

Deep Learning

A PRACTITIONER'S APPROACH

Josh Patterson &
Adam Gibson

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 dev needs to know to get started with deep learning in the real world.”

—Grant Ingersoll
CTO, Lucidworks

Josh Patterson is currently VP of Field Engineering for Skymind. Previously, Josh worked as a Principal Solutions Architect at Cloudera and as a machine learning and distributed systems engineer at the Tennessee Valley Authority.

Adam Gibson is the CTO of Skymind. Adam has worked with Fortune 500 companies, hedge funds, PR firms, and startup accelerators to create their machine learning projects. He has a strong track record helping companies handle and interpret big realtime data.

US \$49.99 CAN \$65.99

ISBN: 978-1-491-91425-0



9 781491 914250



Twitter: @oreillymedia
facebook.com/oreilly

Deep Learning

A Practitioner's Approach

Josh Patterson and Adam Gibson

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Deep Learning

by Josh Patterson and Adam Gibson

Copyright © 2017 Josh Patterson and Adam Gibson. All rights reserved.

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/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Mike Loukides and Tim McGovern

Indexer: Judy McConville

Production Editor: Nicholas Adams

Interior Designer: David Futato

Copieditor: Bob Russell, Octal Publishing, Inc.

Cover Designer: Karen Montgomery

Proofreader: Christina Edwards

Illustrator: Rebecca Demarest

August 2017: First Edition

Revision History for the First Edition

2017-07-27: First Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Deep Learning*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91425-0

[M]

For my sons Ethan, Griffin, and Dane: Go forth, be persistent, be bold.

—J. Patterson

Table of Contents

Preface.....	xiii
1. A Review of Machine Learning.....	1
The Learning Machines	1
How Can Machines Learn?	2
Biological Inspiration	4
What Is Deep Learning?	6
Going Down the Rabbit Hole	7
Framing the Questions	8
The Math Behind Machine Learning: Linear Algebra	8
Scalars	9
Vectors	9
Matrices	10
Tensors	10
Hyperplanes	10
Relevant Mathematical Operations	11
Converting Data Into Vectors	11
Solving Systems of Equations	13
The Math Behind Machine Learning: Statistics	15
Probability	16
Conditional Probabilities	18
Posterior Probability	19
Distributions	19
Samples Versus Population	22
Resampling Methods	22
Selection Bias	22
Likelihood	23
How Does Machine Learning Work?	23

Regression	23
Classification	25
Clustering	26
Underfitting and Overfitting	26
Optimization	27
Convex Optimization	29
Gradient Descent	30
Stochastic Gradient Descent	32
Quasi-Newton Optimization Methods	33
Generative Versus Discriminative Models	33
Logistic Regression	34
The Logistic Function	35
Understanding Logistic Regression Output	35
Evaluating Models	36
The Confusion Matrix	36
Building an Understanding of Machine Learning	40
2. Foundations of Neural Networks and Deep Learning.....	41
Neural Networks	41
The Biological Neuron	43
The Perceptron	45
Multilayer Feed-Forward Networks	50
Training Neural Networks	56
Backpropagation Learning	57
Activation Functions	65
Linear	66
Sigmoid	66
Tanh	67
Hard Tanh	68
Softmax	68
Rectified Linear	69
Loss Functions	71
Loss Function Notation	71
Loss Functions for Regression	72
Loss Functions for Classification	75
Loss Functions for Reconstruction	77
Hyperparameters	78
Learning Rate	78
Regularization	79
Momentum	79
Sparsity	80

