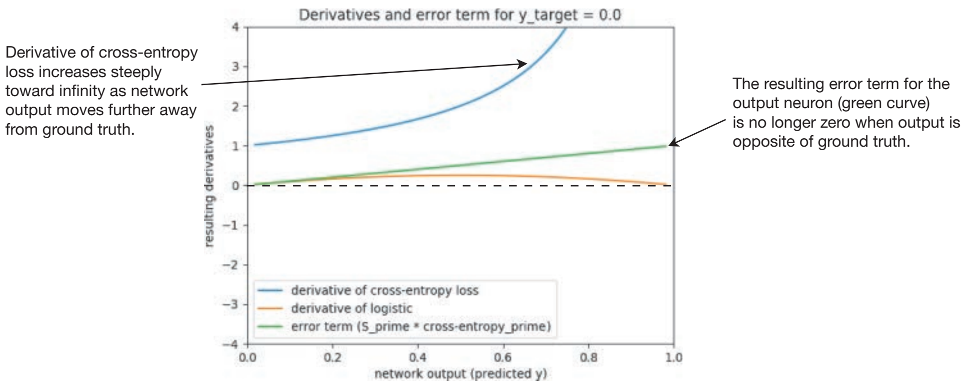

Figure 5-5 Derivatives and error term when using cross-entropy loss function. Ground truth y (denoted y_target in the figure) is 0, as in Figure 5-4.

We have now seen in Figure 5-4 why this is the case. Still, let us at least give you some insight into why using the cross-entropy loss function instead of the MSE loss function is acceptable. Figure 5-6 shows how the value of the MSE and cross-entropy loss function varies as the output of the neuron changes from 0 to 1 in the case of a ground truth of 0. As you can see, as *y* moves further away from the true value, both MSE and the cross-entropy function increase in value, which is the behavior that we want from a loss function.

Intuitively, by looking at the chart in Figure 5-6, it is hard to argue that one function is better than the other, and because we have already shown in Figure 5-4 that MSE is not a good function, you can see the benefit of using the cross-entropy loss function instead. One thing to note is that, from a mathematical perspective, it does not make sense to use the cross-entropy loss function together with a tanh neuron because the logarithm for negative numbers is not defined.

As further reading, we recommend learning about information theory and maximum-likelihood estimation, which provides a rationale for the use of the cross-entropy loss function.

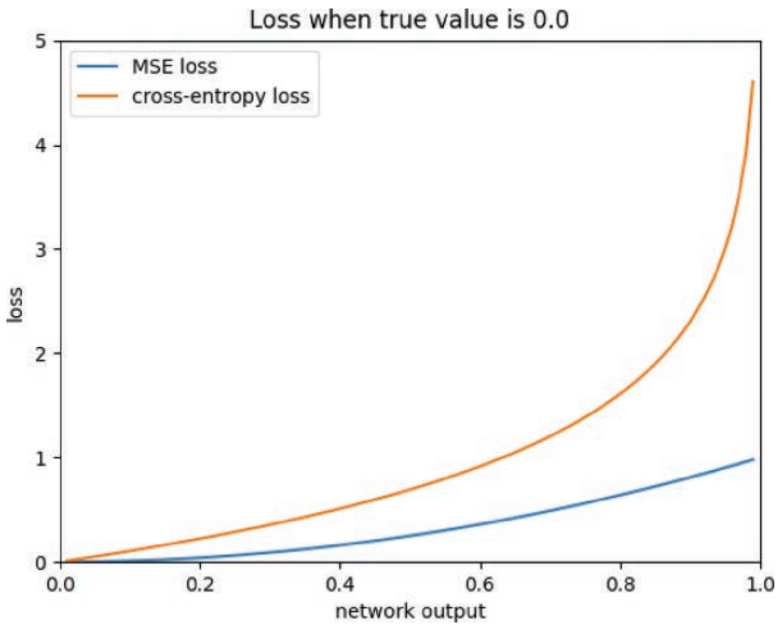

Figure 5-6 Value of the mean squared error (blue) and cross-entropy loss (orange) functions as the network output $\hat{y}$ changes (horizontal axis). The assumed ground truth is 0.

In the preceding examples, we assumed a ground truth of 0. For completeness, Figure 5-7 shows how the derivatives behave in the case of a ground truth of 1.

The resulting charts are flipped in both directions, and the MSE function shows exactly the same problem as for the case when ground truth was 0. Similarly, the cross-entropy loss function solves the problem in this case as well.

