Josh Patterson &
Adam Gibson

O'REILLY®

Deep Learning

Although interest in machine learning has reached a high point, lofty expectations often scuttle projects before they get very far. How can machine learning—especially deep neural networks—make a real difference in your organization? This hands-on guide not only provides the most practical information available on the subject, but also helps you get started building efficient deep learning networks.

Authors Josh Patterson and Adam Gibson provide the fundamentals of deep learning—tuning, parallelization, vectorization, and building pipelines—that are valid for any library before introducing the open source Deeplearning4j (DL4J) library for developing production-class workflows. Through real-world examples, you'll learn methods and strategies for training deep network architectures and running deep learning workflows on Spark and Hadoop with DL4J.

- Dive into machine learning concepts in general, as well as deep learning in particular
- Understand how deep networks evolved from neural network fundamentals
- Explore the major deep network architectures, including Convolutional and Recurrent
- Learn how to map specific deep networks to the right problem
- Walk through the fundamentals of tuning general neural networks and specific deep network architectures
- Use vectorization techniques for different data types with DataVec, DL4J's workflow tool
- Learn how to use DL4J natively on Spark and Hadoop

"Everything a de
to know to get s
with deep learn
real world."

—G
(

Josh Patterson is cu
of Field Engineering f
Previously, Josh work
Principal Solutions Ar
Cloudera and as a ma
and distributed syste
at the Tennessee Vall

Adam Gibson is the
Skymind. Adam has w
Fortune 500 compar
funds, PR firms, and s
accelerators to create
machine learning pro
has a strong track rec
companies handle an
realtime data.

US $49.99 CAN $65.99
[illegible]

Deep Learning

A Practitioner's Approach

Josh Patterson and Adam Gibson

Beijing Boston Farnham Sebastopol Tokyo

Deep Learning
by Josh Patterson and Adam Gibson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/insti- tutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Tim McGovern **Production Editor:** Nicholas Adams
Copyeditor: Bob Russell, Octal Publishing, Inc. **Proofreader:** Christina Edwards

August 2017: First Edition

Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

Revision History for the First Edition

2017-07-27: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491914250* for release details.

978-1-491-91425-0

[M]

For my sons Ethan, Grin, and Dane: Go forth, be persistent, be bold. —J. Patterson

Table of Contents

Preface

What’s in This Book?

The first four chapters of this book are focused on enough theory and fundamentals to give you, the practitioner, a working foundation for the rest of the book. The last five chapters then work from these concepts to lead you through a series of practical paths in deep learning using DL4J:

- Building deep networks
- Advanced tuning techniques
- Vectorization for different data types
- Running deep learning workflows on Spark

DL4J as Shorthand for Deeplearning4j

We use the names DL4J and Deeplearning4j interchangeably in this book. Both terms refer to the suite of tools in the Deeplearning4j library.

We designed the book in this manner because we felt there was a need for a book cov- ering “enough theory” while being practical enough to build production-class deep learning workflows. We feel that this hybrid approach to the book’s coverage fits this space well.

Chapter 1 is a review of machine learning concepts in general as well as deep learning in particular, to bring any reader up to speed on the basics needed to understand the rest of the book. We added this chapter because many beginners can use a refresher or primer on these concepts and we wanted to make the project accessible to the larg- est audience possible.

Chapter 2 builds on the concepts from Chapter 1 and gives you the foundations of neural networks. It is largely a chapter in neural network theory but we aim to

present the information in an accessible way. Chapter 3 further builds on the first two chapters by bringing you up to speed on how deep networks evolved from the funda- mentals of neural networks. Chapter 4 then introduces the four major architectures of deep networks and provides you with the foundation for the rest of the book.

In Chapter 5, we take you through a number of Java code examples using the techni- ques from the first half of the book. Chapters 6 and 7 examine the fundamentals of tuning general neural networks and then how to tune specific architectures of deep networks. These chapters are platform-agnostic and will be applicable to the practi- tioner of any deep learning library. Chapter 8 is a review of the techniques of vectori- zation and the basics on how to use DataVec (DL4J's ETL and vectorization workflow tool). Chapter 9 concludes the main body of the book with a review on how to use DL4J natively on Spark and Hadoop and illustrates three real examples that you can run on your own Spark clusters.

The book has many Appendix chapters for topics that were relevant yet didn't fit directly in the main chapters. Topics include:

- Artificial Intelligence
- Using Maven with DL4J projects
- Working with GPUs
- Using the ND4J API
- and more

Who Is "The Practitioner"?

Today, the term "data science" has no clean definition and often is used in many dif- ferent ways. The world of data science and artificial intelligence (AI) is as broad and hazy as any terms in computer science today. This is largely because the world of machine learning has become entangled in nearly all disciplines.

This widespread entanglement has historical parallels to when the World Wide Web (90s) wove HTML into every discipline and brought many new people into the land of technology. In the same way, all types—engineers, statisticians, analysts, artists— are entering the machine learning fray every day. With this book, our goal is to democratize deep learning (and machine learning) and bring it to the broadest audi- ence possible.

If you find the topic interesting and are reading this preface—*you are the practitioner, and this book is for you.*

Who Should Read This Book?

As opposed to starting out with toy examples and building around those, we chose to start the book with a series of fundamentals to take you on a full journey through deep learning.

We feel that too many books leave out core topics that the enterprise practitioner often needs for a quick review. Based on our machine learning experiences in the field, we decided to lead-off with the materials that entry-level practitioners often need to brush up on to better support their deep learning projects.

You might want to skip Chapters 1 and 2 and get right to the deep learning funda- mentals. However, we expect that you will appreciate having the material up front so that you can have a smooth glide path into the more difficult topics in deep learning that build on these principles. In the following sections, we suggest some reading strategies for different backgrounds.

The Enterprise Machine Learning Practitioner

We split this category into two subgroups:

- Practicing data scientist
- Java engineer

The practicing data scientist

This group typically builds models already and is fluent in the realm of data sci- ence. If this is you, you can probably skip Chapter 1 and you'll want to lightly skim Chapter 2. We suggest moving on to Chapter 3 because you'll probably be ready to jump into the fundamentals of deep networks.

The Java engineer

Java engineers are typically tasked with integrating machine learning code with pro- duction systems. If this is you, starting with Chapter 1 will be interesting for you because it will give you a better understanding of the vernacular of data science. Appendix E should also be of keen interest to you because integration code for model scoring will typically touch ND4J's API directly.

The Enterprise Executive

Some of our reviewers were executives of large Fortune 500 companies and appreci- ated the content from the perspective of getting a better grasp on what is happening in deep learning. One executive commented that it had "been a minute" since college, and Chapter 1 was a nice review of concepts. If you're an executive, we suggest that you begin with a quick skim of Chapter 1 to reacclimate yourself to some terminol- ogy. You might want to skip the chapters that are heavy on APIs and examples, how- ever.

The Academic

If you're an academic, you likely will want to skip Chapters 1 and 2 because graduate school will have already covered these topics. The chapters on tuning neural net- works in general and then architecture-specific tuning will be of keen interest to you because this information is based on research and transcends any specific deep learn- ing implementation. The coverage of ND4J will also be of interest to you if you prefer to do high-performance linear algebra on the Java Virtual Machine (JVM).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program ele- ments such as variable or function names, databases, data types, environment variables, statements, and keywords. Also used for module and package names, and to show commands or other text that should be typed literally by the user and the output of commands.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values deter- mined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element signifies a warning or caution.

Using Code Examples

Supplemental material (virtual machine, data, scripts, and custom command-line tools, etc.) is available for download at *https://github.com/deeplearning4j/oreilly-book dl4j-examples*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a signifi- cant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Deep Learning: A Practitioner's Approach* by Josh Patterson and Adam Gibson (O'Reilly). Copyright 2017 Josh Patter- son and Adam Gibson, 978-1-4919-1425-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Administrative Notes

In Java code examples, we often omit the import statements. You can see the full import listings in the actual code repository. The API information for DL4J, ND4J, DataVec, and more is available on this website:

http://deeplearning4j.org/documentation

You can find all code examples at:

https://github.com/deeplearning4j/oreilly-book-dl4j-examples

For more resources on the DL4J family of tools, check out this

website: *http://deeplearning4j.org*

O'Reilly Safari

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookques- tions@oreilly.com*. If you find any errors or glaring omissions, if you find anything confusing, or if you have any ideas for improving the book, please email Josh Patter- son at *jpatterson@floe.tv*.

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/deep_learning_oreilly*.

For more information about our books, courses, conferences, and news, see our web- site at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Follow Josh Patterson on Twitter: @jpatanooga

Follow Adam Gibson on Twitter: @agibsonccc

Acknowledgments

Josh

I relied on many folks far smarter than myself to help shape the ideas and review the content in this book. No project the size of DL4J operates in a vacuum, and I relied on many of the community experts and engineers at Skymind to construct many of the ideas and guidelines in this book.

Little did I know that hacking on what became DL4J with Adam (after a chance meet- ing at MLConf) would end up as a book. To be fair, while I was there from the begin- ning on DL4J, Adam did far more commits than I ever did. So, to Adam I am considerably grateful in the commitment to the project, commitment to the idea of deep learning on the JVM, and to staying the course during some considerably uncer- tain early days. And, yes, you were right: ND4J was a good idea.

Writing can be a long, lonely path and I'd like to specifically thank Alex Black for his considerable efforts, not only in reviewing the book, but also for contributing content in the appendixes. Alex's encyclopedia-like knowledge of neural network published literature was key in crafting many of the small details of this book and making sure that all the big and little things were correct. Chapters 6 and 7 just wouldn't be half of what they became without Alex Black.

Susan Eraly was key in helping construct the loss function section and contributed appendix material, as well (many of the equations in this book owe a debt of correct- ness to Susan), along with many detailed review notes. Melanie Warrick was key in reviewing early drafts of the book, providing feedback, and providing notes for the inner workings of Convolutional Neural Networks (CNNs).

David Kale was a frequent ad hoc reviewer and kept me on my toes about many key network details and paper references. Dave was always there to provide the academ- ic's view on how much rigor we needed to provide while understanding what kind of audience we were after.

James Long was a critical ear for my rants on what should or should not be in the book, and was able to lend a practical viewpoint from a practicing statistician's point of view. Many times there was not a clear correct answer regarding how to communi- cate a complex topic, and James was my sounding board for arguing the case from multiple sides. Whereas David Kale and Alex Black would frequently remind me of the need for mathematical rigor, James would often play the rational devil's advocate in just how much of it we needed before we "drown the reader in math."

Vyacheslav "Raver" Kokorin added quality insight to the development of the Natural Language Processing (NLP) and Word2Vec examples.

I'd like to make note of the support we received from our CEO at Skymind, Chris Nicholson. Chris supported this book at every turn and in no small part helped us with the needed time and resources to make this happen.

I would like to thank the people who contributed appendix chapters: Alex Black (Backprop, DataVec), Vyacheslav "Raver" Kokorin (GPUs), Susan Eraly (GPUs), and Ruben Fiszel (Reinforcement Learning). Other reviewers of the book at various stages include Grant Ingersol, Dean Wampler, Robert Chong, Ted Malaska, Ryan Geno, Lars George, Suneel Marthi, Francois Garillot, and Don Brown. Any errors that you might discover in this book should be squarely placed on my doorstep.

I'd like to thank our esteemed editor, Tim McGovern, for the feedback, notes, and just overall patience with a project that spanned years and grew by three chapters. I felt like he gave us the space to get this right, and we appreciate it.

Following are some other folks I'd like to recognize who had an impact on my career leading up to this book: my parents (Lewis and Connie), Dr. Andy Novobiliski (grad school), Dr. Mina Sartipi (thesis advisor), Dr. Billy Harris (graduate algorithms), Dr. Joe Dumas (grad school), Ritchie Carroll (creator of the openPDC), Paul Trachian, Christophe Bisciglia and Mike Olson (for recruiting me to Cloudera), Malcom Ramey (for my first real programming job), The University of Tennessee at Chatta- nooga, and Lupi's Pizza (for feeding me through grad school).

Last, and especially not least, I'd like to thank my wife Leslie and my sons Ethan, Griffin, and Dane for their patience while I worked late, often, and sometimes on vacation.

Adam

I would like to thank my team at Skymind for all the work they piled on in assisting with review of the book and content as we continued to iterate on the book. I would especially like to thank Chris who tolerated my crazy idea of writing a book while attempting to do a startup.

DL4J started in 2013 with a chance meeting with Josh at MLConf and it has grown in to quite the project now used all over the world. DL4J has taken me all over the world and has really opened my world up to tons of new experiences.

Firstly, I would like to thank my coauthor Josh Patterson who did the lion's share of the book and deserves much of the credit. He put in nights and weekends to get the book out the door while I continued working on the codebase and continuing to adapt the content to new features through the years.

Echoing Josh, many of our team mates and contributors who joined early on such as Alex, Melanie, Vyacheslav "Raver" Kokorin, and later on folks like Dave helping us as an extra pair of eyes on the math due diligence.

Tim McGovern has been a great ear for some of my crazy ideas on content for O'Reilly and was also amazing in letting me name the book.

CHAPTER 1

A Review of Machine Learning

To condense fact from the vapor of nuance
—Neal Stephenson, *Snow Crash*

The Learning Machines

Interest in machine learning has exploded over the past decade. You see machine learning in computer science programs, industry conferences, and the *Wall Street Journal* almost daily. For all the talk about machine learning, many conflate what it can do with what they wish it could do. Fundamentally, *machine learning* is using algorithms to extract information from raw data and represent it in some type of *model*. We use this model to infer things about other data we have not yet modeled.

Neural networks are one type of model for machine learning; they have been around for at least 50 years. The fundamental unit of a neural network is a *node*, which is loosely based on the biological neuron in the mammalian brain. The connections between neurons are also modeled on biological brains, as is the way these connec- tions develop over time (with "training"). We'll dig deeper into how these models work over the next two chapters.

In the mid-1980s and early 1990s, many important architectural advancements were made in neural networks. However, the amount of time and data needed to get good results slowed adoption, and thus interest cooled. In the early 2000s computational power expanded exponentially and the industry saw a "Cambrian explosion" of com- putational techniques that were not possible prior to this. Deep learning emerged from that decade's explosive computational growth as a serious contender in the field, winning many important machine learning competitions. The interest has not cooled as of 2017; today, we see deep learning mentioned in every corner of machine learning.

We'll discuss our definition of deep learning in more depth in the section that follows. This book is structured such that you, the practitioner, can pick it up off the shelf and do the following:

• Review the relevant basic parts of linear algebra and machine

learning • Review the basics of neural networks
- Study the four major architectures of deep networks
- Use the examples in the book to try out variations of practical deep networks

We hope that you will find the material practical and approachable. Let's kick off the book with a quick primer on what machine learning is about and some of the core concepts you will need to better understand the rest of the book.

How Can Machines Learn?