3. Fundamentals of Deep Networks.....	81
Defining Deep Learning	81
What Is Deep Learning?	81
Organization of This Chapter	91
Common Architectural Principles of Deep Networks	92
Parameters	92
Layers	93
Activation Functions	93
Loss Functions	95
Optimization Algorithms	96
Hyperparameters	100
Summary	105
Building Blocks of Deep Networks	105
RBMs	106
Autoencoders	112
Variational Autoencoders	114
4. Major Architectures of Deep Networks.....	117
Unsupervised Pretrained Networks	118
Deep Belief Networks	118
Generative Adversarial Networks	121
Convolutional Neural Networks (CNNs)	125
Biological Inspiration	126
Intuition	126
CNN Architecture Overview	128
Input Layers	130
Convolutional Layers	130
Pooling Layers	140
Fully Connected Layers	140
Other Applications of CNNs	141
CNNs of Note	141
Summary	142
Recurrent Neural Networks	143
Modeling the Time Dimension	143
3D Volumetric Input	146
Why Not Markov Models?	148
General Recurrent Neural Network Architecture	149
LSTM Networks	150
Domain-Specific Applications and Blended Networks	159
Recursive Neural Networks	160
Network Architecture	160
Varieties of Recursive Neural Networks	161

Applications of Recursive Neural Networks	161
Summary and Discussion	162
Will Deep Learning Make Other Algorithms Obsolete?	162
Different Problems Have Different Best Methods	162
When Do I Need Deep Learning?	163
5. Building Deep Networks.....	165
Matching Deep Networks to the Right Problem	165
Columnar Data and Multilayer Perceptrons	166
Images and Convolutional Neural Networks	166
Time-series Sequences and Recurrent Neural Networks	167
Using Hybrid Networks	169
The DL4J Suite of Tools	169
Vectorization and DataVec	170
Runtimes and ND4J	170
Basic Concepts of the DL4J API	172
Loading and Saving Models	172
Getting Input for the Model	173
Setting Up Model Architecture	173
Training and Evaluation	174
Modeling CSV Data with Multilayer Perceptron Networks	175
Setting Up Input Data	178
Determining Network Architecture	178
Training the Model	181
Evaluating the Model	181
Modeling Handwritten Images Using CNNs	182
Java Code Listing for the LeNet CNN	183
Loading and Vectorizing the Input Images	185
Network Architecture for LeNet in DL4J	186
Training the CNN	190
Modeling Sequence Data by Using Recurrent Neural Networks	191
Generating Shakespeare via LSTMs	191
Classifying Sensor Time-series Sequences Using LSTMs	200
Using Autoencoders for Anomaly Detection	207
Java Code Listing for Autoencoder Example	207
Setting Up Input Data	211
Autoencoder Network Architecture and Training	211
Evaluating the Model	213
Using Variational Autoencoders to Reconstruct MNIST Digits	214
Code Listing to Reconstruct MNIST Digits	214
Examining the VAE Model	217
Applications of Deep Learning in Natural Language Processing	221

Learning Word Embedding Using Word2Vec	221
Distributed Representations of Sentences with Paragraph Vectors	227
Using Paragraph Vectors for Document Classification	231
6. Tuning Deep Networks.....	237
Basic Concepts in Tuning Deep Networks	237
An Intuition for Building Deep Networks	238
Building the Intuition as a Step-by-Step Process	239
Matching Input Data and Network Architectures	240
Summary	241
Relating Model Goal and Output Layers	242
Regression Model Output Layer	242
Classification Model Output Layer	243
Working with Layer Count, Parameter Count, and Memory	246
Feed-Forward Multilayer Neural Networks	246
Controlling Layer and Parameter Counts	247
Estimating Network Memory Requirements	250
Weight Initialization Strategies	251
Using Activation Functions	253
Summary Table for Activation Functions	255
Applying Loss Functions	256
Understanding Learning Rates	258
Using the Ratio of Updates-to-Parameters	259
Specific Recommendations for Learning Rates	260
How Sparsity Affects Learning	263
Applying Methods of Optimization	263
SGD Best Practices	265
Using Parallelization and GPUs for Faster Training	265
Online Learning and Parallel Iterative Algorithms	266
Parallelizing SGD in DL4J	269
GPUs	272
Controlling Epochs and Mini-Batch Size	273
Understanding Mini-Batch Size Trade-Offs	274
How to Use Regularization	275
Priors as Regularizers	275
Max-Norm Regularization	276
Dropout	277
Other Regularization Topics	279
Working with Class Imbalance	280
Methods for Sampling Classes	282
Weighted Loss Functions	282
Dealing with Overfitting	283