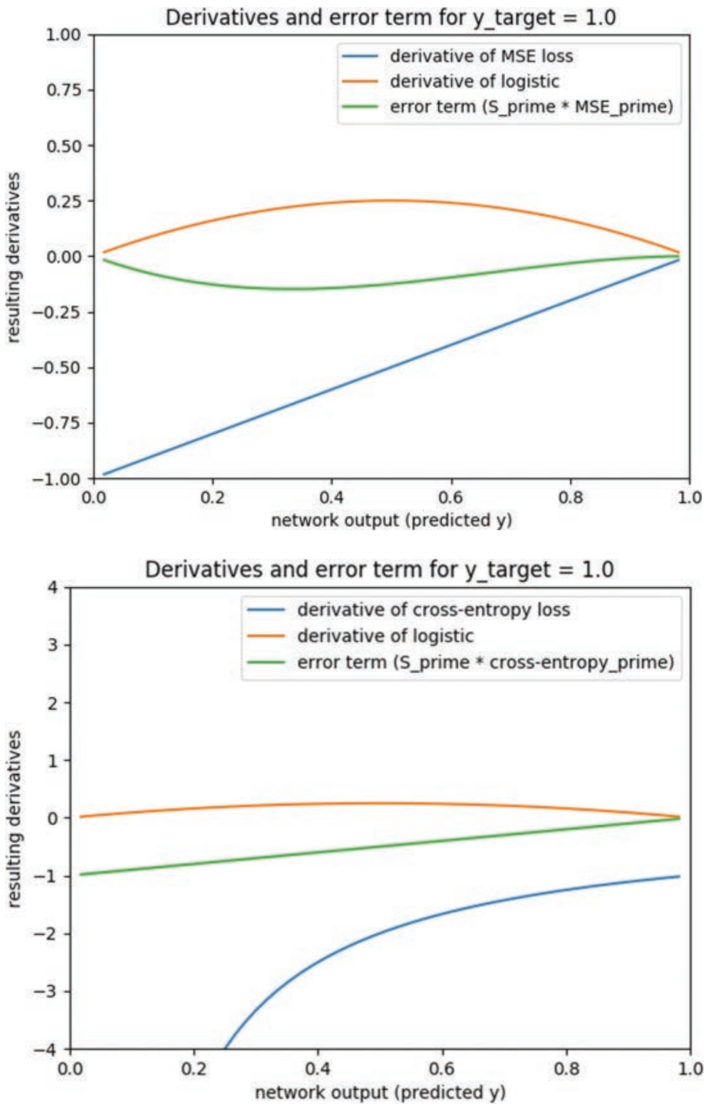

Figure 5-7 Behavior of the different derivatives when assuming a ground truth of 1. Top: Mean squared error loss function. Bottom: Cross-entropy loss function.

COMPUTER IMPLEMENTATION OF THE CROSS-ENTROPY LOSS FUNCTION

If you find an existing implementation of a code snippet that calculates the cross-entropy loss function, then you might be confused at first because it does not resemble what is stated in Equation 5-1. A typical implementation can look like that in Code Snippet 5-8. The trick is that, because we know that y in Equation 5-1 is either 1.0 or 0.0, the factors y and (1-*y*) will serve as an `if` statement and select one of the *ln* statements.

Code Snippet 5-8 Python Implementation of the Cross-Entropy Loss Function

```
def cross_entropy(y_truth, y_predict):
        if y_truth == 1.0:
            return -np.log(y_predict)
        else:
            return -np.log(1.0-y_predict)
```

Apart from what we just described, there is another thing to consider when implementing backpropagation using the cross-entropy loss function in a computer program. It can be troublesome if you first compute the derivative of the cross-entropy loss (as in Equation 5-2) and then multiply by the derivative of the activation function for the output unit. As shown in Figure 5-5, in certain points, one of the functions approaches 0 and one approaches infinity, and although this mathematically can be simplified to the product approaching 1, due to rounding errors, a numerical computation might not end up doing the right thing. The solution is to analytically simplify the product to arrive at the combined expression in Equation 5-2, which does not suffer from this problem.

In reality, we do not need to worry about these low-level details because we are using a DL framework. Code Snippet 5-9 shows how we can tell Keras to use the cross-entropy loss function for a binary classification problem. We simply state `loss='binary_crossentropy'` as an argument to the `compile` function.

Code Snippet 5-9 Use Cross-Entropy Loss for a Binary Classification Problem in TensorFlow

```
model.compile(loss='binary_crossentropy',
              optimizer = optimizer_type,
              metrics =['accuracy'])
```

In Chapter 6, “Fully Connected Networks Applied to Regression,” we detail the formula for the categorical cross-entropy loss function, which is used for multiclass classification problems. In TensorFlow, it is as simple as stating `loss='categorical_crossentropy'`.

Different Activation Functions to Avoid Vanishing Gradient in Hidden Layers

The previous section showed how we can solve the problem of saturated neurons in the output layer by choosing a different loss function. However, this does not help for the hidden layers. The hidden neurons can still be saturated, resulting in derivatives close to 0 and vanishing gradients. At this point, you may wonder if we are solving the problem or just fighting symptoms. We have modified (standardized) the input data, used elaborate techniques to initialize the weights based on the number of inputs and outputs, and changed our loss function to accommodate the behavior of our activation function. Could it be that the activation function itself is the cause of the problem?

How did we end up with the tanh and logistic sigmoid functions as activation functions anyway? We started with early neuron models from McCulloch and Pitts (1943) and Rosenblatt (1958) that were both binary in nature. Then Rumelhart, Hinton, and Williams (1986) added the constraint that the activation function needs to be differentiable, and we switched to the tanh and logistic sigmoid functions. These functions kind of look like the sign function yet are still differentiable, but what good is a differentiable function in our algorithm if its derivative is 0 anyway?

Based on this discussion, it makes sense to explore alternative activation functions. One such attempt is shown in Figure 5-8, where we have complicated the activation function further by adding a linear term 0.2**x* to the output to prevent the derivative from approaching 0.