To define how machines can learn, we need to define what we mean by "learning." In everyday parlance, when we say learning, we mean something like "gaining knowl- edge by studying, experience, or being taught." Sharpening our focus a bit, we can think of machine learning as using algorithms for acquiring structural descriptions from data examples. A computer learns something about the structures that represent the information in the raw data. Structural descriptions are another term for the models we build to contain the information extracted from the raw data, and we can use those structures or models to predict unknown data. Structural descriptions (or models) can take many forms, including the following:

- Decision trees
- Linear regression
- Neural network weights

Each model type has a different way of applying rules to known data to predict unknown data. Decision trees create a set of rules in the form of a tree structure and linear models create a set of parameters to represent the input data.

Neural networks have what is called a *parameter vector* representing the weights on the connections between the nodes in the network. We'll describe the details of this type of model later on in this chapter.

Machine Learning Versus Data Mining

Data mining has been around for many decades, and like many terms in machine learning, it is misunderstood or used poorly. For the context of this book, we consider the practice of "data mining" to be "extracting information from data." Machine learn- ing differs in that it refers to the algorithms used during data mining for acquiring the structural descriptions from the raw data. Here's a simple way to think of data mining:

- To learn concepts
  - — we need examples of raw data
- Examples are made of rows or instances of the data
  - — Which show specific patterns in the data
- The machine learns concepts from these patterns in the data
  - — Through algorithms in machine learning

Overall, this process can be considered "data mining."

Arthur Samuel, a pioneer in artificial intelligence (AI) at IBM and Stanford, defined machine learning as follows:

> [The f]ield of study that gives computers the ability to learn without being explicitly programmed.

Samuel created software that could play checkers and adapt its strategy as it learned to associate the probability of winning and losing with certain dispositions of the board. That fundamental schema of searching for patterns that lead to victory or defeat and then recognizing and reinforcing successful patterns underpins machine learning and AI to this day.

The concept of machines that can learn to achieve goals on their own has captivated us for decades. This was perhaps best expressed by the modern grandfathers of AI, Stuart Russell and Peter Norvig, in their book *Artificial Intelligence: A Modern Approach*:

> How is it possible for a slow, tiny brain, whether biological or electronic, to perceive, understand, predict, and manipulate a world far larger and more complicated than itself?

This quote alludes to ideas around how the concepts of learning were inspired from processes and algorithms discovered in nature. To set deep learning in context visu- ally, Figure 1-1 illustrates our conception of the relationship between AI, machine learning, and deep learning.

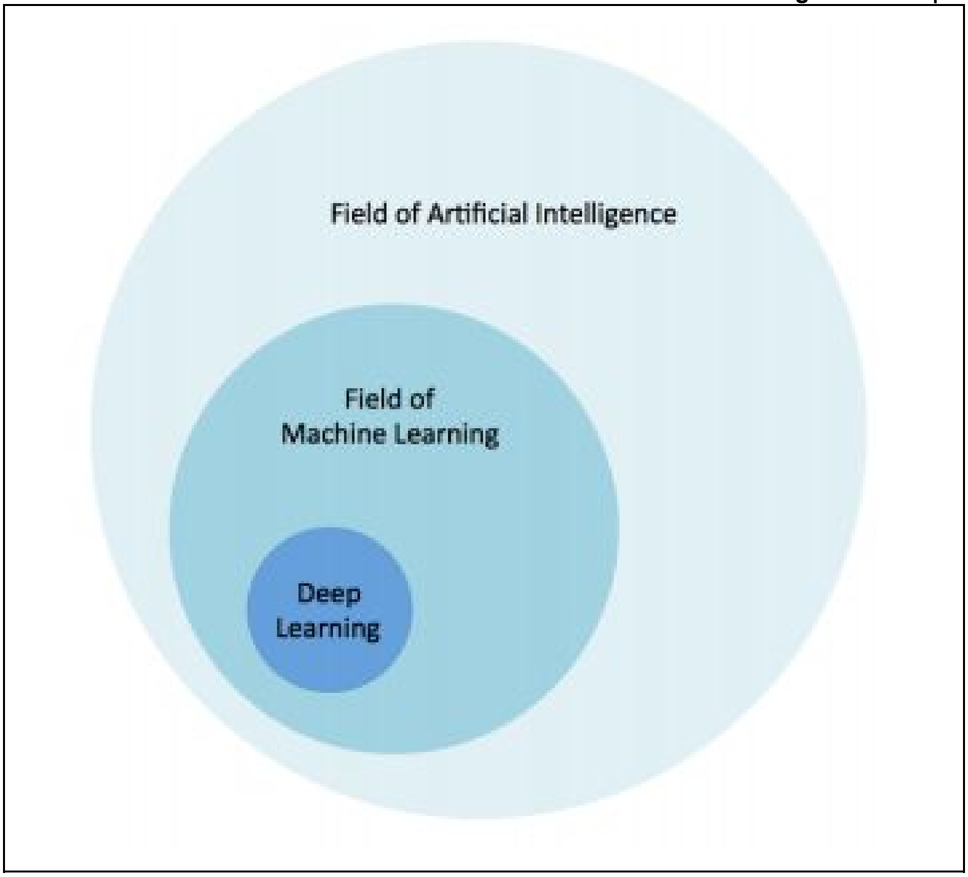


Figure 1-1. The relationship between AI and deep learning

The field of AI is broad and has been around for a long time. Deep learning is a sub- set of the field of machine learning, which is a subfield of AI. Let’s now take a quick look at another of the roots of deep learning: how neural networks are inspired by biology.

Biological Inspiration

Biological neural networks (brains) are composed of roughly 86 billion neurons con- nected to many other neurons.

Total Connections in the Human Brain

Researchers conservatively estimate there are more than 500 trillion connections between neurons in the human brain. Even the largest artificial neural networks today don't even come close to approaching this number.

From an information processing point of view a biological neuron is an excitable unit that can process and transmit information via electrical and chemical signals. A neu- ron in the biological brain is considered a main component of the brain, spinal cord of the central nervous system, and the ganglia of the peripheral nervous system. As we'll see later in this chapter, artificial neural networks are far simpler in their compa- rative structure.

Comparing Biological with Articial

Biological neural networks are considerably more complex (several orders of magnitude) than the artificial neural network versions!

There are two main properties of artificial neural networks that follow the general idea of how the brain works. First is that the most basic unit of the neural network is the *artificial neuron* (or *node* in shorthand). Artificial neurons are modeled on the biological neurons of the brain, and like biological neurons, they are stimulated by inputs. These artificial neurons pass on some—but not all—information they receive to other artificial neurons, often with transformations. As we progress through this chapter, we'll go into detail about what these transformations are in the context of neural networks.

Second, much as the neurons in the brain can be trained to pass forward only signals that are useful in achieving the larger goals of the brain, we can train the neurons of a neural network to pass along only useful signals. As we move through this chapter we'll build on these ideas and see how artificial neural networks are able to model their biological counterparts through bits and functions.

Biological Inspiration Across Computer Science

Biological inspiration is not limited to artificial neural networks in computer science. Over the past 50 years, academic research has explored other topics in nature for computational inspiration, such as the following:

- Ants
- Termites[1]
- Bees

• Genetic algorithms

Ant colonies, for instance, have been by researchers to be a powerful decentralized computer in which no single ant is a central point of failure. Ants constantly switch

1 Patterson. 2008. "TinyTermite: A Secure Routing Algorithm" and Sartipi and Patterson. 2009. "TinyTermite: A Secure Routing Algorithm on Intel Mote 2 Sensor Network Platform."

tasks to find near optimal solutions for load balancing through meta-heuristics such as quantitative stigmergy. Ant colonies are able to perform midden tasks, defense, nest construction, and forage for food while maintaining a near-optimal number of workers on each task based on the relative need with no individual ant directly coor- dinating the work.

What Is Deep Learning?

Deep learning has been a challenge to define for many because it has changed forms slowly over the past decade. One useful definition specifies that deep learning deals with a "neural network with more than two layers." The problematic aspect to this definition is that it makes deep learning sound as if it has been around since the 1980s. We feel that neural networks had to transcend architecturally from the earlier network styles (in conjunction with a lot more processing power) before showing the spectacular results seen in more recent years. Following are some of the facets in this evolution of neural networks:

• More neurons than previous networks
• More complex ways of connecting layers/neurons in NNs
• Explosion in the amount of computing power available to
train • Automatic feature extraction

For the purposes of this book, we'll define deep learning as neural networks with a large number of parameters and layers in one of four fundamental network architec- tures:

• Unsupervised pretrained networks
• Convolutional neural networks
• Recurrent neural networks
• Recursive neural networks

There are some variations of the aforementioned architectures—a hybrid convolu- tional and recurrent neural network, for example–as well. For the purpose of this book, we'll consider the four listed architectures as our focus.

Automatic feature extraction is another one of the great advantages that deep learning has over traditional machine learning algorithms. By feature extraction, we mean that the network’s process of deciding which characteristics of a dataset can be used as indicators to label that data reliably. Historically, machine learning practitioners have spent months, years, and sometimes decades of their lives manually creating exhaus- tive feature sets for the classification of data. At the time of deep learning’s Big Bang beginning in 2006, state-of-the-art machine learning algorithms had absorbed deca- des of human effort as they accumulated relevant features by which to classify input. Deep learning has surpassed those conventional algorithms in accuracy for almost

every data type with minimal tuning and human effort. These deep networks can help data science teams save their blood, sweat, and tears for more meaningful tasks.

Going Down the Rabbit Hole

Deep learning has penetrated the computer science consciousness beyond most tech- niques in recent history. This is in part due to how it has shown not only top-flight accuracy in machine learning modeling, but also demonstrated generative mechanics that fascinate even the noncomputer scientist. One example of this would be the art generation demonstrations for which a deep network was trained on a particular famous painter’s works, and the network was able to render other photographs in the painter’s unique style, as demonstrated in Figure 1-2.

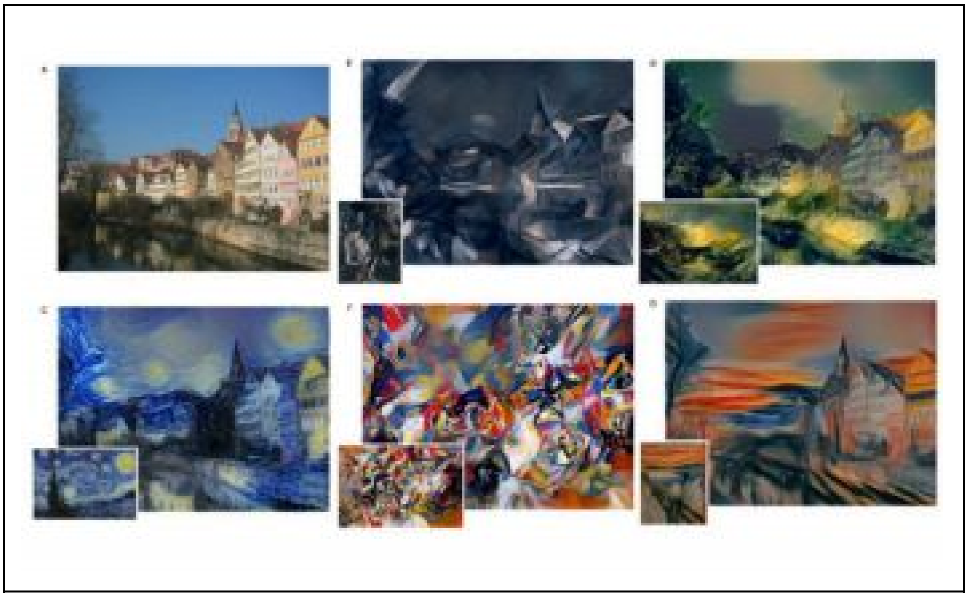


Figure 1-2. Stylized images by Gatys et al., 2015[2]

This begins to enter into many philosophical discussions, such as, “can machines be creative?” and then “what is creativity?” We’ll leave those questions for you to ponder at a later time. Machine learning has evolved over the years, like the seasons change: subtle but steady until you wake up one day and a machine has become a champion on *Jeopardy* or beat a Go Grand Master.

Can machines be intelligent and take on human-level intelligence? What is AI and how powerful could it become? These questions have yet to be answered and will not

2 Gatys et. al, 2015. “A Neural Algorithm of Artistic Style.”

be completely answered in this book. We simply seek to illustrate some of the shards of machine intelligence with which we can imbue our environment today through the practice of deep learning.

For an Extended Discussion on AI

If you would like to read more about AI, take a look at Appendix A.

Framing the Questions

The basics of applying machine learning are best understood by asking the correct questions to begin with. Here's what we need to define:

• What is the input data from which we want to extract information (model)? • What kind of model is most appropriate for this data?
• What kind of answer would we like to elicit from new data based on this model?

If we can answer these three questions, we can set up a machine learning workflow that will build our model and produce our desired answers. To better support this workflow, let's review some of the core concepts we need to be aware of to practice machine learning. Later, we'll come back to how these come together in machine learning and then use that information to better inform our understanding of both neural networks and deep learning.

The Math Behind Machine Learning: Linear Algebra

Linear algebra is the bedrock of machine learning and deep learning. Linear algebra provides us with the mathematical underpinnings to solve the equations we use to build models.

A great primer on linear algebra is James E. Gentle's *Matrix Algebra: Theory, Computations, and Applications in Statistics*.

Let's take a look at some core concepts from this field before we move on starting with the basic concept called a *scalar*.

Scalars

In mathematics, when the term scalar is mentioned, we are concerned with elements in a vector. A scalar is a real number and an element of a field used to define a vector space.

In computing, the term scalar is synonymous with the term variable and is a storage location paired with a symbolic name. This storage location holds an unknown quan- tity of information called a *value*.

Vectors

For our use, we define a vector as follows:

For a positive integer n*, a vector is an* n*-tuple, ordered (multi)set or array of* n *numbers, called elements or scalars.*

What we're saying is that we want to create a data structure called a vector via a pro- cess called *vectorization*. The number of elements in the vector is called the "order" (or "length") of the vector. Vectors also can represent points in *n*-dimensional space. In the spatial sense, the Euclidean distance from the origin to the point represented by the vector gives us the "length" of the vector.

In mathematical texts, we often see vectors written as follows:

x_1

x_2

x_3

. . .

x_n

Or:

$x =$

$x = x_1, x_2, x_3, \ldots, x_n$

There are many different ways to handle the vectorization, and you can apply many preprocessing steps, giving us different grades of effectiveness on the output models. We cover more on the topic of converting raw data into vectors later in this chapter and then more fully in Chapter 5.

Matrices

Consider a matrix to be a group of vectors that all have the same dimension (number of columns). In this way a matrix is a two-dimensional array for which we have rows and columns.

If our matrix is said to be an *n* × *m* matrix, it has *n* rows and *m* columns. Figure 1-3 shows a 3 × 3 matrix illustrating the dimensions of a matrix. Matrices are a core structure in linear algebra and machine learning, as we'll show as we progress through this chapter.

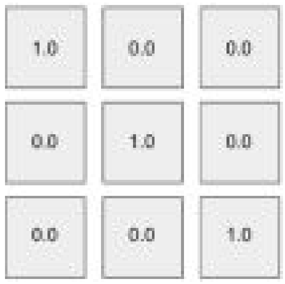


Figure 1-3. A 3 x 3 matrix

Tensors

A *tensor* is a multidimensional array at the most fundamental level. It is a more gen- eral mathematical structure than a vector. We can look at a vector as simply a subclass of tensors.