Using Network Statistics from the Tuning UI	284
Detecting Poor Weight Initialization	287
Detecting Nonshuffled Data	288
Detecting Issues with Regularization	290
7. Tuning Specific Deep Network Architectures.....	293
Convolutional Neural Networks (CNNs)	293
Common Convolutional Architectural Patterns	294
Configuring Convolutional Layers	297
Configuring Pooling Layers	303
Transfer Learning	304
Recurrent Neural Networks	306
Network Input Data and Input Layers	307
Output Layers and RnnOutputLayer	308
Training the Network	309
Debugging Common Issues with LSTMs	311
Padding and Masking	312
Evaluation and Scoring With Masking	313
Variants of Recurrent Network Architectures	314
Restricted Boltzmann Machines	314
Hidden Units and Modeling Available Information	315
Using Different Units	316
Using Regularization with RBMs	317
DBNs	317
Using Momentum	318
Using Regularization	319
Determining Hidden Unit Count	319
8. Vectorization.....	321
Introduction to Vectorization in Machine Learning	321
Why Do We Need to Vectorize Data?	322
Strategies for Dealing with Columnar Raw Data Attributes	325
Feature Engineering and Normalization Techniques	327
Using DataVec for ETL and Vectorization	334
Vectorizing Image Data	336
Image Data Representation in DL4J	337
Image Data and Vector Normalization with DataVec	339
Working with Sequential Data in Vectorization	340
Major Variations of Sequential Data Sources	340
Vectorizing Sequential Data with DataVec	341
Working with Text in Vectorization	347
Bag of Words	348

TF-IDF	349
Comparing Word2Vec and VSM Comparison	353
Working with Graphs	354
9. Using Deep Learning and DL4J on Spark.....	357
Introduction to Using DL4J with Spark and Hadoop	357
Operating Spark from the Command Line	360
Configuring and Tuning Spark Execution	362
Running Spark on Mesos	363
Running Spark on YARN	364
General Spark Tuning Guide	367
Tuning DL4J Jobs on Spark	371
Setting Up a Maven Project Object Model for Spark and DL4J	372
A pom.xml File Dependency Template	374
Setting Up a POM File for CDH 5.X	378
Setting Up a POM File for HDP 2.4	378
Troubleshooting Spark and Hadoop	379
Common Issues with ND4J	380
DL4J Parallel Execution on Spark	381
A Minimal Spark Training Example	383
DL4J API Best Practices for Spark	385
Multilayer Perceptron Spark Example	387
Setting Up MLP Network Architecture for Spark	390
Distributed Training and Model Evaluation	390
Building and Executing a DL4J Spark Job	392
Generating Shakespeare Text with Spark and Long Short-Term Memory	392
Setting Up the LSTM Network Architecture	395
Training, Tracking Progress, and Understanding Results	396
Modeling MNIST with a Convolutional Neural Network on Spark	397
Configuring the Spark Job and Loading MNIST Data	400
Setting Up the LeNet CNN Architecture and Training	401
A. What Is Artificial Intelligence?.....	405
B. RL4J and Reinforcement Learning.....	417
C. Numbers Everyone Should Know.....	441
D. Neural Networks and Backpropagation: A Mathematical Approach.....	443
E. Using the ND4J API.....	449

F. Using DataVec.....	463
G. Working with DL4J from Source.....	475
H. Setting Up DL4J Projects.....	477
I. Setting Up GPUs for DL4J Projects.....	483
J. Troubleshooting DL4J Installations.....	487
Index.....	495

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 covering “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 largest 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 fundamentals 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 techniques 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 practitioner of any deep learning library. [Chapter 8](#) is a review of the techniques of vectorization 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 different 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 audience 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 fundamentals. 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 science. 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 production 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 appreciated 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 terminology. You might want to skip the chapters that are heavy on APIs and examples, however.