Although this function might well do the trick, it turns out that there is no good reason to overcomplicate things, so we do not need to use this function. We remember from the charts in the previous section that a derivative of 0 was a problem only in one direction because, in the other direction, the output value already matched the ground truth anyway. In other words, it is fine with a derivative of 0 on one side of the chart. Based on this reasoning, we can consider

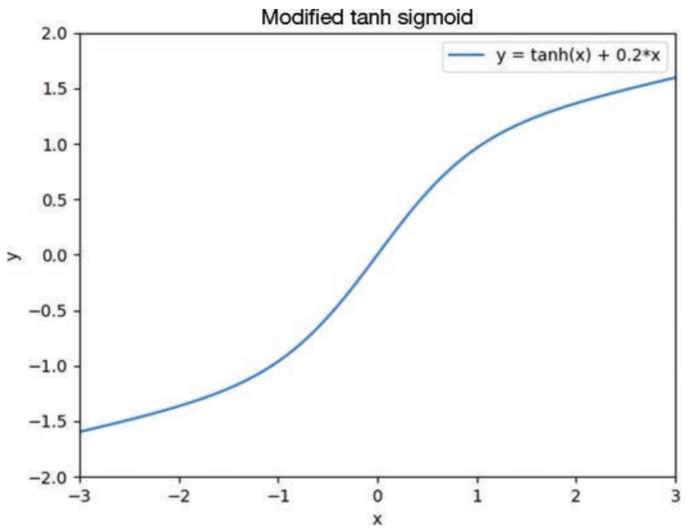

Figure 5-8 Modified tanh function with an added linear term

the rectified linear unit (ReLU) activation function in Figure 5-9, which has been shown to work for neural networks (Glorot, Bordes, and Bengio, 2011).

Now, a fair question is how this function can possibly be used after our entire obsession with differentiable functions. The function in Figure 5-9 is not

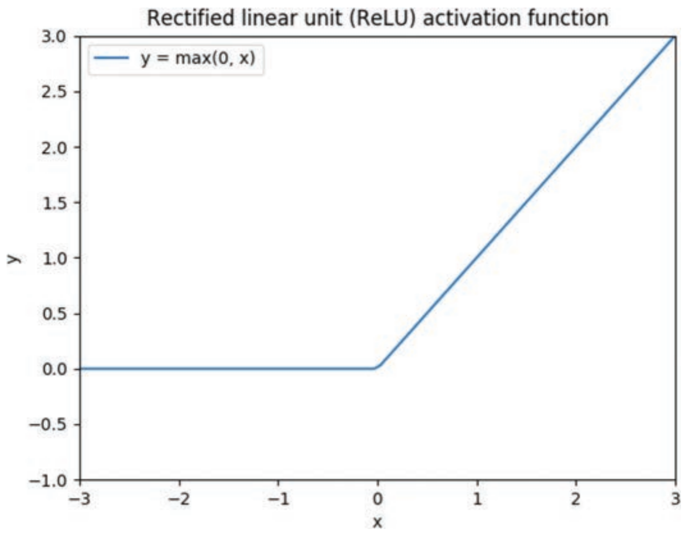

Figure 5-9 Rectified linear unit (ReLU) activation function

differentiable at $x = 0$. However, this does not present a big problem. It is true that from a mathematical point of view, the function is not differentiable in that one point, but nothing prevents us from just defining the derivative as 1 in that point and then trivially using it in our backpropagation algorithm implementation. The key issue to avoid is a function with a discontinuity, like the sign function. Can we simply remove the kink in the line altogether and use $y = x$ as an activation function? The answer is that this does not work. If you do the calculations, you will discover that this will let you collapse the entire network into a linear function and, as we saw in Chapter 1, "The Rosenblatt Perceptron," a linear function (like the perceptron) has severe limitations. It is even common to refer to the activation function as a *nonlinearity,* which stresses how important it is to not pick a linear function as an activation function.

The **activation function** should be **nonlinear** and is even often referred to as a **nonlinearity** instead of activation function.

An obvious benefit with the ReLU function is that it is cheap to compute. The implementation involves testing only whether the input value is less than 0, and if so, it is set to 0. A potential problem with the ReLU function is when a neuron starts off as being saturated in one direction due to a combination of how the weights and inputs happen to interact. Then that neuron will not participate in the network at all because its derivative is 0. In this situation, the neuron is said to be dead. One way to look at this is that using ReLUs gives the network the ability to remove certain connections altogether, and it thereby builds its own network topology, but it could also be that it accidentally kills neurons that could be useful if they had not happened to die. Figure 5-10 shows a variation of the ReLU function known as *leaky ReLU,* which is defined so its derivative is never 0.

Given that humans engage in all sorts of activities that arguably kill their brain cells, it is reasonable to ask whether we should prevent our network from killing its neurons, but that is a deeper discussion.