With tensors, the rows extend along the y-axis and the columns along the x-axis. Each axis is a dimension, and tensors have additional dimensions. Tensors also have a rank. Comparatively, a scalar is of rank 0 and a vector is rank 1. We also see that a matrix is rank 2. Any entity of rank 3 and above is considered a tensor.

Hyperplanes

Another linear algebra object you should be aware of is the *hyperplane*. In the field of geometry, the hyperplane is a subspace of one dimension less than its ambient space. In a three-dimensional space, the hyperplanes would have two dimensions. In two dimensional space we consider a one-dimensional line to be a hyperplane.

A hyperplane is a mathematical construct that divides an *n*-dimensional space into separate "parts" and therefore is useful in applications like classification. Optimizing the parameters of the hyperplane is a core concept in linear modeling, as you'll see further on in this chapter.

Relevant Mathematical Operations

In this section, we briefly review common linear algebra operations you should know.

Dot product

A core linear algebra operation we see often in machine learning is the *dot product*. The dot product is sometimes called the “scalar product” or “inner product." The dot product takes two vectors of the same length and returns a single number. This is done by matching up the entries in the two vectors, multiplying them, and then sum- ming up the products thus obtained. Without getting too mathematical (immedi- ately), it is important to mention that this single number encodes a lot of information.

To begin with, the dot product is a measure of how big the individual elements are in each vector. Two vectors with rather large values can give rather large results, and two vectors with rather small values can give rather small values. When the relative values of these vectors are accounted for mathematically with something called *normaliza- tion*, the dot product is a measure of how similar these vectors are. This mathematical notion of a dot product of two normalized vectors is called the *cosine similarity*.

Element-wise product

Another common linear algebra operation we see in practice is the *element-wise prod- uct* (or the “Hadamard product”). This operation takes two vectors of the same length and produces a vector of the same length with each corresponding element multi- plied together from the two source vectors.

Outer product

This is known as the “tensor product” of two input vectors. We take each element of a column vector and multiply it by all of the elements in a row vector creating a new row in the resultant matrix.

Converting Data Into Vectors

In the course of working in machine learning and data science we need to analyze all types of data. A key requirement is being able to take each data type and represent it as a vector. In machine learning we use many types of data (e.g., text, time-series, audio, images, and video).

So, why can’t we just feed raw data to our learning algorithm and let it handle every- thing? The issue is that machine learning is based on linear algebra and solving sets of equations. These equations expect floating-point numbers as input so we need a way to translate the raw data into sets of floating-point numbers. We'll connect these con- cepts together in the next section on solving these sets of equations. An example of raw data would be the canonical iris dataset:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

Another example might be a raw text document:

```
Go, Dogs. Go!
Go on skates
or go by bike.
```

Both cases involve raw data of different types, yet both need some level of vectoriza- tion to be of the form we need to do machine learning. At some point, we want our input data to be in the form of a matrix but we can convert the data to intermediate representations (e.g., "svmlight" file format, shown in the code example that fol- lows). We want our machine learning algorithm's input data to look more like the serialized sparse vector format svmlight, as shown in the following example:

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468
4:0.  5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0
.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746 1.0
1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0
.5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107
4:0  .5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149
4:0  .5217391304347826
0.0 2:0.7500000000000002
```

This format can quickly be read into a matrix and a column vector for the labels (the first number in each row in the preceding example). The rest of the indexed numbers in the row are inserted into the proper slot in the matrix as “features” at runtime to get ready for various linear algebra operations during the machine learning process. We’ll discuss the process of vectorization in more detail in Chapter 8.

Here’s a very common question: “why do machine learning algorithms want the data represented (typically) as a (sparse) matrix?” To understand that, let’s make a quick detour into the basics of solving systems of equations.

Solving Systems of Equations

In the world of linear algebra, we are interested in solving systems of linear equations of the form:

$$Ax = b$$

where *A* is a matrix of our set of input row vectors and *b* is the column vector of labels for each vector in the *A* matrix. If we take the first three rows of serialized sparse output from the previous example and place the values in its linear algebra form, it looks like this:

Column 1	Column 2	Column 3	Column 4
0.7500000000000001	0.41666666666666663	0.702127659574468	
0.5652173913043479	0.6666666666666666		0.5
0.9148936170212765		0.6956521739130436	
0.45833333333333326		0.3333333333333336	
0.8085106382978723	0.7391304347826088		

This matrix of numbers is our *A* variable in our equation, and each independent value or value in each row is considered a feature of our input data.

What Is a Feature?

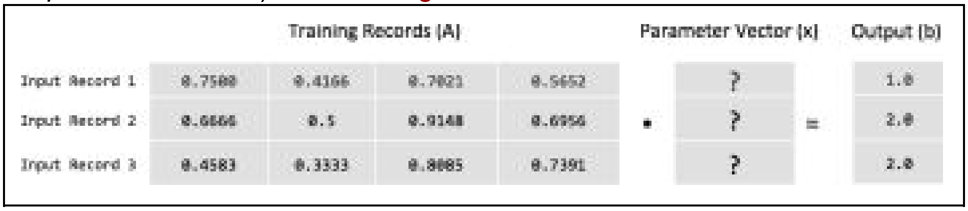
A feature in machine learning is any column value in the input matrix *A* that we're using as an independent variable. Features can be taken straight from the source data, but most of the time we're going to use some sort of transformation to get the raw input data into a form that is more appropriate for modeling.

An example would be a column of input that has four different text labels in the source data. We'd need to scan all of the input data and index the labels being used. We'd then need to normalize these values (0, 1, 2, 3) between 0.0 and 1.0 based on each label's index for every row's column value. These types of transforms greatly help machine learning find better solutions to modeling problems. We'll see more techni- ques for vectorization transforms in Chapter 5.

We want to find coefficients for each column in a given row for a predictor function that give us the output b, or the label for each row. The labels from the serialized sparse vectors we looked at earlier would be as follows:

Labels
1.0
2.0
2.0

The coefficients mentioned earlier become the x column vector (also called the *parameter vector*) shown in Figure 1-4.



	Training Records (A)					Parameter Vector (x)		Output (b)
Input Record 1	0.7500	0.4166	0.7021	0.5652		?		1.0
Input Record 2	0.6666	0.5	0.9148	0.6956	•	?	=	2.0
Input Record 3	0.4583	0.3333	0.8085	0.7391		?		2.0

Figure 1-4. Visualizing the equation Ax = b

This system is said to be "consistent" if there exists a parameter vector *x* such that the solution to this equation can be directly written as follows:

$$x = A^{-1}b$$

It’s important to delineate the expression $x = A^{-1}b$ from the method of actually com- puting the solution. This expression only represents the solution itself. The variable A^{-1} is the matrix A inverted and is computed through a process called *matrix inver- sion*. Given that not all matrices can be inverted, we’d like a method to solve this equa- tion that does not involve matrix inversion. One method is called *matrix decomposition*. An example of matrix decomposition in solving systems of linear equations is using lower upper (LU) decomposition to solve for the matrix A. Beyond matrix decomposition, let’s take a look at the general methods for solving sets of lin- ear equations.

Methods for solving systems of linear equations

There are two general methods for solving a system of linear equations. The first is called the “direct method,” in which we know algorithmically that there are a fixed number of computations. The other approach is a class of methods known as *iterative methods*, in which through a series of approximations and a set of termination condi- tions we can derive the parameter vector x. The direct class of methods is particularly effective when we can fit all of the training data (A and b) in memory on a single computer. Well-known examples of the direct method of solving sets of linear equa- tions are *Gaussian Elimination* and the *Normal Equations*.

Iterative methods

The iterative class of methods is particularly effective when our data doesn’t fit into the main memory on a single computer, and looping through individual records from disk allows us to model a much larger amount of data. The canonical example of iterative methods most commonly seen in machine learning today is *Stochastic Gradient Descent* (SDG), which we discuss later in this chapter. Other techniques in this space are *Conjugate Gradient Methods* and *Alternating Least Squares* (discussed further in Chapter 3). Iterative methods also have been shown to be effective in scale out methods, for which we not only loop through local records, but the entire dataset is sharded across a cluster of machines, and periodically the parameter vector is aver- aged across all agents and then updated at each local modeling agent (described in more detail in Chapter 9).

Iterative methods and linear algebra

At the mathematical level, we want to be able to operate on our input dataset with these algorithms. This constraint requires us to convert our raw input data into the input matrix A. This quick overview of linear algebra gives us the "why" for going through the trouble to vectorize data. Throughout this book, we show code examples of converting the raw input data into the input matrix *A*, giving you the "how." The mechanics of how we vectorize our data also affects the results of the learning pro- cess. As we'll see later in the book, how we handle data in the preprocess stage before vectorization can create more accurate models.

The Math Behind Machine Learning: Statistics

Let's review just enough statistics to let this chapter move forward. We need to high- light some basic concepts in statistics, such as the following:

- Probabilities
- Distributions
- Likelihood

There are also some other basic relationships we'd like to highlight in descriptive sta- tistics and inferential statistics. Descriptive statistics include the following:

- Histograms
- Boxplots
- Scatterplots
- Mean
- Standard deviation
- Correlation coefficient

This contrasts with how inferential statistics are concerned with techniques for gener- alizing from a sample to a population. Here are some examples of inferential statis- tics:

- p-values
- credibility intervals

The relationship between probability and inferential statistics:

- Probability reasons from the population to the sample (deductive reasoning) • Inferential statistics reason from the sample to the population

Before we can understand what a specific sample tells us about the source population, we need to understand the uncertainty associated with taking a sample from a given population.

Regarding general statistics, we won't linger on what is an inherently broad topic already covered in depth by other books. This section is in no way meant to serve as a true statistics review; rather, it is designed to direct you toward relevant topics that you can investigate in greater depth from other resources. With that disclaimer out of the way, let's begin by defining probability in statistics.

Probability

We define probability of an event *E* as a number always between 0 and 1. In this con- text, the value 0 infers that the event *E* has no chance of occurring, and the value 1 means that the event *E* is certain to occur. Many times we'll see this probability expressed as a floating-point number, but we also can express it as a percentage between 0 and 100 percent; we will not see valid probabilities lower than 0 percent and greater than 100 percent. An example would be a probability of 0.35 expressed as 35 percent (e.g., 0.35 x 100 == 35 percent).

The canonical example of measuring probability is observing how many times a fair coin flipped comes up heads or tails (e.g., 0.5 for each side). The probability of the sample space is always 1 because the sample space represents all possible outcomes for a given trial. As we can see with the two outcomes ("heads" and its complement, "tails") for the flipped coin, 0.5 + 0.5 == 1.0 because the total probability of the sam- ple space must always add up to 1. We express the probability of an event as follows:

$P(E) = 0.5$

And we read this like so:

The probability of an event *E* is 0.5

Probability Versus Odds, Explained

Many times practitioners new to statistics or machine learning will conflate the mean- ing of probability and odds. Before we proceed, let's clarify this here.

The probability of an event E is defined as:

$P(E)$ = (Chances for E) / (Total Chances)

We see this in the example of drawing an ace card (4) out of a deck of cards (52) where we'd have this:

4/52 = 0.077

Conversely, odds are defined as:

(Chances for E) : (Chances Against E)

Now our card example becomes the "odds of drawing an ace":

4 : (52 – 4) = 1/12 = 0.0833333...

The primary difference here is the choice of denominator (Total Chances versus Chances Against) making these two distinct concepts in statistics.

Probability is at the center of neural networks and deep learning because of its role in feature extraction and classification, two of the main functions of deep neural net- works. For a larger review of statistics, check out O'Reilly's *Statistics in a Nutshell: A Desktop Quick Reference* by Boslaugh and Watters.

Further Dening Probability: Bayesian Versus Frequentist

There are two different approaches in statistics called *Bayesianism* and *frequentism*. The basic difference between the approaches is how probability is defined.

With frequentists, probability only has meaning in the context of repeating a meas- urement. As we measure something, we'll see slight variations due to variances in the equipment we use to collect data. As we measure something a large number of times, the frequency of the given value indicates the probability of measuring that value.

With the Bayesian approach, we extend the idea of probability to cover aspects of cer- tainty about statements. The probability gives us a statement of our knowledge of what the measurement result will be. For Bayesians, our own knowledge about an event is fundamentally related to probability.

Frequentists rely on many, many blind trials of an experiment before making state- ments about an estimate for a variable. Bayesians, on the other hand, deal in “beliefs” (in mathematical terms, “distributions”) about the variable and update their beliefs about the variable as new information comes in.

Conditional Probabilities

When we want to know the probability of a given event based on the existing pres- ence of another event occurring, we express this as a *conditional probability*. This is expressed in literature in the form:

$$P(E \mid F)$$

where:

E is the event for which we’re interested in a probability.

F is the event that has already occurred.

An example would be expressing how a person with a healthy heart rate has a lower probability of ICU death during a hospital visit:

$P(\text{ICU Death} \mid \text{Poor Heart Rate}) > P(\text{ICU Death} \mid \text{Healthy Heart Rate})$

Sometimes, we’ll hear the second event, *F*, referred to as the “condition.” Conditional probability is interesting in machine learning and deep learning because we’re often interested in when multiple things are happening and how they interact. We’re inter- ested in conditional probabilities in machine learning in the context in which we’d learn a classifier by learning

$$P (E \mid F)$$

where *E* is our label and *F* is a number of attributes about the entity for which we’re predicting *E*. An example would be predicting mortality (here, *E*) given that measure- ments taken in the ICU for each patient (here, *F*).

Bayes’s Theorem

One of the more common applications of conditional probabilities is Bayes’s theorem (or Bayes’s formula). In the field of medicine, we see it used to calculate the probabil- ity that a patient who tests positive on a test for a specific disease actually has the dis- ease.

We define Bayes’s formula for any two events, *A* and *B*, as:

$$P\ A\ B = \frac{P\ B\ A\ P\ A}{P\ B}$$

Posterior Probability

In Bayesian statistics, we call the posterior probability of the random event the condi- tional probability we assign after the evidence is considered. Posterior probability dis- tribution is defined as the probability distribution of an unknown quantity conditional on the evidence collected from an experiment treated as a random vari- able. We see this concept in action with softmax activation functions (explained later in this chapter), in which raw input values are converted to posterior probabilities.

Distributions

A probability distribution is a specification of the stochastic structure of random vari- ables. In statistics, we rely on making assumptions about how the data is distributed to make inferences about the data. We want a formula that specifies how frequent val- ues of observations in the distribution are and how values can be taken by points in the distribution. A common distribution is known as the *normal distribution* (also called *Gaussian distribution*, or the *bell curve*). We like to fit a dataset to a distribution because if the dataset is reasonably close to the distribution, we can make assump- tions based on the theoretical distribution in how we operate with the data.

We classify distributions as *continuous* or *discrete*. A discrete distribution has data that can assume only certain values. In a continuous distribution, data can be any value within the range. An example of a continuous distribution would be normal distribution. An example of a discrete distribution would be binomial distribution.

Normal distribution allows us to assume sampling distributions of statistics (e.g., "sample mean") are normally distributed under specified conditions. The normal dis- tribution (see Figure 1-5), or *Gaussian distribution*, was named after the eighteenth century mathematician and physicist Karl Gauss. Normal distribution is defined by its mean and standard deviation and has generally the same shape across all varia- tions.

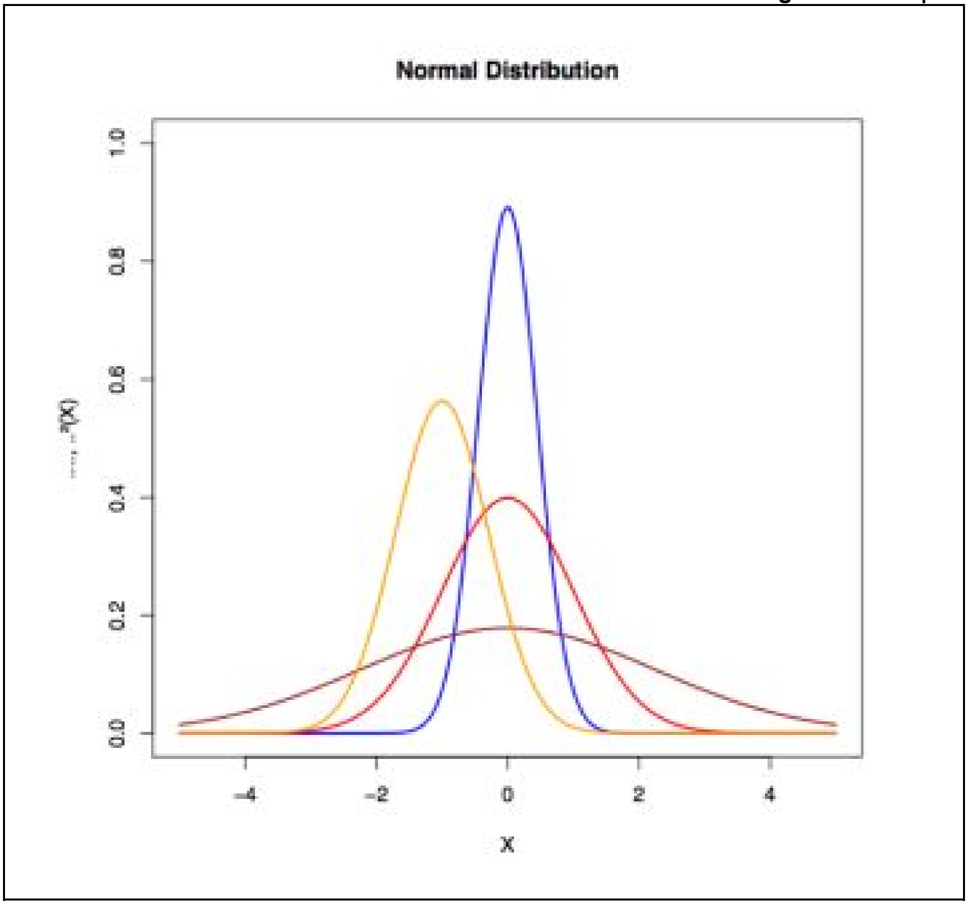


Figure 1-5. Examples of normal distributions

Other relevant distributions in machine learning include the following:

- Binomial distribution
- Inverse Gaussian distribution
- Log normal distribution

Distribution of the training data in machine learning is important to understand how to vectorize the data for modeling.

Central Limit Theorem

If the sample size is sufficiently large, the sampling distribution of the sample mean approximates the normal distribution. This holds true despite the distribution of the population from which the samples were collected.

Based on this fact, we can make statistical inferences using tests based on the approxi- mate normality of the mean. We see this hold true regardless of whether the sample is drawn from a population that is not normally distributed.

In computer science, we see this used where an algorithm repeatedly draws samples of specified size from a nonnormal population. When we graph the histogram of the sample population of the draws from a normal distribution, we can see this effect in action.

A long-tailed distribution (such as Zipf, power laws, and Pareto distributions) is a scenario in which a high-frequency population is followed by a low-frequency popu- lation that gradually decreases in asymptotic fashion. These distributions were dis- covered by Benoit Mandelbrot in the 1950s and later popularized by the writer Chris Anderson in his book *The Long Tail: Why the Future of Business is Selling Less of More*.