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 networks 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 learning 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 elements 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 determined 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 significant 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 Patterson 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



Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/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 bookquestions@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 Patterson 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 website 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 meeting at MLConf) would end up as a book. To be fair, while I was there from the beginning 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 uncertain 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 correctness 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 academic'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 communicate 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 Ingwersol, 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 Chattanooga, 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.

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 connections 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 computational 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 knowledge 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 learning 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 visually, [Figure 1-1](#) illustrates our conception of the relationship between AI, machine learning, and deep learning.

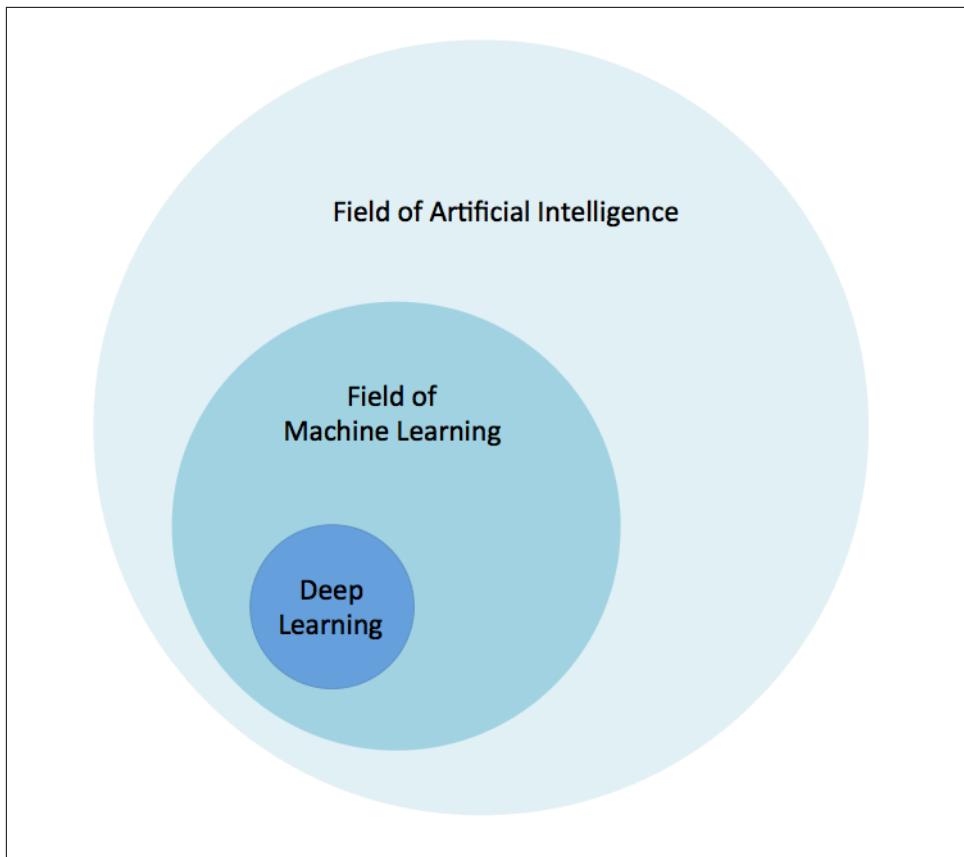


Figure 1-1. The relationship between AI and deep learning

The field of AI is broad and has been around for a long time. Deep learning is a subset 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 connected 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 neuron 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 comparative structure.



Comparing Biological with Artificial

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
- Termites¹
- 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 coordinating 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 architectures:

- Unsupervised pretrained networks
- Convolutional neural networks
- Recurrent neural networks
- Recursive neural networks

There are some variations of the aforementioned architectures—a hybrid convolutional 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 exhaustive 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 decades 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 techniques 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](#).