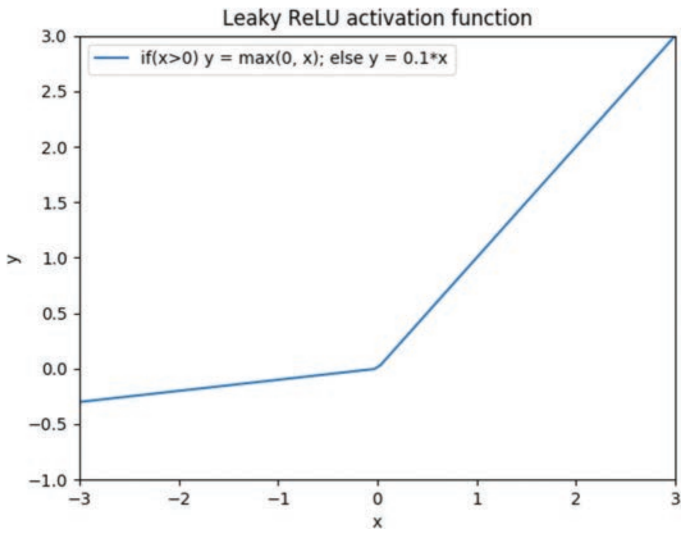

Figure 5-10 Leaky rectified linear unit (ReLU) activation function

All in all, the number of activation functions we can think of is close to unlimited, and many of them work equally well. Figure 5-11 shows a number of important activation functions that we should add to our toolbox. We have already seen tanh, ReLU, and leaky ReLU (Xu, Wang, et al., 2015). We now add the softplus function (Dugas et al., 2001), the exponential linear unit also known as *elu* (Shah et al., 2016), and the maxout function (Goodfellow et al., 2013). The maxout function is a generalization of the ReLU function in which, instead of taking the max value of just two lines (a horizontal line and a line with positive slope), it takes the max value of an arbitrary number of lines. In our example, we use three lines, one with a negative slope, one that is horizontal, and one with a positive slope.

All of these activation functions except for tanh should be effective at fighting vanishing gradients when used as *hidden units*. There are also some alternatives to the logistic sigmoid function for the *output units,* but we save that for Chapter 6.

The **tanh, ReLU, leaky ReLU, softplus, elu,** and **maxout** functions can all be considered for hidden units, but **tanh** has a problem with **vanishing gradients.**

There is no need to memorize the formulas for the activation functions at this point, but just focus on their shape.

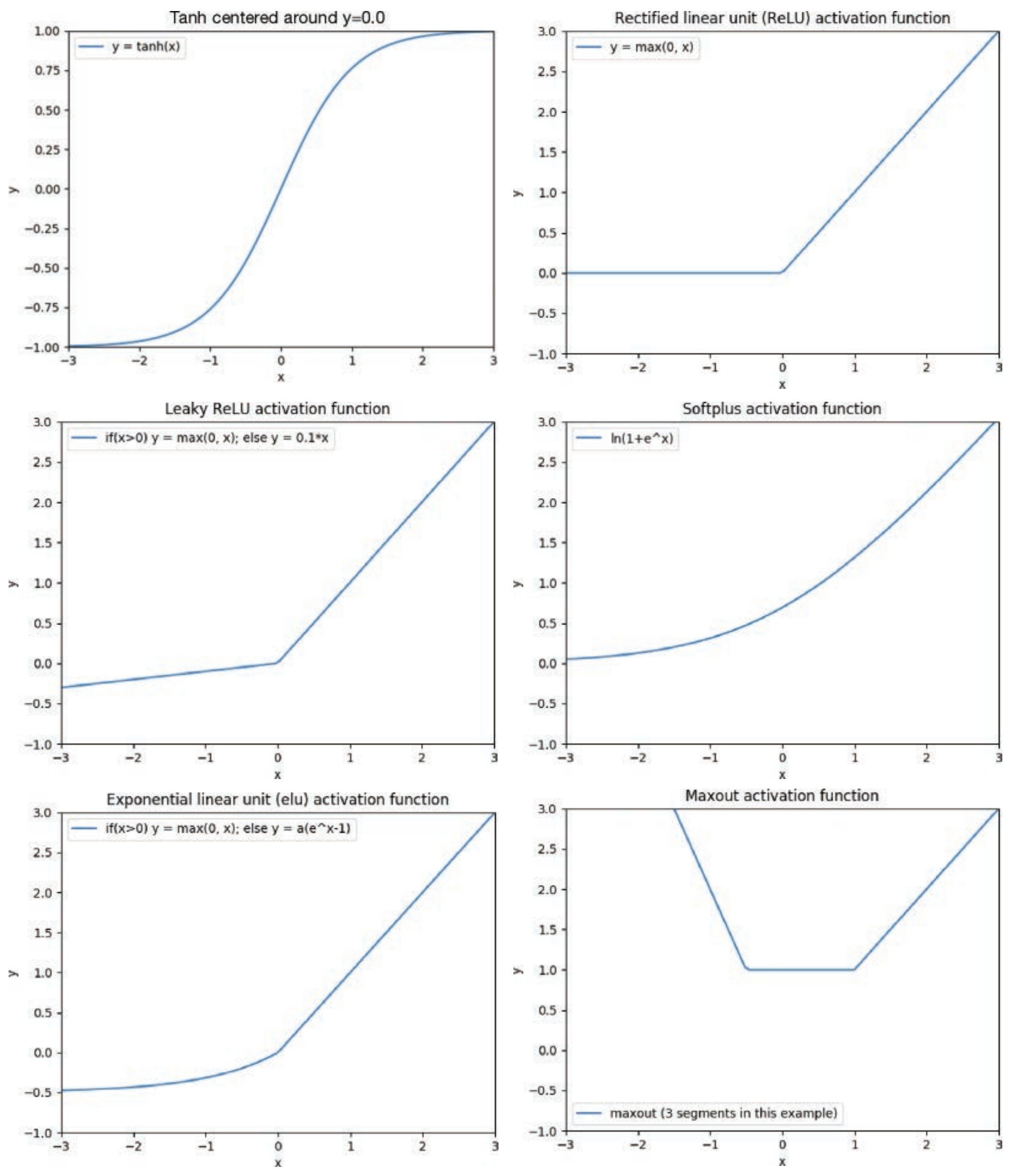

Figure 5-11 Important activation functions for hidden neurons. Top row: tanh, ReLU. Middle row: leaky ReLU, softplut. Bottom row: elu, maxout.