An example would be ranking the items a retailer sells among which a few items are exceptionally popular and then we see a large number of unique items with relatively small quantities sold. This rank-frequency distribution (primarily of popularity or “how many were sold”) often forms power laws. From this perspective, we can con- sider them to be long-tailed distributions.

We see these long-tailed distributions manifested in the following:

Earthquake damage
: The damage becomes worse as the scale of the quake increases, so worse case shifts.

Crop yields
: We sometimes see events outside of the historical record, whereas our model tends to be tuned around the mean.

Predicting fatality post ICU visit
: We can have events far outside the scope of what happens inside the ICU visit that affect mortality.

These examples are relevant in the context of this book for classification problems because most statistical models depend on inference from lots of data. If the more interesting events occur out in the tail of the distribution and we don’t have this rep- resented in the training sample data, our model might perform unpredictably. This effect can be enhanced in nonlinear models such as neural networks. We’d consider

this situation the special case of the “in sample/out of sample” problem. Even a seas- oned machine learning practitioner can be surprised at how well a model performs on a skewed training data sample yet fails to generalize well on the larger population of data.

Long-tailed distributions deal with the real possibility of events occurring that are five times the standard deviation. We must be mindful to get a decent representation of events in our training data to prevent overfitting the training data. We’ll look in greater detail at ways to do this later when we talk about overfitting and then in Chapter 4 on tuning.

Samples Versus Population

A population of data is defined as all of the units we’d like to study or model in our experiment. An example would be defining our population of study as “all Java pro- grammers in the state of Tennessee.”

A sample of data is a subset of the population of data that hopefully represents the accurate distribution of the data without introducing sampling bias (e.g., skewing the sample distribution based on how we sampled the population).

Resampling Methods

Bootstrapping and *cross-validation* are two common methods of resampling in statis- tics that are useful to machine learning practitioners. In the context of machine learn- ing with bootstrapping, we're drawing random samples from another sample to generate a new sample that has a balance between the number of samples per class. This is useful when we'd like to model against a dataset with highly unbalanced classes.

Cross-validation (also called *rotation estimation*) is a method to estimate how well a model generalizes on a training dataset. In cross-validation we split the training data- set into *N* number of splits and then separate the splits into training and test groups. We train on the training group of splits and then test the model on the test group of splits. We rotate the splits between the two groups many times until we've exhausted all the variations. There is no hard number for the number of splits to use but researchers have found 10 splits to work well in practice. It also is common to see a separate portion of the held-out data used as a validation dataset during training.

Selection Bias

In *selection bias* we're dealing with a sampling method that does not have proper ran- domization and skews the sample in a way such that the sample is not representative of the population we'd like to model. We need to be aware of selection bias when

resampling datasets so that we don't introduce bias into our models that will lower our model's accuracy on data from the larger population.

Likelihood

When we discuss the likeliness that an event will occur yet do not specifically refer- ence its numeric probability, we're using the informal term, *likelihood*. Typically, when we use this term, we're talking about an event that has a reasonable probability of happening but still might not. There also might be factors not yet observed that will influence the event, as well. Informally, likelihood is also used as a synonym for probability.

How Does Machine Learning Work?

In a previous section on solving systems of linear equations, we introduced the basics of solving $Ax = b$. Fundamentally, machine learning is based on algorithmic techni- ques to minimize the error in this equation through *optimization*.

In optimization, we are focused on changing the numbers in the *x* column vector (parameter vector) until we find a good set of values that gives us the closest out- comes to the actual values. Each weight in the weight matrix will be adjusted after the loss function calculates the error (based on the actual outcome, as shown earlier, as the *b* column vector) produced by the network. An error matrix attributing some portion of the loss to each weight will be multiplied by the weights themselves.

We discuss SDG further on in this chapter as one of the major methods to perform machine learning optimization, and then we'll connect these concepts to other opti- mization algorithms as the book progresses. We'll also cover the basics of hyperpara- meters such as regularization and learning rate.

Regression

Regression refers to functions that attempt to predict a real value output. This type of function estimates the dependent variable by knowing the independent variable. The most common class of regression is *linear regression*, based on the concepts we've pre- viously described in modeling systems of linear equations. Linear regression attempts to come up with a function that describes the relationship between *x* and *y*, and, for known values of *x*, predicts values of *y* that turn out to be accurate.

Setting up the model

The prediction of a linear regression model is the linear combination of coefficients (from the parameter vector *x*) and then input variables (features from the input vec- tor). We can model this by using the following equation:

$$y = a + Bx$$

where *a* is the *y*-intercept, *B* is the input features, and *x* is the

parameter vector. This equation expands to the following:

$$y = a + b_0 * x_0 + b_1 * x_1 + \ldots + b_n * x_n$$

A simple example of a problem that linear regression solves would be predicting how much we'd spend per month on gasoline based on the length of our commute. Here, what you pay at the tank is a function of how far you drive. The gas cost is the depen- dent variable and the length of the commute is the independent variable. It's reason- able to keep track of these two quantities and then define a function, like so:

$$cost = f(distance)$$

This allows us to reasonably predict our gasoline spending based on mileage. In this example, we'd consider distance to be our independent variable and cost to be the dependent variable in our model *f*.

Here are some other examples of linear regression modeling:

- Predicting weight as a function of height
- Predicting a house's sale price based on its square footage

Visualizing linear regression

Visually, we can represent linear regression as finding a line that comes as close to as many points as possible in a scatterplot of data, as demonstrated in Figure 1-6.

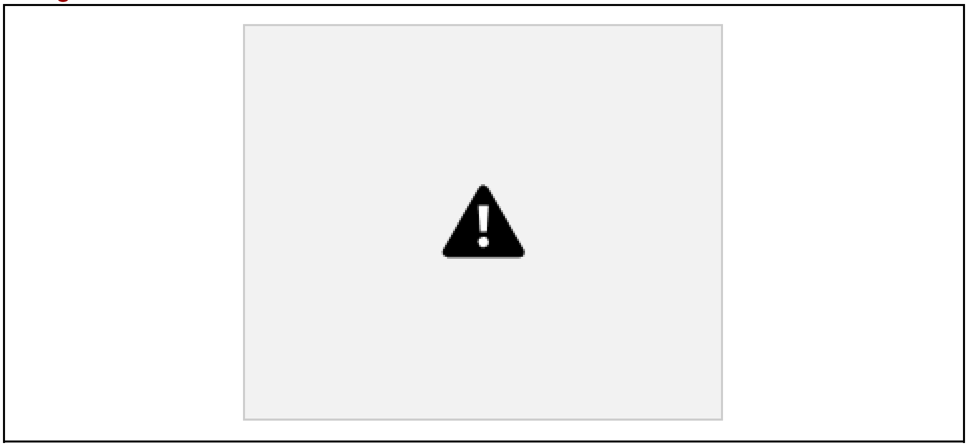


Figure 1-6. Linear regression plotted

Fitting is defining a function *f(x)* that produces *y*-values close to the measured or real world *y* values. The line produced by *y* = *f(x)* comes close to the scattered coordi- nates, the pairs of dependent and independent variables.

Relating the linear regression model

We can relate this function to the earlier equation, *Ax* = *b*, where *A* is the features (e.g., "weight" or "square footage") for all of the input examples that we want to model. Each input record is a row in the matrix *A*. The column vector *b* is the out- comes for all of the input records in the *A* matrix. Using an error function and an optimization method (e.g., SGD), we can find a set of *x* parameters such that we min- imize the error across all of the predictions versus the true outcomes.

Using SGD, as we discussed earlier, we'd have three components to solve for our parameter vector *x*:

A hypothesis about the data
: The inner product of the parameter vector *x* and the input features (as displayed above)

A cost function
: Squared error (prediction – actual) of prediction

An update function
: The derivative of the squared error loss function (cost function)

While linear regression deals with straight lines, nonlinear curve fitting handles everything else, most notably curves that deal with *x* to higher exponents than 1. (That's why we hear machine learning sometimes described as "curve fitting.") An absolute fit would hit every dot on a scatterplot. Ironically, absolute fit is usually a very poor outcome, because it means your model has trained too perfectly on the training set, and has almost no predictive power beyond the data it has seen (e.g., does not generalize well), as we previously discussed.

Classication

Classification is modeling based on delineating classes of output based on some set of input features. If regression give us an outcome of "*how much*," classification gives us an outcome of "*what kind*." The dependent variable *y* is categorical rather than numerical.

The most basic form of classification is a binary classifier that only has a single output with two labels (two classes: 0 and 1, respectively). The output can also be a floating point number between 0.0 and 1.0 to indicate a classification below absolute certain- tity. In this case, we need to determine a threshold (typically 0.5) at which we delineate between the two classes. These classes are often referred to as positive classi-

fication in the literature (e.g., 1.0) and then negative (e.g., 0.0) classifications. We'll talk more about this in "Evaluating Models" on page 36.

Examples of binary classification include:

- Classifying whether someone has a disease or not
- Classifying an email as spam or not spam
- Classifying a transaction as fraudulent or nominal

Beyond two labels, we can have classification models that have *N* labels for which we'd score each of the output labels, and then the label with the highest score is the output label. We'll discuss this further as we talk about neural networks with multiple outputs versus neural networks with a single output (binary classification). We'll also discuss classification more in this chapter when we talk about logistic regression and then dive into the full architecture of neural networks.

Recommendation

Recommendation is the process of suggesting items to users of a system based on similar other users or other items they have looked at before. One of the more famous variations of recommendation algorithms is called *Collaborative Filtering* made famous by Amazon.com.

Clustering

Clustering is an unsupervised learning technique that involves using a distance meas- ure and iteratively moving similar items more closely together. At the end of the pro- cess, the items clustered most densely around *n* centroids are considered to be classified in that group. *K*-means clustering is one of the more famous variations of clustering in machine learning.

Undertting and Overtting

As we mentioned earlier, optimization algorithms first attempt to solve the problem of underfitting; that is, of taking a line that does not approximate the data well and making it approximate the data better. A straight line cutting across a curving scatter- plot would be a good example of underfitting, as is illustrated in Figure 1-7.

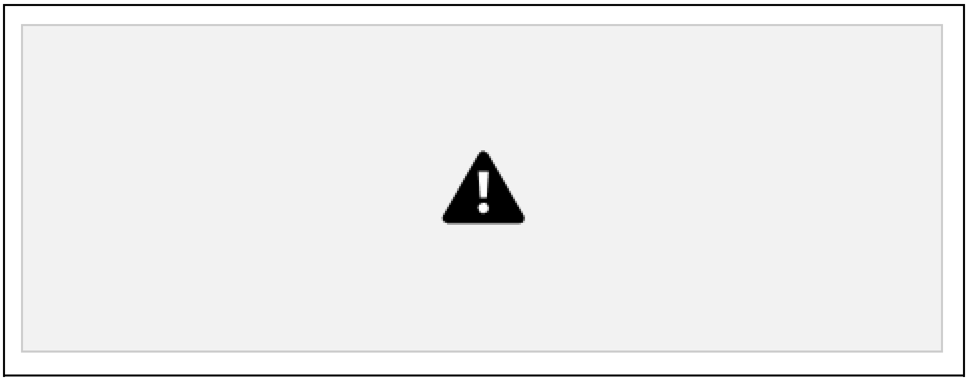


Figure 1-7. Underfitting and overfitting in machine learning

If the line fits the data too well, we have the opposite problem, called "overfitting." Solving underfitting is the priority, but much effort in machine learning is spent attempting not to overfit the line to the data. When we say a model overfits a dataset, we mean that it may have a low error rate for the training data, but it does not gener- alize well to the overall population of data in which we're interested.

Another way of explaining overfitting is by thinking about probable distributions of data. The training set of data that we're trying to draw a line through is just a sample of a larger unknown set, and the line we draw will need to fit the larger set equally well if it is to have any predictive power. We must assume, therefore, that our sample is loosely representative of a larger set.

Optimization

The aforementioned process of adjusting weights to produce more and more accurate guesses about the data is known as *parameter optimization*. You can think of this pro- cess as a scientific method. You formulate a hypothesis, test it against reality, and refine or replace that hypothesis again and again to better describe events in the world.

Every set of weights represents a specific hypothesis about what inputs mean; that is, how they relate to the meanings contained in one's labels. The weights represent con- jectures about the correlations between networks' input and the target labels they seek to guess. All possible weights and their combinations can be described as the hypothesis space of this problem. Our attempt to formulate the best hypothesis is a matter of searching through that hypothesis space, and we do so by using error and optimization algorithms. The more input parameters we have, the larger the search space of our problem. Much of the work of learning is deciding which parameters to ignore and which to hear.

The Decision Boundary and Hyperplanes

When we mention the “decision boundary,” we’re talking about the *n*-dimensional hyperplane created by the parameter vector in linear modeling.

Fitting lines to data by gauging their cost (i.e., their distance from the ground-truth data points) is at the center of machine learning. The line should more or less fit the data, and it does so by minimizing the aggregate distance of all points from the line. You minimize the sum of the difference between the line at point *x* and the target point *y* to which it corresponds. In a three-dimensional space, you can imagine the error-scape of hills and valleys, and picture your algorithm as a blind hiker who feels for the slope. An optimization algorithm, like gradient descent, is what informs the hiker which direction is downhill so that she knows where to step.

The goal is to find the weights that minimize the difference between what your net- work predicts (*b*, or the dot-product of *A* and *x*) and what your test set knows to be true (*b*), as we saw earlier in Figure 1-4. he parameter vector (*x*) above is where you would find the weights. The accuracy of a network is a function of its input and parameters, and the speed at which it becomes accurate is a function of its hyperpara- meters.

Hyperparameters

In machine learning, we have both model parameters and then we have parameters we tune to make networks train better and faster. These tuning parameters are called *hyperparameters*, and they deal with controlling optimization function and model selection during training with our learning algorithm.

Convergence

Convergence refers to an optimization algorithm finding values for a parameter vector that gives our optimization algorithm the smallest error possible across all training examples. The optimization algorithm is said to “converge” on the solution iteratively after it tries several different variations of the parameters.

Following are the three important functions at work in machine learning optimiza- tion:

Parameters

Transform input to help determine the classifications a network infers

Loss function

Gauges how well it classifies (minimizes error) at each step

Optimization function

Guides it toward the points of least error

Now, let’s take a closer look at one subclass of optimization called convex optimiza- tion.

Convex Optimization

In *convex optimization*, learning algorithms deal with convex cost functions. If the x axis represents a single weight, and the y-axis represents that cost, the cost will descend as low as 0 at one point on the x-axis and rise exponentially on either side as the weight strays away from its ideal in two directions.

Figure 1-8 demonstrates that we also can turn the idea of a cost function upside down.

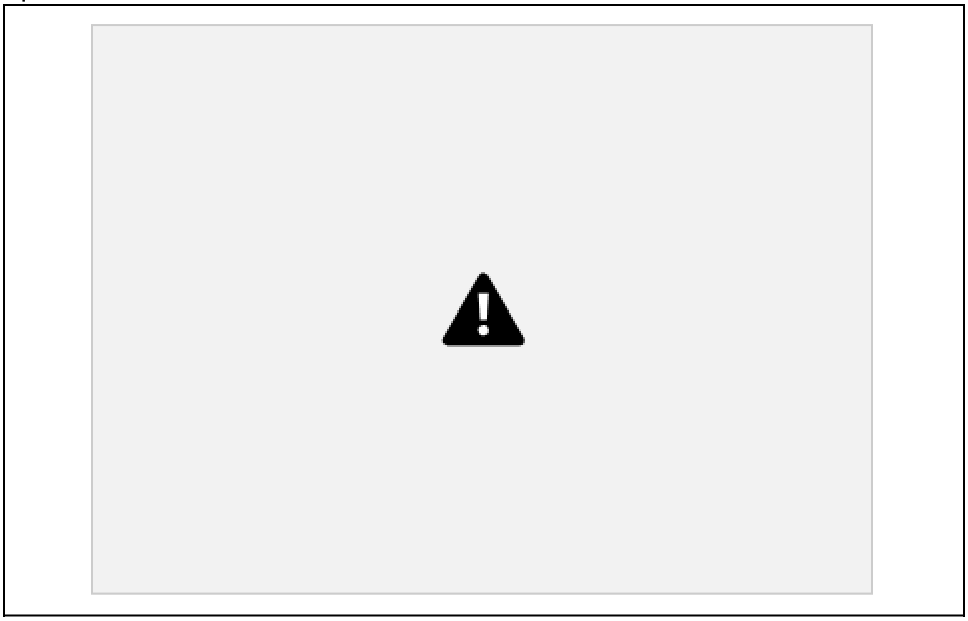


Figure 1-8. Visualizing convex functions