We saw previously how we can choose tanh as an activation function for the neurons in a layer in TensorFlow, also shown in Code Snippet 5-10.

Code Snippet 5-10 Setting the Activation Function for a Layer

```
keras.layers.Dense(25, activation='tanh',
                   kernel_initializer=initializer,
                   bias_initializer='zeros'),
```

If we want a different activation function, we simply replace `'tanh'` with one of the other supported functions (e.g., `'sigmoid'`, `'relu'`, or `'elu'`). We can also omit the `activation` argument altogether, which results in a layer without an activation function; that is, it will just output the weighted sum of the inputs. We will see an example of this in Chapter 6.

Variations on Gradient Descent to Improve Learning

There are a number of variations on gradient descent aiming to enable better and faster learning. One such technique is momentum, where in addition to computing a new gradient every iteration, the new gradient is combined with the gradient from the previous iteration. This can be likened with a ball rolling down a hill where the direction is determined not only by the slope in the current point but also by how much momentum the ball has picked up, which was caused by the slope in previous points. Momentum can enable faster convergence due to a more direct path in cases where the gradient is changing slightly back and forth from point to point. It can also help with getting out of a local minimum. One example of a momentum algorithm is Nesterov momentum (Nesterov, 1983).

Nesterov momentum, AdaGrad, RMSProp, and **Adam** are important variations (also known as *optimizers*) on gradient descent and stochastic gradient descent.

Another variation is to use an adaptive learning rate instead of a fixed learning rate, as we have used previously. The learning rate adapts over time on the basis of historical values of the gradient. Two algorithms using adaptive learning rate are *adaptive gradient,* known as *AdaGrad* (Duchi, Hazan, and Singer, 2011), and *RMSProp* (Hinton, n.d.). Finally, *adaptive moments,* known as *Adam* (Kingma and Ba, 2015), combines both adaptive learning rate and momentum. Although these algorithms adaptively modify the learning rate, we still have to set an initial learning rate. These algorithms even introduce a number of additional parameters that control how the algorithms perform, so we now have even more parameters to tune for our model. However, in many cases, the default values work well.

We do not go into the details of how to implement momentum and adaptive learning rate; we simply use implementations available in the DL framework. Understanding these techniques is important when tuning your models, so consider exploring these topics. You can find them summarized in *Deep Learning* (Goodfellow, Bengio, and Courville, 2016), or you can read the original sources (Duchi, Hazan, and Singer, 2011; Hinton, n.d.; Kingma and Ba, 2015; Nesterov, 1983).

Finally, we discussed earlier how to avoid vanishing gradients, but there can also be a problem with exploding gradients, where the gradient becomes too big in some point, causing a huge step size. It can cause weight updates that completely throw off the model. Gradient clipping is a technique to avoid exploding gradients by simply not allowing overly large values of the gradient in the weight update step. Gradient clipping is available for all optimizers in Keras.

Gradient clipping is used to avoid the problem of **exploding gradients.**

Code Snippet 5-11 shows how we set an optimizer for our model in Keras. The example shows stochastic gradient descent with a learning rate of 0.01 and no other bells and whistles.

Code Snippet 5-11 Setting an Optimizer for the Model

```
opt = keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0,
                           nesterov=False)
model.compile(loss='mean_squared_error', optimizer = opt,
              metrics =['accuracy'])
```

Just as we can for initializers, we can choose a different optimizer by declaring any one of the supported optimizers in Tensorflow, such as the three we just described:

```
opt = keras.optimizers.Adagrad(lr=0.01, epsilon=None)
opt = keras.optimizers.RMSprop(lr=0.001, rho=0.8, epsilon=None)
opt = keras.optimizers.Adam(lr=0.01, epsilon=0.1, decay=0.0)
```

In the example, we freely modified some of the arguments and left out others, which will then take on the default values. If we do not feel the need to modify the default values, we can just pass the name of the optimizer to the model `compile` function, as in Code Snippet 5-12.

Code Snippet 5-12 Passing the Optimizer as a String to the Compile Function

```
model.compile(loss='mean_squared_error', optimizer ='adam',
              metrics =['accuracy'])
```

We now do an experiment in which we apply some of these techniques to our neural network.

Experiment: Tweaking Network and Learning Parameters

To illustrate the effect of the different techniques, we have defined five different configurations, shown in Table 5-1. Configuration 1 is the same network that we studied in Chapter 4 and at beginning of this chapter. Configuration 2 is the same network but with a learning rate of 10.0. In configuration 3, we change the initialization method to Glorot uniform and change the optimizer to Adam with all parameters taking on the default values. In configuration 4, we change the activation function for the hidden units to ReLU, the initializer for the hidden layer to He normal, and the loss function to cross-entropy. When we described the cross-entropy loss function earlier, it was in the context of a binary classification problem, and the output neuron used the logistic sigmoid function. For multiclass classification problems, we use the categorical cross-entropy loss function, and it is paired with a different output activation known as *softmax*. The details of softmax are described in Chapter 6, but we use it here with the categorical

Table 5-1 Configurations with Tweaks to Our Network

CONFIGURATION	HIDDEN ACTIVATION	HIDDEN INITIALIZER	OUTPUT ACTIVATION	OUTPUT INITIALIZER	LOSS FUNCTION	OPTIMIZER	MINI-BATCH SIZE
Conf1	tanh	Uniform 0.1	Sigmoid	Uniform 0.1	MSE	SGD lr=0.01	1
Conf2	tanh	Uniform 0.1	Sigmoid	Uniform 0.1	MSE	SGD lr=10.0	1
Conf3	tanh	Glorot uniform	Sigmoid	Glorot uniform	MSE	Adam	1
Conf4	ReLU	He normal	Softmax	Glorot uniform	CE	Adam	1
Conf5	ReLU	He normal	Softmax	Glorot uniform	CE	Adam	64

Note: CE, cross-entropy; MSE, mean squared error; SGD, stochastic gradient descent.

cross-entropy loss function. Finally, in configuration 5, we change the mini-batch size to 64.

Modifying the code to model these configurations is trivial using our DL framework. In Code Snippet 5-13, we show the statements for setting up the model for configuration 5, using ReLU units with He normal initialization in the hidden layer and softmax units with Glorot uniform initialization in the output layer. The model is then compiled using categorical cross-entropy as the loss function and Adam as the optimizer. Finally, the model is trained for 20 epochs using a mini-batch size of 64 (set to BATCH_SIZE=64 in the init code).

Code Snippet 5-13 Code Changes Needed for Configuration 5

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(25, activation='relu',
                       kernel_initializer='he_normal',
                       bias_initializer='zeros'),
    keras.layers.Dense(10, activation='softmax',
                         kernel_initializer='glorot_uniform',
                         bias_initializer='zeros')])

model.compile(loss='categorical_crossentropy',
                    optimizer = 'adam',
                    metrics =['accuracy'])