Another way to relate parameters to the data is with a maximum likelihood estima- tion, or MLE. The MLE traces a parabola whose edges point downward, with likeli- hood measured on the vertical axis, and a parameter on the horizontal. Each point on the parabola measures the likelihood of the data, given a certain set of parameters. The goal of MLE is to iterate over possible parameters until it finds the set that makes the given data most likely.

In a sense, maximum likelihood and minimum cost are two sides of the same coin. Calculating the cost function of two weights against the error (which puts us in a

three-dimensional space) will produce something that looks more like a sheet held at each corner and drooping convexly in the middle—a rather bowl-shaped function. The slopes of these convex curves allow our algorithm a hint as to what direction to take the next parameter step, as we'll see in the gradient descent optimization algo- rithm discussed next.

Gradient Descent

In gradient descent, we can imagine the quality of our network's predictions (as a function of the weight/parameter values) as a landscape. The hills represent locations (parameter values or weights) that give a lot of prediction error; valleys represent locations with less error. We choose one point on that landscape at which to place our initial weight. We then can select the initial weight based on domain knowledge (if we're training a network to classify a flower species we know petal length is impor- tant, but color isn't). Or, if we're letting the network do all the work, we might choose the initial weights randomly.

The purpose is to move that weight downhill, to areas of lower error, as quickly as possible. An optimization algorithm like gradient descent can sense the actual slope of the hills with regard to each weight; that is, it knows which direction is down. Gra- dient descent measures the slope (the change in error caused by a change in the weight) and takes the weight one step toward the bottom of the valley. It does so by taking a derivative of the loss function to produce the gradient. The gradient gives the algorithm the direction for the next step in the optimization algorithm, as depicted in Figure 1-9.

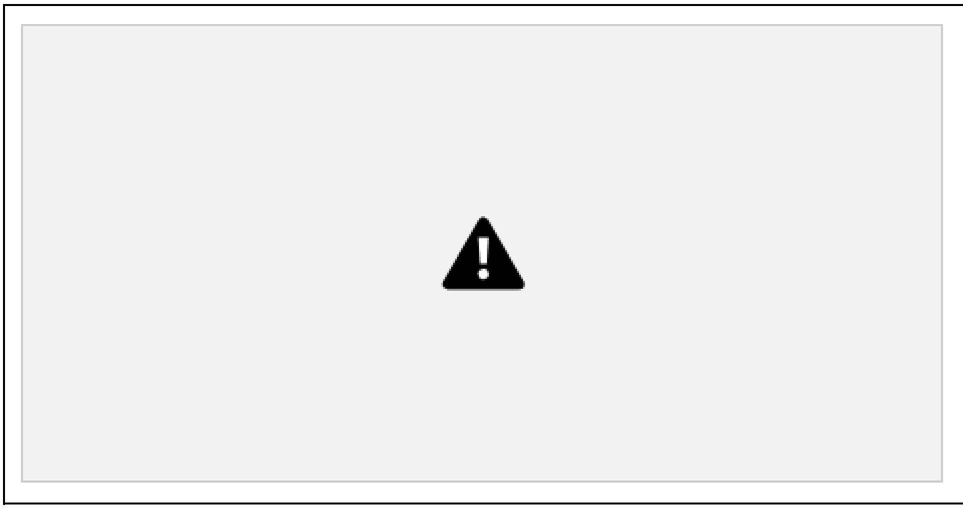


Figure 1-9. Showing weight changes toward global minimum in SGD

The derivative measures “rate of change” of a function. In convex optimization, we’re looking for the point at which the derivative is equal to 0 for the function. This point is also known as the *stationary point* of the function or the *minimum point*. In optimi- zation, we consider optimizing a function to be minimizing a function (outside of inverting the cost function).

What Is Gradient?

Gradient is defined as the generalization of the derivative of a function in one dimen- sion to a function *f* in several dimensions. It is represented as a vector of *n* partial derivatives of the function *f*. It is useful in optimization in that the gradient points in the direction of the greatest rate of increase of the function for which the magnitude is the slope of the graph in that direction.

Gradient descent calculates the slope of the loss function by taking a derivative, which should be a familiar term from calculus. On a two-dimensional loss function, the derivative would simply be the tangent of any point on the parabola; that is, the change in *y* over the change in *x*, rise over run.

As we know from trigonometry, a tangent is just a ratio: the opposite side (which measures vertical change) over the adjacent side (which measures horizontal change) of a right triangle.

One definition of a curve is a line of constantly changing slope. The slope of each point on the curve is represented by the tangent line touching that point. Because slopes are derived from two points, how exactly does one find the slope of one point on a curve? We find the derivative by calculating the slope of a line between two points on the curve separated by a small distance and then slowing decreasing that distance until it approaches zero. In calculus, this is a *limit*.

This process of measuring loss and changing the weight by one step in the direction of less error is repeated until the weight arrives at a point beyond which it cannot go lower. It stops in the trough, the point of greatest accuracy. When using a convex loss function (typically in linear modeling), we see a loss function that has only a global minimum.

You can think of linear modeling in terms of three components to solve for our parameter vector *x*:

• A hypothesis about the data; for example, “the equation we use to make a predic- tion in the model.”
• A cost function. Also called a loss function; for example, “sum of squared errors.” • An update function; we take the derivative of the loss function.

Our hypothesis is the combination of the learned parameters *x* and the input values (features) that gives us a classification or real valued (regression) output. The cost function tells us how far we are from the global minimum of the loss function, and we use the derivative of the loss function as the update function to change the param- eter vector *x*.

Taking the derivative of the loss function indicates for each parameter in *x* the degree to which we need to adjust the parameter to get closer to the 0-point on the loss curves. We'll look more closely at these equations later on in this chapter when we show how they work for both linear regression and logistic regression (classification).

However, in other nonlinear problems we don't always get such a clean loss curve. The problem with these other nonlinear hypothetical landscapes is that there might be several valleys, and gradient descent's mechanism for taking the weight lower cannot know if it has reached the lowest valley or simply the lowest point in a higher valley, so to speak. The lowest point in the lowest valley is known as *the global minimum*, whereas the nadirs of all other valleys are known as *local minima*. If gradi- ent descent reaches a local minimum, it is effectively trapped, and this is one draw- back of the algorithm. We'll look at ways to overcome this issue in Chapter 6 when we examine hyperparameters and learning rate.

A second problem that gradient descent encounters is with non-normalized features. When we write "non-normalized features," we mean features that can be measured by very different scales. If you have one dimension measured in the millions, and another in decimals, gradient descent will have a difficult time finding the steepest slope to minimize error.

Dealing with Normalization

In Chapter 8 we take an extended look at methods of normalization in the context of vectorization and illustrate some ways to better deal with this issue.

Stochastic Gradient Descent

In gradient descent we'd calculate the overall loss across all of the training examples before calculating the gradient and updating the parameter vector. In SGD, we com- pute the gradient and parameter vector update after every training sample. This has been shown to speed up learning and also parallelizes well as we'll talk about more later in the book. SGD is an approximation of "full batch" gradient descent.

Mini-batch training and SGD

Another variant of SGD is to use more than a single training example to compute the gradient but less than the full training dataset. This variant is referred to as the *mini* *batch* size of training with SGD and has been shown to be more performant than using only single training instances. Applying mini-batch to stochastic gradient descent has also shown to lead to smoother convergence because the gradient is com- puted at each step it uses more training examples to compute the gradient.

As the mini-batch size increases the gradient computed is closer to the "true" gradient of the entire training set. This also gives us the advantage of better computational effi- ciency. If our mini-batch size is too small (e.g., 1 training record), we're not using hardware as effectively as we could, especially for situations such as GPUs. Con- versely, making the mini-batch size larger (beyond a point) can be inefficient, as well, because we can produce the same gradient with less computational effort (in some cases) with regular gradient descent.

Quasi-Newton Optimization Methods

Quasi-Newton optimization methods are iterative algorithms that involve a series of "line searches." Their distinguishing feature with respect to other optimization meth- ods is how they choose the search direction. These methods are discussed further in later chapters of the book.

The Jacobian and the Hessian

The Jacobian is an $m \times n$ matrix containing the first-order partial derivatives of vec- tors with respect to vectors.

The Hessian is the square matrix of second-order partial derivatives of a function. This matrix describes the local curvature of a function of many variables. We see the Hessian matrix used in large-scale optimization problems using Newton-type meth- ods because they are the coefficients of the quadratic term of a local Taylor expansion. In practice, the Hessian can be computationally difficult to compute. We tend to see quasi-Newton algorithms used instead that approximate the Hessian. An example of this class of quasi-Newton optimization algorithm would be L-BFGS, which we cover in greater detail in Chapter 2.

We won't reference the Jacobian and the Hessian matricies much in this book, but we want the reader to be aware of them and their place in the wider scope of the machine learning landscape.

Generative Versus Discriminative Models

We can generate different types of output from a model depending on what type of model we set up. The two major types are *generative* models and *discriminative* mod- els. Generative models understand how the data was created in order to generate a type of response or output. Discriminative models are not concerned with how the data was created and simply give us a classification or category for a given input sig- nal. Discriminative models focus on closely modeling the boundary between classes and can yield a more nuanced representation of this boundary than a generative model. Discriminative models are typically used for classification in machine learn- ing.

A generative model learns the *joint* probability distribution $p(x, y)$, whereas a dis- criminative model learns the *conditional* probability distribution $p(y|x)$. The distribu- tion $p(y|x)$ is the natural distribution for taking input x and producing an output (or classification) y, hence the name "discriminative model." With generative models learning the distribution $p(x,y)$, we see them used to generate likely output, given a certain input. Generative models are typically set up as probabilistic graphical models which capture the subtle relations in the data.

Logistic Regression

Logistic regression is a well-known type of classification in linear modeling. It works for both binary classification as well as for multiple labels in the form of multinomial logistic regression. Logistic regression is a regression model (technically) in which the dependent variable is categorical (e.g., "classification"). The binary logistic model is used to estimate the probability of a binary response based on a set of one or more input variables (independent variables or "features"). This output is the statistical probability of a category, given certain input predictors.

Similar to linear regression, we can express a logistic regression modeling problem in the form of $Ax = b$, where A is the feature (e.g., "weight" or "square footage") for all of the input examples we want to model. Each input record is a row in the matrix A, and the column vector b is the outcomes for all of the input records in the A matrix. Using a cost function and an optimization method, we can find a set of x parameters such that we minimize the error across all of the predictions versus the true outcomes.

Again, we'll use SGD to set up this optimization problem and we have three compo- nents to solve for our parameter vector x:

A hypothesis about the data

$$f\,x = \frac{1}{1 + e^{-\theta x}}$$

A cost function

"max likelihood estimation"

An update function

A derivative of the cost function

In this case, the input comprises independent variables (e.g., the input columns or “features”), whereas the output is the dependent variables (e.g., “label scores”). An

easy way to think about it is logistic regression function pairs input values with weights to determine whether an outcome is likely. Let’s take a closer look at the logis- tic function.

The Logistic Function

In logistic regression we define the *logistic function* (“hypothesis”) as follows:

$$f x = \frac{1}{1 + e^{-\theta x}}$$

This function is useful in logistic regression because it takes any input in the range of negative to positive infinity and maps it to output in the range of 0.0 to 1.0. This allows us to interpret the output value as a probability. Figure 1-10 shows a plot of the logistic function equation.

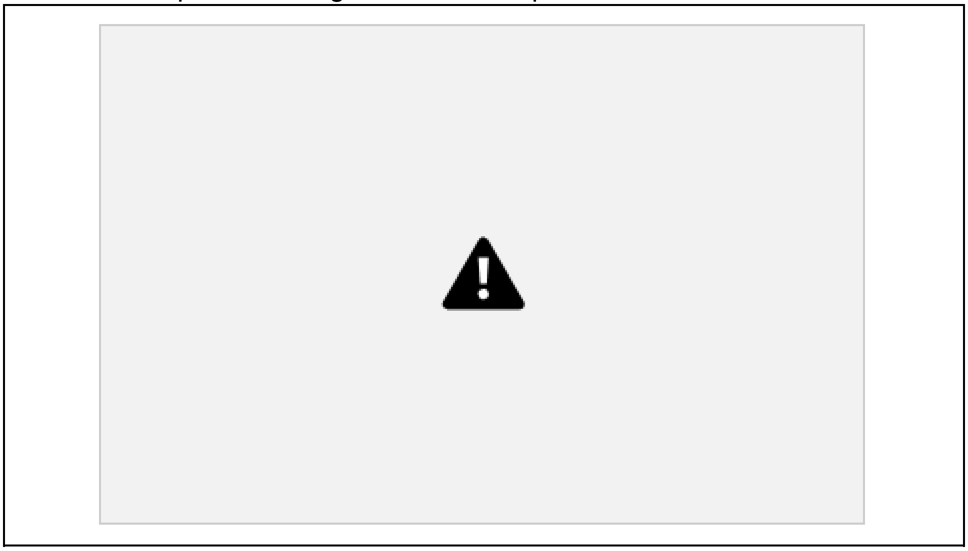


Figure 1-10. Plot of the logistic function

This function is known as a *continuous log-sigmoid function* with a range of 0.0 to 1.0. We’ll see this function covered again later in “Activation Functions” on page 65.

Understanding Logistic Regression Output

The logistic function is often denoted with the Greek letter sigma, or σ, because the relationship between *x* and *y* on a two-dimensional graph resembles an elongated, wind-blown "s" whose maximum and minimum asymptotically approach 1 and 0, respectively.

If *y* is a function of *x*, and that function is sigmoidal or logistic, the more *x* increases, the closer we come to 1/1, because *e* to the power of an infinitely large negative num- ber approaches zero; in contrast, the more *x* decreases below zero, the more the expression $(1 + e^{-\theta x})$ grows, shrinking the entire quotient. Because $(1 + e^{-\theta x})$ is in the denominator, the larger it becomes, the closer the quotient itself comes to zero.

With logistic regression, *f*(*x*) represents the probability that *y* equals 1 (i.e., is true) given each input *x*. If we are attempting to estimate the probability that an email is spam, and *f*(*x*) happens to equal 0.6, we could paraphrase that by saying *y* has a 60 percent of being 1, or the email has a 60 percent chance of being spam, given the input. If we define machine learning as a method to infer unknown outputs from known inputs, the parameter vector *x* in a logistic regression model determines the strength and certainty of our deductions.

The Logit Transformation

The logit function is the inverse of the logistic function ("logistic transform").

Evaluating Models

Evaluating models is the process of understanding how well they give the correct clas- sification and then measuring the value of the prediction in a certain context. Some- times, we only care how often a model gets any prediction correct; other times, it's important that the model gets a certain type of prediction correct more often than the others. In this section, we cover topics like bad positives, harmless negatives, unbal- anced classes, and unequal costs for predictions. Let's take a look at the basic tool for evaluating models: the *confusion matrix*.

The Confusion Matrix

The confusion matrix (see Figure 1-11)—also called a *table of confusion*—is a table of rows and columns that represents the predictions and the actual outcomes (labels) for a classifier. We use this table to better understand how well the model or classifier is performing based on giving the correct answer at the appropriate time.

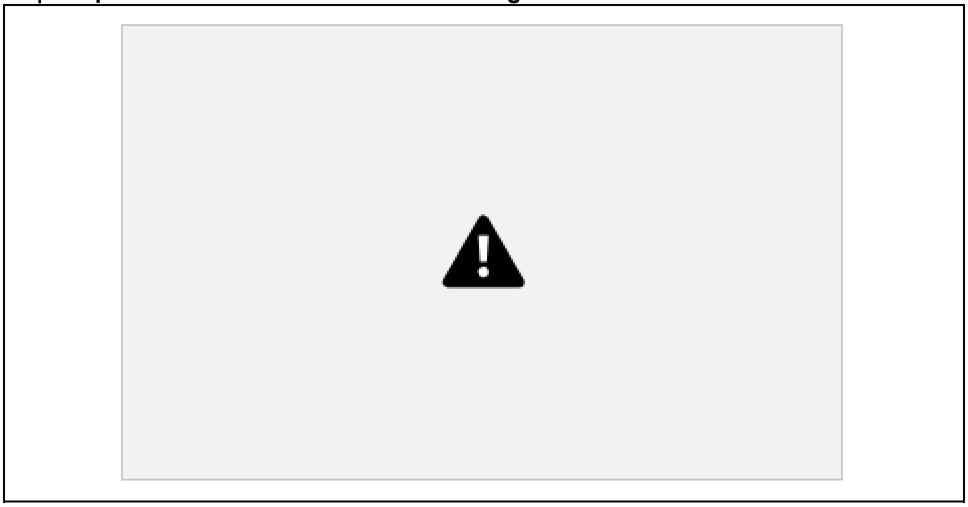


Figure 1-11. The confusion matrix

We measure these answers by counting the number of the following:

- True positives
  - Positive prediction
  - Label was positive
- False positives
  - Positive prediction
  - Label was negative
- True negatives
  - Negative prediction
  - Label was negative
- False negatives
  - Negative prediction
  - Label was positive

In traditional statistics a false positive is also known as "type I error" and a false nega- tive is known as a "type II error." By tracking these metrics, we can achieve a more detailed analysis on the performance of the model beyond the basic percentage of guesses that were correct. We can calculate different evaluations of the model based on combinations of the aforementioned four counts in the confusion matrix, as shown here:

```
Accuracy: 0.94
 Precision: 0.8662
 Recall: 0.8955
 F1 Score: 0.8806
```

In the preceding example, we can see four different common measures in the evalua- tion of machine learning models. We'll cover each of them shortly, but for now, let's begin with the basics of evaluating model sensitivity versus model specificity.

Sensitivity versus specicity

Sensitivity and *specificity* are two different measures of a binary classification model. The true positive rate measures how often we classify an input record as the positive class and its the correct classification. This also is called sensitivity, or recall; an exam- ple would be classifying a patient as having a condition who was actually sick. Sensi- tivity quantifies how well the model avoids false negatives.

Sensitivity = TP / (TP + FN)

If our model was to classify a patient from the previous example as not having the condition and she actually did not have the condition then this would be considered a true negative (also called specificity). Specificity quantifies how well the model avoids false positives.

Specificity = TN / (TN + FP)

Many times we need to evaluate the trade-off between sensitivity and specificity. An example would be to have a model that detects serious illness in patients more fre- quently because of the high cost of misdiagnosing a truly sick patient. We'd consider this model to have low specificity. A serious sickness could be a danger to the patient's life and to the others around him, so our model would be considered to have a high sensitivity to this situation and its implications. In a perfect world, our model would be 100 percent sensitive (i.e., all sick are detected) and 100 percent specific (i.e., no one who is not sick is classified as sick).

Accuracy

Accuracy is the degree of closeness of measurements of a quantity to that quantity's true value.

Accuracy = (TP + TN) / (TP + FP + FN + TN)

Accuracy can be misleading in the quality of the model when the class imbalance is high. If we simply classify everything as the larger class, our model will automatically get a large number of its guesses correct and provide us with a high accuracy score yet misleading indication of value based on a real application of the model (e.g., it will never predict the smaller class or rare event).

Precision

The degree to which repeated measurements under the same conditions give us the same results is called precision in the context of science and statistics. Precision is also known as the *positive prediction value*. Although sometimes used interchangeably with "accuracy" in colloquial use, the terms are defined differently in the frame of the scientific method.

Precision = TP / (TP + FP)

A measurement can be accurate yet not precise, not accurate but still precise, neither accurate nor precise, or both accurate and precise. We consider a measurement to be valid if it is both accurate and precise.

Recall

This is the same thing as sensitivity and is also known as the *true positive rate* or the *hit rate*.

F1

In binary classification we consider the F1 score (or F-score, F-measure) to be a measure of a model's accuracy. The F1 score is the harmonic mean of both the preci- sion and recall measures (described previously) into a single score, as defined here:

F1 = 2TP / (2TP + FP + FN)

We see scores for F1 between 0.0 and 1.0, where 0.0 is the worst score and 1.0 is the best score we'd like to see. The F1 score is typically used in information retrieval to see how well a model retrieves relevant results. In machine learning, we see the F1 score used as an overall score on how well our model is performing.

Context and interpreting scores

Context can play a role in how we evaluate our model and dictate when we use differ- ent types of scores, as described previously in this section. Class imbalance can play a large role in dictating choice of evaluation score, and in many datasets, we'll find that the classes or label counts are not well balanced. Here are some typical domains in which we see this:

- Web click prediction
- ICU mortality prediction
- Fraud detection

In these contexts, an overall "percent correct" score can be misleading to the overall value, in practical terms, of the model. An example of this would be the PhysioNet Challenge dataset from 2012.

The goal of the challenge was to "predict in-hospital *mortality* with the greatest accu- racy using a binary classifier." The difficulty and challenge in modeling this dataset is that predicting that a patient will live is the easy part because the bulk of examples in the dataset have outcomes in which the patient does live. Predicting death accurately in this scenario is the goal; this is where the model has the most value in the context of being clinically relevant in the real world. In this competition, the scores were cal- culate as follows:

Score = MIN(Precision, Recall)

This was set up such that it kept the contestants focused on not just predicting that the patient would live most of the time and get a nice F1 score, but focus on predict- ing when the patient would die (keeping the focus on being clinically relevant). This is a great example of how context can change how we evaluate our models.

Methods to Handle Class Imbalance

In Chapter 6 we illustrate practical ways to deal with class imbal- ance. We take a closer look at the different facets of class imbalance and error distributions in the context of classification and regres- sion.

Building an Understanding of Machine Learning

In this chapter, we introduced the core concepts needed for practicing machine learn- ing. We looked at the core mathematical concepts of modeling based around the equation:

$$Ax = b$$

We also looked at the core ideas of getting features into the matrix A, ways to change the parameter vector x, and setting the outcomes in the vector b. We illustrated some basic ways to change the parameter vector x to minimize the score (or "loss") of the objective function.

As we move forward in this book, we'll continue to expand on these key con- cepts. We'll see how neural networks and deep learning are based on these fundamen- tals but add more complex ways to create the A matrix, change the x parameter vector through optimization methods, and measure loss during training. Let's now move on to Chapter 2, where we build further on these concepts with the basics of neural net- works.

CHAPTER 2

Foundations of Neural Networks and Deep Learning

With your feet in the air and your head on the ground
Try this trick and spin it, yeah
Your head will collapse
But there's nothing in it
And you'll ask yourself
Where is my mind

—The Pixies, "Where is My Mind?"

Neural Networks

Neural networks are a computational model that shares some properties with the ani- mal brain in which many simple units are working in parallel with no centralized control unit. The weights between the units are the primary means of long-term information storage in neural networks. Updating the weights is the primary way the neural network learns new information.

In Chapter 1 we discussed modeling sets of equations in the form of the equation $Ax = b$. In the context of neural networks, the A matrix is still the input data and the b column vector is still the labels or outcomes for each row in the A matrix. The weights on the neural network connections becomes x (the parameter vector).

The behavior of neural networks is shaped by its network architecture. A network's architecture can be defined (in part) by the following:

- Number of neurons
- Number of layers
- Types of connections between layers

The most well-known and simplest-to-understand neural network is the feed forward multilayer neural network. It has an input layer, one or many hidden layers, and a single output layer. Each layer can have a different number of neurons and each layer is fully connected to the adjacent layer. The connections between the neurons in the layers form an acyclic graph, as illustrated in Figure 2-1.

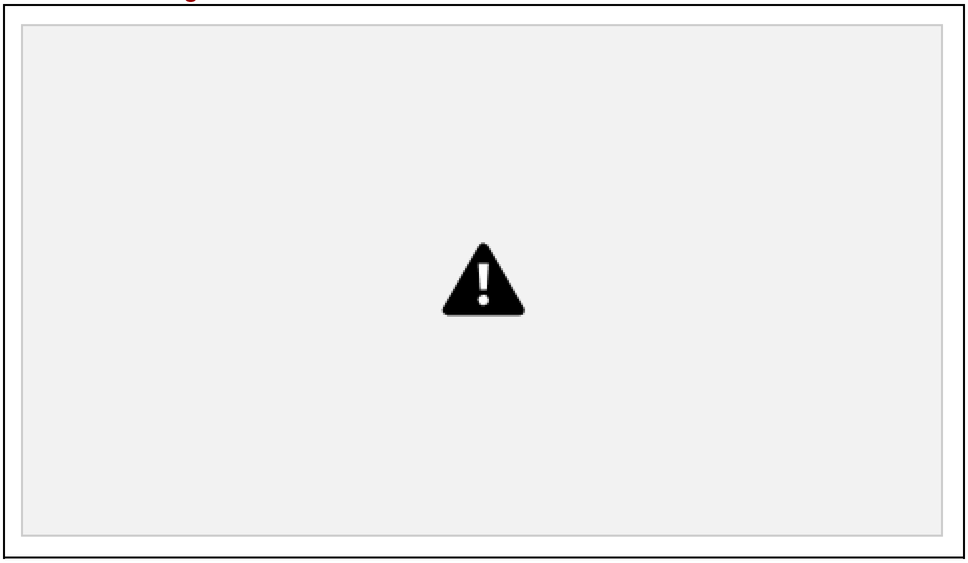


Figure 2-1. Multilayer neural network topology

A feed-forward multilayer neural network can represent any function, given enough artificial neuron units. It is generally trained by a learning algorithm called *backpro- pagation learning*. Backpropagation uses gradient descent (see Chapter 1) on the weights of the connections in a neural network to minimize the error on the output of the network.

Local Minima and Backpropagation

Backpropagation can become stuck in local minima, but in practice it generally performs well.

Historically, backpropagation has been considered slow, but recent advances in com- putational power through parallelism and graphics processing units (GPUs) have renewed interest in neural networks.

Much hubris and many words have been transmitted across the internet and in the literature about the connection of neural networks to the human mind. Let's separate some signal from the vast noise and begin with the biological inspiration of artificial neural networks.

The Mechanistic View of the Mind

Rather than constructing a rigid tree that requires all inputs to be one thing or another, we can construct a model that reflects a world sending us partial, ambiguous information, from which we can draw inferences with relative but not total certainty.

This aspect of neural networks represents a break from the mechanistic view of the mind, dominant in the early twentieth century, which assumed that our brain interlocked with the world in a deterministic way—like two gears meshing—with clear inputs leading to clear outputs. Now, we assume that, based on incomplete and sometimes contradictory information, humans find ways to plunge forward and act. The human brain infers from probabilities and so do neural networks.

We'll provide a brief review of the biological neuron and then take a look at the early precursor to modern neural networks: *the perceptron*. Building on our understanding of the perceptron, we'll then see how it evolved into the more generalized artificial neuron that supports today's modern feed-forward multilayer perceptrons. The cul- mination of this chapter will give you, the modern neural network practitioner, the fundamentals to delve further into more exotic deep network architectures.

The Biological Neuron

The biological neuron (see Figure 2-2) is a nerve cell that provides the fundamental functional unit for the nervous systems of all animals. Neurons exist to communicate with one another, and pass electro-chemical impulses across synapses, from one cell to the next, as long as the impulse is strong enough to activate the release of chemi- cals across a synaptic cleft. The strength of the impulse must surpass a minimum threshold or chemicals will not be released.

Figure 2-2 presents the major parts of the nerve cell:

- Soma
- Dendrites
- Axons
- Synapses

The neuron is made up of a nerve cell consisting of a soma (cell body) that has many dendrites but only one axon. The single axon can branch hundreds of times, however. Dendrites are thin structures that arise from the main cell body. Axons are nerve fibers with a special cellular extension that comes from the cell body.

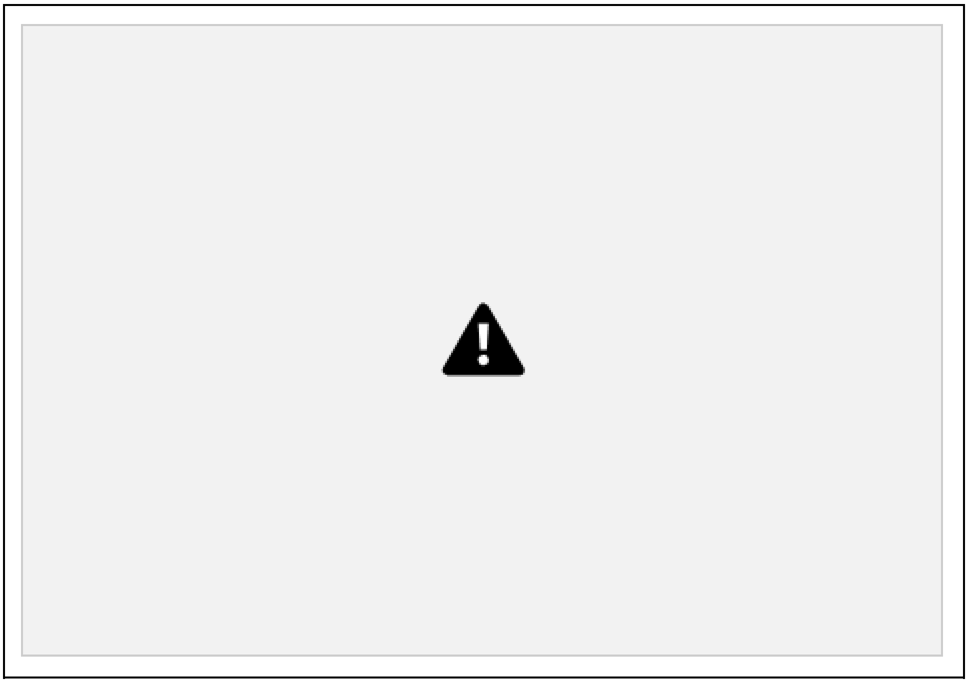


Figure 2-2. The biological neuron

Synapses

Synapses are the connecting junction between axon and dendrites. The majority of synapses send signals from the axon of a neuron to the dendrite of another neuron. The exceptions for this case are when a neuron might lack dendrites, or a neuron lacks an axon, or a synapse, which connects an axon to another axon.

Dendrites

Dendrites have fibers branching out from the soma in a bushy network around the nerve cell. Dendrites allow the cell to receive signals from connected neighboring neurons and each dendrite is able to perform multiplication by that dendrite’s weight value. Here multiplication means an increase or decrease in the ratio of synaptic neu- rotransmitters to signal chemicals introduced into the dendrite.

Axons

Axons are the single, long fibers extending from the main soma. They stretch out longer distances than dendrites and measure generally 1 centimeter in length (100 times the diameter of the soma). Eventually, the axon will branch and connect to other dendrites. Neurons are able to send electrochemical pulses through cross

membrane voltage changes generating *action potential*. This signal travels along the cell's axon and activates synaptic connections with other neurons.

Information ow across the biological neuron

Synapses that increase the potential are considered *excitatory*, and those that decrease the potential are considered *inhibitory*. *Plasticity* refers the long-term changes in strength of connections in response to input stimulus. Neurons also have been shown to form new connections over time and even migrate. These combined mechanics of connection change drive the learning process in the biological brain.

From biological to articial

The animal brain has been shown to be responsible for the fundamental components of the mind. We can study the basic components of the brain and understand them. Research has shown ways to map out functionality of the brain and track signals as they move through neurons.

Convolutional Neural Networks and the Mammalian Vision System

Later in the book, we take a look at a deep network called a Convolutional Neural Network (CNN). A CNN's image representation at different layers is similar to how the brain processes visual information. Although this research is interesting, it does not mean that a CNN gives us a full approximation of mammalian brain activity.

However, we still do not completely understand how this collection of decentralized functional units provides the foundation for thought and the seat of consciousness.

The Seat of Consciousness

In the eighteenth century, the brain began to be recognized as the "seat of consciousness." By the late nineteenth century, animal brains began to be mapped out to better understand its functional regions. Previous locations of consciousness included the heart and, oddly enough, the spleen.

Now that we've established the basics of how a biological neuron works, let's take a look at the first attempts at modeling the neuron with the advent of the perceptron.

The Perceptron

The perceptron is a linear model used for binary classification. In the field of neural networks the perceptron is considered an artificial neuron using the Heaviside step function for the activation function, both of which we'll define further later in the

chapter. The precursor to the perceptron was the Threshold Logic Unit (TLU) devel- oped by McCulloch and Pitts in 1943, which could learn the AND and OR logic func- tions. The perceptron training algorithm is considered a supervised learning algorithm. Both the TLU and the perceptron were inspired by the biological neuron as we'll explore.

History of the perceptron

The perceptron was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It was funded by the US Office of Naval Research and was covered by the *New York Times*:

> [T]he embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

Obviously these predictions were a bit premature—as we've seen many times with the promise of machine learning and AI. Early versions were intended to be implemented as a physical machine rather than a software program. The first software implementa- tion was for the IBM 704, and then later it was implemented in the Mark I Perceptron machine.

It also should be noted that McCulloch and Pitts introduced the basic concept of ana- lyzing neural activity in 1943[1] based on thresholds and weighted sums. These con- cepts were key in developing a model for later variations like the perceptron.

The Mark I Perceptron

The Mark I Perceptron was designed for image recognition for military purposes by the US Navy. The Mark I Perceptron had 400 photocells connected to artificial neurons in the machine, and the weights were implemented by potentiometers. Weight updates were physically performed by electric motors.

Denition of the perceptron

The perceptron is a linear-model binary classifier with a simple input–output rela- tionship as depicted in Figure 2-3, which shows we're summing *n* number of inputs times their associated weights and then sending this "net input" to a step function with a defined threshold. Typically with perceptrons, this is a Heaviside step function with a threshold value of 0.5. This function will output a real-valued single binary value (0 or a 1), depending on the input.

[1] McCulloch and Pitts. 1943. “A logical calculus of the ideas immanent in nervous activity.”

$\mathbf{w} \cdot \mathbf{x}$ Dot product ($\sum_{i=1}^{n} w_i x_i$)

Figure 2-3. Single-layer perceptron

We can model the decision boundary and the classification output in the Heaviside step function equation, as follows:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x >= 0 \end{cases}$$

To produce the net input to the activation function (here, the Heaviside step func- tion) we take the dot product of the input and the connection weights. We see this summation in the left half of Figure 2-3 as the input to the summation function. Table 2-1 provides an explanation of how the summation function is performed as well as notes about the parameters involved in the summation function.

Table 2-1. The summation function parameters

w	Vector of real-valued weights on the connections
n	Number of inputs to the perceptron
b	The bias term (input value does not affect its value; shifts decision boundary away from origin)