history = model.fit(train_images, train_labels,
                    validation_data=(test_images, test_labels),
                    epochs=EPOCHS, batch_size=BATCH_SIZE,
                    verbose=2, shuffle=True)
```

If you run this configuration on a GPU-accelerated platform, you will notice that it is much faster than the previous configuration. The key here is that we have a batch size of 64, which results in 64 training examples being computed in parallel, as opposed to the initial configuration where they were all done serially.

The results of the experiment are shown in Figure 5-12, which shows how the test errors for all configurations evolve during the training process.

We use Matplotlib to visualize the learning process. A more powerful approach is to use the TensorBoard functionality that is included in TensorFlow. We highly recommend that you get familiar with TensorBoard when you start building and tuning your own models.

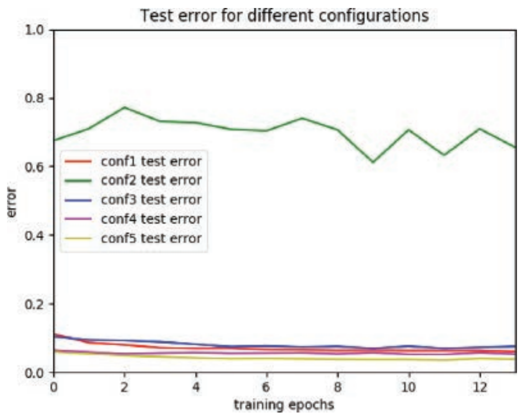

Figure 5-12 Error on the test dataset for the five configurations

Configuration 1 (red line) ends up at an error of approximately 6%. We spent a nontrivial amount of time on testing different parameters to come up with that configuration (not shown in this book).

Configuration 2 (green) shows what happens if we set the learning rate to 10.0, which is significantly higher than 0.01. The error fluctuates at approximately 70%, and the model never learns much.

Configuration 3 (blue) shows what happens if, instead of using our tuned learning rate and initialization strategy, we choose a “vanilla configuration” with Glorot initialization and the Adam optimizer with its default values. The error is approximately 7%.

For Configuration 4 (purple), we switch to using different activation functions and the cross-entropy error function. We also change the initializer for the hidden layer to He normal. We see that the test error is reduced to 5%.

For Configuration 5 (yellow), the only thing we change compared to Configuration 4 is the mini-batch size: 64 instead of 1. This is our best configuration, which ends up with a test error of approximately 4%. It also runs much faster than the other configurations because the use of a mini-batch size of 64 enables more examples to be computed in parallel.

Although the improvements might not seem that impressive, we should recognize that reducing the error from 6% to 4% means removing one-third of the error cases, which definitely is significant. More important, the presented techniques enable us to train deeper networks.

Hyperparameter Tuning and Cross-Validation

The programming example showed the need to tune different hyperparameters, such as the activation function, weight initializer, optimizer, mini-batch size, and loss function. In the experiment, we presented five configurations with some different combinations, but clearly there are many more combinations that we could have evaluated. An obvious question is how to approach this *hyperparameter tuning* process in a more systematic manner. One popular approach is known as *grid search* and is illustrated in Figure 5-13 for the case of two hyperparameters (optimizer and initializer). We simply create a grid with each axis representing a

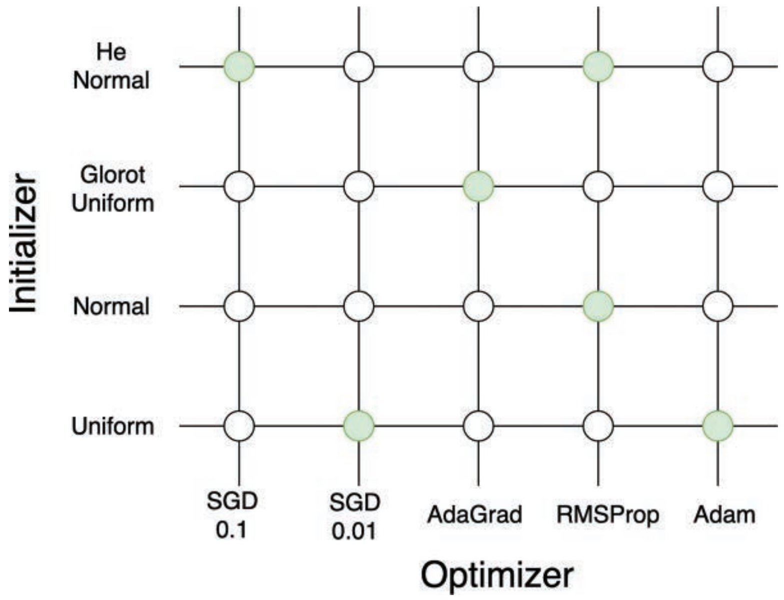

Figure 5-13 Grid search for two hyperparameters. An exhaustive grid search would simulate all combinations, whereas a random grid search might simulate only the combinations highlighted in green.

single hyperparameter. In the case of two hyperparameters, it becomes a 2D grid, as shown in the figure, but we can extend it to more dimensions, although we can only visualize, at most, three dimensions. Each intersection in the grid (represented by a circle) represents a combination of different hyperparameter values, and together, all the circles represent all possible combinations. We then simply run an experiment for each data point in the grid to determine what is the best combination.

What we just described is known as *exhaustive* grid search, but needless to say, it can be computationally expensive as the number of combinations quickly grows with the number of hyperparameters that we want to evaluate. An alternative is to do a random grid search on a randomly selected a subset of all combinations. This alternative is illustrated in the figure by the green dots that represent randomly chosen combinations. We can also do a hybrid approach in which we start with a random grid search to identify one or a couple of promising combinations, and then we can create a finer-grained grid around those combinations and do an exhaustive grid search in this zoomed-in part of the search space. Grid search is not the only method available for hyperparameter tuning. For hyperparameters that are differentiable, it is possible to do a gradient-based search, similar to the learning algorithm used to tune the normal parameters of the model.

Implementing grid search is straightforward, but a common alternative is to use a framework known as *sci-kit learn*.[3] This framework plays well with Keras. At a high level, we wrap our call to `model.fit()` into a function that takes hyperparameters as input values. We then provide this wrapper function to sci-kit learn, which will call it in a systematic manner and monitor the training process. The sci-kit learn framework is a general ML framework and can be used with both traditional ML algorithms as well as DL.

USING A VALIDATION SET TO AVOID OVERFITTING

The process of hyperparameter tuning introduces a new risk of overfitting. Consider the example earlier in the chapter where we evaluated five configurations on our test set. It is tempting to believe that the measured error on our test dataset is a good estimate of what we will see on not-yet-seen data. After all, we did not use the test dataset during the training process, but there is a subtle issue with this reasoning. Even though we did not use the test set to train the weights of the model, we did use the test set when deciding which set of hyperparameters performed best. Therefore, we run the risk of having picked a set of hyperparameters that are particularly good for the test dataset but not as good for the general case. This is somewhat subtle in that the risk of overfitting exists even if we do not have a feedback loop in which results from one set of hyperparameters guide the experiment of a next set of hyperparameters. This risk exists even if we decide on all combinations up front and only use the test dataset to select the best performing model.

We can solve this problem by splitting up our dataset into a training dataset, a validation dataset, and a test dataset. We train the weights of our model using the training dataset, and we tune the hyperparameters using our validation dataset. Once we have arrived at our final model, we use our test dataset to determine how well the model works on not-yet-seen data. This process is illustrated in the left part of Figure 5-14. One challenge is to decide how much of the original dataset to use as training, validation, and test set. Ideally, this is determined on a case-by-case basis and depends on the variance in the data distribution. In absence of any such information, a common split between training set and test set when there is no need for a validation set is 70/30 (70% of original data used for training and 30% used for test) or 80/20. In cases where we need a validation set for hyperparameter tuning, a typical split is 60/20/20. For datasets with low variance, we can get away with a smaller fraction being used for validation, whereas if the variance is high, a larger fraction is needed.

3. https://scikit-learn.org

CROSS-VALIDATION TO IMPROVE USE OF TRAINING DATA

One unfortunate effect of introducing the validation set is that we can now use only 60% of the original data to train the weights in our network. This can be a problem if we have a limited amount of training data to begin with. We can address this problem using a technique known as *cross-validation*, which avoids holding out parts of the dataset to be used as validation data but at the expense of additional computation. We focus on one of the most popular cross-validation techniques, known as *k-fold cross-validation*. We start by splitting our data into a training set and a test set, using something like an 80/20 split. The test set is not used for training or hyperparameter tuning but is used only in the end to establish how good the final model is. We further split our training dataset into *k* similarly sized pieces known as *folds*, where a typical value for *k* is a number between 5 and 10.

We can now use these folds to create k instances of a training set and validation set by using $k - 1$ folds for training and 1 fold for validation. That is, in the case of $k = 5$, we have five alternative instances of training/validations sets. The first one uses folds 1, 2, 3, and 4 for training and fold 5 for validation, the second instance uses folds 1, 2, 3, and 5 for training and fold 4 for validation, and so on.

Let us now use these five instances of train/validation sets to both train the weights of our model and tune the hyperparameters. We use the example presented earlier in the chapter where we tested a number of different configurations. Instead of training each configuration once, we instead train each configuration *k* times with our *k* different instances of train/validation data. Each of these *k* instances of the same model is trained from scratch, without reusing weights that were learned by a previous instance. That is, for each configuration, we now have *k* measures of how well the configuration performs. We now compute the average of these measures for each configuration to arrive at a single number for each configuration that is then used to determine the best-performing configuration.

Now that we have identified the best configuration (the best set of hyperparameters), we again start training this model from scratch, but this time we use all of the *k* folds as training data. When we finally are done training this best-performing configuration on all the training data, we can run the model on the test dataset to determine how well it performs on not-yet-seen data. As noted earlier, this process comes with additional computational cost because we must train each configuration *k* times instead of a single time. The overall process is illustrated on the right side of Figure 5-14.

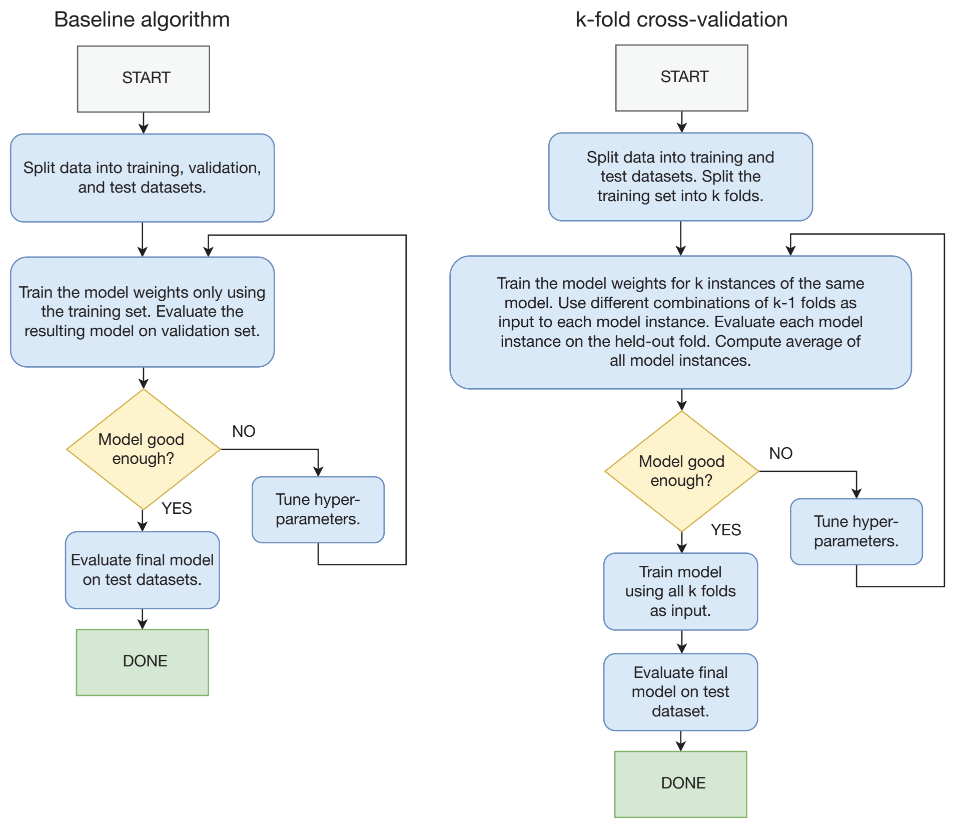
Baseline algorithm
START
Split data into training, validation, and test datasets.
Train the model weights only using the training set. Evaluate the resulting model on validation set.
Model good enough?
NO
Tune hyper-parameters.
YES
Evaluate final model on test datasets.
DONE
k-fold cross-validation
START
Split data into training and test datasets. Split the training set into k folds.
Train the model weights for k instances of the same model. Use different combinations of k-1 folds as input to each model instance. Evaluate each model instance on the held-out fold. Compute average of all model instances.
Model good enough?
NO
Tune hyper-parameters.
YES
Train model using all k folds as input.
Evaluate final model on test dataset.
DONE

Figure 5-14 Tuning hyperparameters with a validation dataset (left) and using k-fold cross-validation (right)

We do not go into the details of why cross-validation works, but for more information, you can consult The Elements of Statistical Learning (Hastie, Tibshirani, and Friedman, 2009).

Concluding Remarks on the Path Toward Deep Learning

This chapter introduced the techniques that are regarded as enablers of the DL revolution that started with the AlexNet paper (Krizhevsky, Sutskever, and Hinton, 2012). In particular, the emergence of large datasets, the introduction of the ReLU unit and the cross-entropy loss function, and the availability of low-cost GPU-powered high-performance computing are all viewed as critical components that had to come together to enable deeper models to learn (Goodfellow et al., 2016).

We also demonstrated how to use a DL framework instead of implementing our models from scratch. The emergence of these DL frameworks is perhaps equally important when it comes to enabling the adoption of DL, especially in the industry.

With this background, we are now ready to move on to Chapter 6 and build our first deep neural network!

This page intentionally left blank

Index