The output of the step function (activation function) is the output for the perceptron and gives us a classification of the input values. If the bias value is negative, it forces

the learned weights sum to be a much greater value to get a 1 classification output. The bias term in this capacity moves the decision boundary around for the model. Input values do not affect the bias term, but the bias term is learned through the per- ceptron learning algorithm.

The Single-Layer Perceptron

The perceptron is more widely known as a “single-layer percep- tron” in neural network research to distinguish it from its successor the “multilayer perceptron.”

As a basic linear classifier we consider the single-layer perceptron to be the simplest form of the family of feed-forward neural networks.

Relating the Perceptron to the Biological Neuron

Although we don’t have a complete model for how the brain works, we do see how the perceptron was modeled after the biological neuron. In this case, we can see how the perceptron takes input through connections with weights on them in a similar fashion to how synapses pass information to other biological neurons.

The perceptron learning algorithm

The perceptron learning algorithm changes the weights in the perceptron model until all input records are all correctly classified. The algorithm will not terminate if the learning input is not linearly separable. A linearly separable dataset is one for which we can find the values of a hyperplane that will cleanly divide the two classes of the dataset.

The perceptron learning algorithm initializes the weight vector with small random values or 0.0s at the beginning of training. The perceptron learning algorithm takes each input record, as we can see in Figure 2-3, and computes the output classification to check against the actual classification label. To produce the classification, the col- umns (features) are matched up to weights where *n* is the number of dimensions in both our input and weights. The first input value is the bias input, which is always 1.0 because we don't affect the bias input. The first weight is our bias term in this dia- gram. The dot product of the input vector and the weight vector gives us the input to our activation function, as we've previously discussed.

If the classification is correct, no weight changes are made. If the classification is incorrect, the weights are adjusted accordingly. Weights are updated between individ- ual training examples in an "online learning" fashion. This loop continues until all of the input examples are correctly classified. If the dataset is not linearly separable, the

training algorithm will not terminate. Figure 2-4 demonstrates a dataset that is not linearly separable, the XOR logic function.

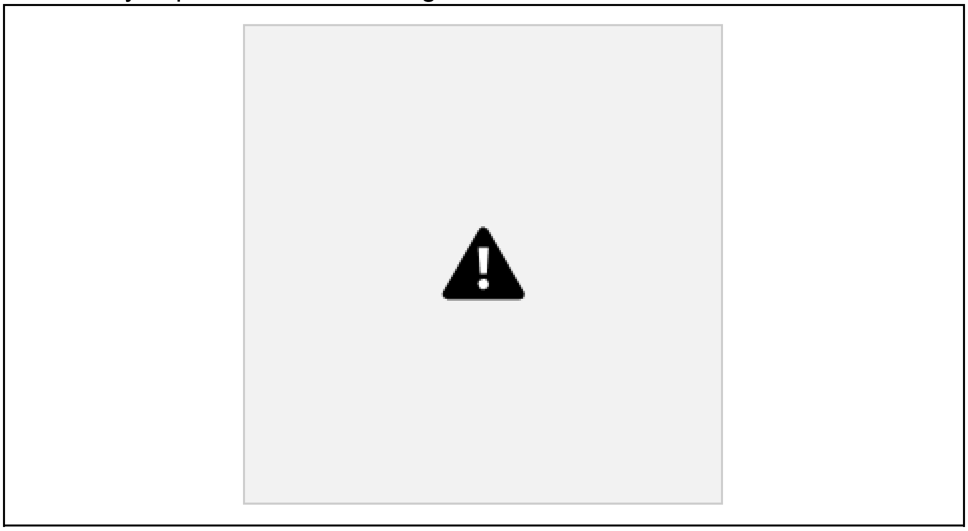


Figure 2-4. The XOR function

A basic perceptron (single-layer variant) cannot solve the XOR logic modeling prob- lem, illustrating an early limitation of the perceptron model.

Limitations of the early perceptron

After initial promise, the perceptron was found to be limited in the types of patterns it could recognize. The initial inability to solve nonlinear (e.g., datasets that are not linearly separable) problems was seen as a failure for the field of neural networks. The 1969 book *Perceptrons* by Minsky and Papert illustrated the limitations of the single layer perceptron. However, what the general industry did not widely realize was that a multilayer perceptron could indeed solve the XOR problem, among many other non- linear problems.

AI Winter I: 1974–1980

The misunderstanding of the multilayer perceptron capabilities was an early public setback that hurt interest in, and funding of, neural networks for the next decade. It wasn't until the resurgence of neural networks in the mid-1980s that backpropagation became popular (although backpropagation was originally discovered in 1974 by Webos) and neural networks enjoyed a second wave of interest.

Multilayer Feed-Forward Networks

The multilayer feed-forward network is a neural network with an input layer, one or more hidden layers, and an output layer. Each layer has one or more artificial neu- rons. These artificial neurons are similar to their perceptron precursor yet have a dif- ferent activation function depending on the layer's specific purpose in the network. We'll look more closely at the layer types in multilayer perceptrons later in the chap- ter. For now, let's look more closely at this evolved artificial neuron that emerged from the limitations of the single-layer perceptron.

Evolution of the articial neuron

The artificial neuron of the multilayer perceptron is similar to its predecessor, the perceptron, but it adds flexibility in the type of activation layer we can use. Figure 2-5 shows an updated diagram for the artificial neuron that is based on the perceptron.

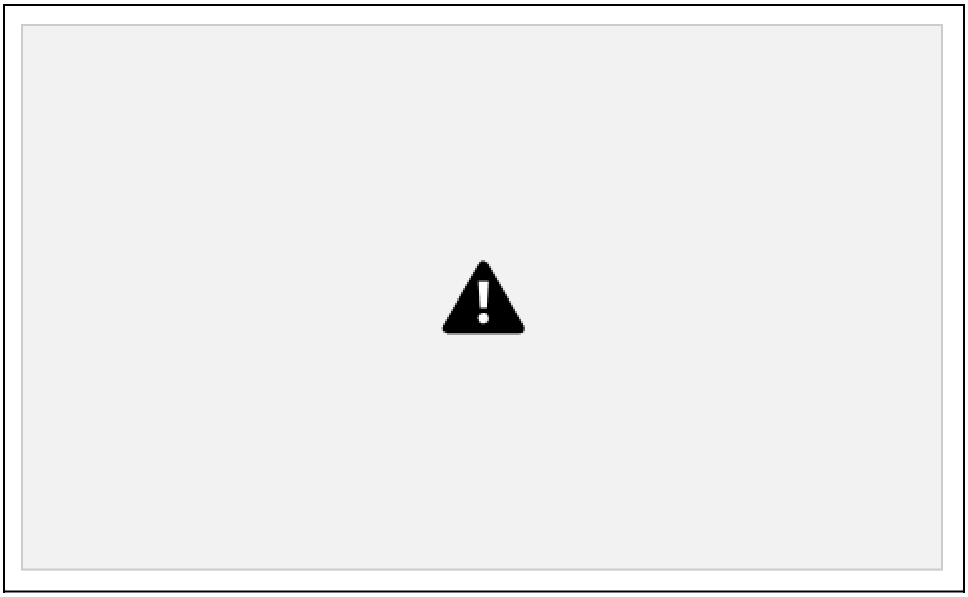


Figure 2-5. Artificial neuron for a multilayer perceptron

This diagram is similar to Figure 2-3 for the single-layer perceptron, yet we notice a more generalized activation function. We'll develop this diagram further in a detailed look at the artificial neuron as we proceed.

A Note About the Term "Neuron"

From this point throughout the remainder of the book, when we use the term "neuron" we're referring to the artificial neuron based on Figure 2-5.

The net input to the activation function is still the dot product of the weights and input features yet the flexible activation function allows us to create different types out of output values. This is a major contrast to the earlier perceptron design that used a piecewise linear Heaviside step function as this improvement now allowed the artificial neuron because express more complex activation output.

Articial neuron input. The artificial neuron (see Figure 2-6) takes input that, based on the weights on the connections, can be ignored (by a 0.0 weight on an input connec- tion) or passed on to the activation function. The activation function also has the ability to filter out data if it does not provide a non-zero activation value as output.

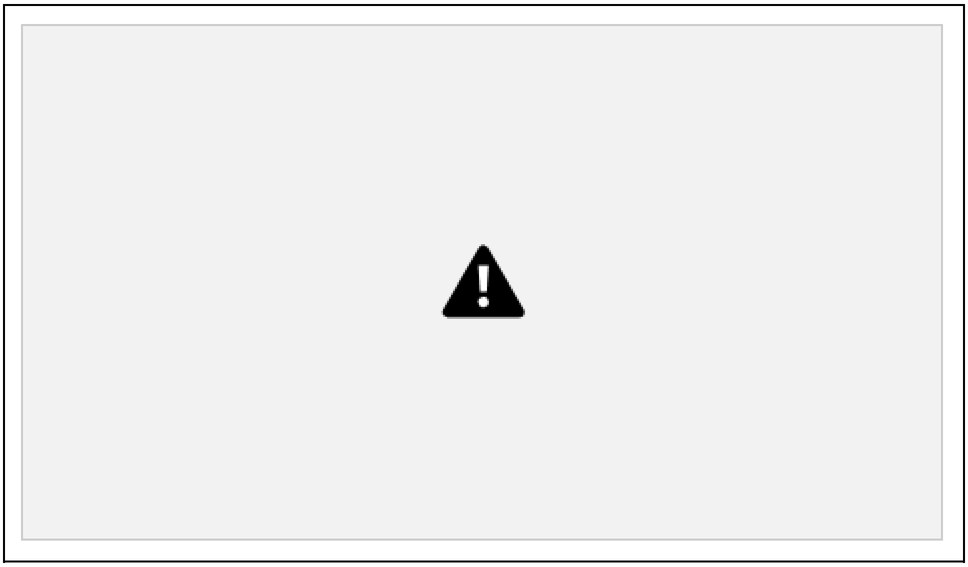


Figure 2-6. Details of an artificial neuron in a multilayer perceptron neural network

We express the net input to a neuron as the weights on connections multiplied by activation incoming on connection, as shown in Figure 2-6. For the input layer, we're just taking the feature at that specific index, and the activation function is linear (it passes on the feature value). For hidden layers, the input is the activation from other neurons. Mathematically, we can express the net input (total weighted input) of the artificial neuron as

$$input\_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i$$

where $\mathbf{W}_i$ is the vector of all weights leading into neuron *i* and $\mathbf{A}_i$ is the vector of acti- vation values for the inputs to neuron *i*. Let's build on this equation by accounting for the bias term that is added per layer (explained further below):

$$input\_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i + b$$

To produce output from the neuron, we'd then wrap this net input with an activation function *g*, as demonstrated in the following equation:

$$a_i = g\ input\_sum_i$$

We can then expand this function with the definition of $input_i$:

$$a_i = g\ \mathbf{W}_i \cdot \mathbf{A}_i + b$$

This activation value for neuron *i* is the output value passed on to the next layer through connections to other artificial neurons (multiplied by weights on connec- tions) as an input value.

Different Notations

An alternate notation for expressing the output of an artificial neuron we often see in research papers is as follows:

$h_{w,b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$

This notation is slightly different from the previous equation above. We show this alternate notation to illustrate how some papers will use slightly different notations to explain these concepts, and we want you to be able to recognize the variants.

If our activation function is the sigmoid function, we'll have this:

$$g(z) = \frac{1}{1 + e^{-z}}$$

This output will have the range [0, 1], which is the same output as the logistic regression function.

Sigmoid Activation Functions in Practice

We list the sigmoid activation function here for historical demonstration. As we'll see later on in this book, the sigmoid activation function has largely fallen out of favor.

The inputs are the data from which you want to produce information, and the con- nection weights and biases are the quantities that govern the activity, activating it or not. As with the perceptron, there is a learning algorithm to change the weights and bias value for each artificial neuron. During the training phase, the weights and biases

change as the network learns. We'll cover the learning algorithm for neural networks later in the chapter.

Just as biological neurons don't pass on every electro-chemical impulse they receive, artificial neurons are not just wires or diodes passing on a signal. They are designed to be selective. They filter the data they receive, and aggregate, convert, and transmit only certain information to the next neuron(s) in the network. As these filters and transformations work on data, they convert raw input data to useful information in the context of the larger multilayer perceptron neural network. We illustrate this effect more in the next section.

Artificial neurons can be defined by the kind of input they are able to receive (binary or continuous) and the kind of transform (activation function) they use to produce output. In DL4J, all neurons in a layer have the same activation function.

Connection weights. Weights on connections in a neural network are coefficients that scale (amplify or minimize) the input signal to a given neuron in the network. In common representations of neural networks, these are the lines/arrows going from one point to another, the edges of the mathematical graph. Often, connections are notated as *w* in mathematical representations of neural networks.

Biases. Biases are scalar values added to the input to ensure that at least a few nodes per layer are activated regardless of signal strength. Biases allow learning to happen by giving the network action in the event of low signal. They allow the network to try new interpretations or behaviors. Biases are generally notated *b*, and, like weights, biases are modified throughout the learning process.

Activation functions. The functions that govern the artificial neuron's behavior are called activation functions. The transmission of that input is known as *forward propa- gation*. Activation functions transform the combination of inputs, weights, and biases. Products of these transforms are input for the next node layer. Many (but not all) nonlinear transforms used in neural networks transform the data into a convenient range, such as 0 to 1 or –1 to 1. When an artificial neuron passes on a nonzero value to another artificial neuron, it is said to be *activated*.

Activations

Activations are the values passed on to the next layer from each previous layer. These values are the output of the activation function of each artificial neuron.

In "Activation Functions" on page 65 we'll review the different types of activation functions and their general function in the broader context of neural networks.

Activation Functions and Their Importance

Activation functions and their usage will be a continuing theme throughout almost every chapter for the remainder of this book. The DL4J library uses a layer-based architecture revolving around different types of activation functions.

Comparing the biological neuron and the articial neuron

If we loop back for a moment and think about the biological neuron on which the artificial neuron is based, we can ask, “How close does the artificial variant match up to the biological version?” The concepts match up with the input connection func- tionality being performed by dendrites in the biological neuron and the summation functionality being provided by the soma. Finally, we see the activation function being performed by the axon in the biological neuron.

Limitations of Comparisons

We note (again) that the biological neuron is still more complex than the artificial variant. Research continues toward a better understanding of the biological neuron’s function.

Feed-forward neural network architecture

Now that we understand the differences between the artificial neuron and the percep- tron, we can better understand the structure of the full multilayer feed-forward neu- ral network. With multilayer feed-forward neural networks, we have artificial neurons arranged into groups called layers. Building on the layer concept, we see that the multilayer neural network has the following:

- A single input layer
- One or many hidden layers, fully connected
- A single output layer

As Figure 2-7 depicts, the neurons in each layer (represented by the circles) are all fully connected to all neurons in all adjacent layers.

The neurons in each layer all use the same type of activation function (most of the time). For the input layer, the input is the raw vector input. The input to neurons of the other layers is the output (activation) of the previous layer’s neurons. As data moves through the network in a feed-forward fashion, it is influenced by the connec- tion weights and the activation function type. Let’s now take a look at the specifics of each layer type.

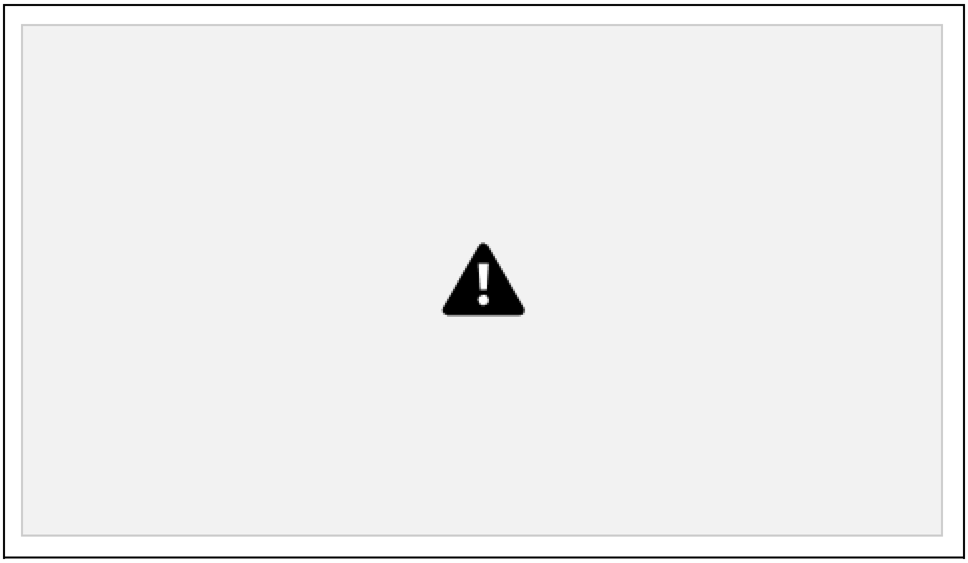


Figure 2-7. Fully connected multilayer feed-forward neural network topology

Input layer. This layer is how we get input data (vectors) fed into our network. The number of neurons in an input layer is typically the same number as the input feature to the network. Input layers are followed by one or more hidden layers (explained in the next section). Input layers in classical feed-forward neural networks are fully con- nected to the next hidden layer, yet in other network architectures, the input layer might not be fully connected.

Hidden layer. There are one or more hidden layers in a feed-forward neural network. The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data. Hidden layers are the key to allowing neural networks to model nonlinear functions, as we saw from the limitations of the single-layer perceptron networks.

Output layer. We get the answer or prediction from our model from the output layer. Given that we are mapping an input space to an output space with the neural network model, the output layer gives us an output based on the input from the input layer. Depending on the setup of the neural network, the final output may be a real valued output (regression) or a set of probabilities (classification). This is controlled by the type of activation function we use on the neurons in the output layer. The out- put layer typically uses either a *somax* or *sigmoid activation function* for classifica- tion. We'll discuss the difference between these two types of activation functions later in the chapter.

Connections between layers. In a fully connected feed-foward network, the connec- tions between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the next layer. We change these weights progressively as our algorithm finds the best solution it can with the backpropagation learning algo- rithm. We can understand the weights mathematically by thinking of them as the parameter vector in the earlier linear algebra section describing the machine learning process as optimizing the parameter vector (e.g., “weights” here) to minimize error.

We’ve now covered the basic structure of feed-forward neural networks. The rest of this chapter provides a more detailed look at the training mechanics of backpropaga- tion as well as specifics of activation functions. The chapter closes with a review of common loss functions and hyperparameters.

Training Neural Networks

A well-trained artificial neural network has weights that amplify the signal and dampen the noise. A bigger weight signifies a tighter correlation between a signal and the network’s outcome. Inputs paired with large weights will affect the network’s interpretation of the data more than inputs paired with smaller weights.

The process of learning for any learning algorithm using weights is the process of re adjusting the weights and biases, making some smaller and others larger, thereby allocating significance to certain bits of information and minimizing other bits. This helps our model learn which predictors (or features) are tied to which outcomes, and adjusts the weights and biases accordingly.

In most datasets, certain features are strongly correlated with certain labels (e.g., square footage relates to the sale price of a house). Neural networks learn these rela- tionships blindly by making a guess based on the inputs and weights and then meas- uring how accurate the results are. The loss functions in optimization algorithms, such as stochastic gradient descent (SGD), reward the network for good guesses and penalize it for bad ones. SGD moves the parameters of the network toward making good predictions and away from bad ones.

Another way to look at the learning process is to view labels as theories and the fea- ture set as evidence. Then, we can make the analogy that the network seeks to estab- lish the correlation between the theory and the evidence. The model attempts to answer the question “which theory does the evidence support?” With these ideas in mind, let’s take a look at the learning algorithm most commonly associated with neu- ral networks: *backpropagation learning*.