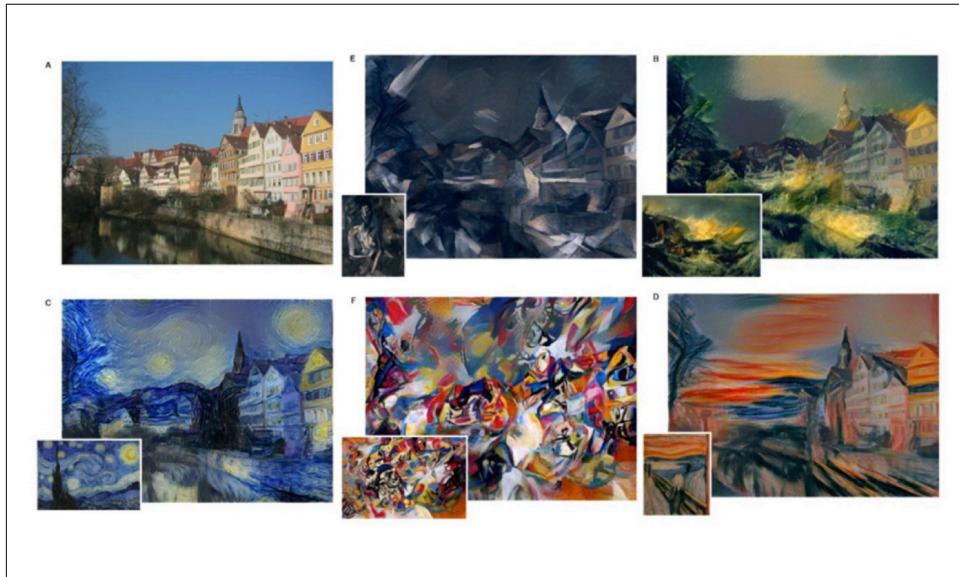


Figure 1-2. Stylized images by Gatys et al., 2015²

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 quantity 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 process 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:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix}$$

Or:

$$x = [x_1, x_2, x_3, \dots, x_n]$$

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 $n \times m$ 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.

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

Figure 1-3. A 3×3 matrix

Tensors

A *tensor* is a multidimensional array at the most fundamental level. It is a more general 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 summing up the products thus obtained. Without getting too mathematical (immediately), 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 *normalization*, 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 *coseine similarity*.

Element-wise product

Another common linear algebra operation we see in practice is the *element-wise product* (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 multiplied 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 everything? 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 concepts 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 vectorization 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 follows). 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 2:0.4166666666666663 3:0.702127659574468 4:0.
5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.4583333333333326 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 1:0.9166666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0
.5652173913043479
0.0 1:0.0833333333333343 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	0.4166666666666663	0.702127659574468	0.5652173913043479
0.6666666666666666	0.5	0.9148936170212765	0.6956521739130436
0.4583333333333326	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 techniques 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 computing 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 inversion*. Given that not all matrices can be inverted, we'd like a method to solve this equation 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 linear 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 conditions 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 equations 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* (SGD), 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 averaged 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 process. 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 highlight 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 statistics 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 generalizing from a sample to a population. Here are some examples of inferential statistics:

- 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 context, 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., $0.35 \times 100 == 35$ 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 sample 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 meaning of probability and odds. Before we proceed, let's clarify this here.

The probability of an event E is defined as:

$$P(E) = (\text{Chances for } E) / (\text{Total Chances})$$

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:

$$(\text{Chances for } E) : (\text{Chances Against } E)$$

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

$$4 : (52 - 4) = 1/12 = 0.0833333\dots$$

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 networks. For a larger review of statistics, check out O'Reilly's *Statistics in a Nutshell: A Desktop Quick Reference* by Boslaugh and Watters.

Further Defining 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 measurement. 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 certainty 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 statements 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 presence of another event occurring, we express this as a *conditional probability*. This is expressed in literature in the form:

$$P(E | F)$$

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:

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

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 interested in conditional probabilities in machine learning in the context in which we’d learn a classifier by learning

$$P(E | F)$$

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 measurements 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 probability that a patient who tests positive on a test for a specific disease actually has the disease.

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 conditional probability we assign after the evidence is considered. Posterior probability distribution is defined as the probability distribution of an unknown quantity conditional on the evidence collected from an experiment treated as a random variable. 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 variables. 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 values 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 assumptions 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 distribution (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 variations.

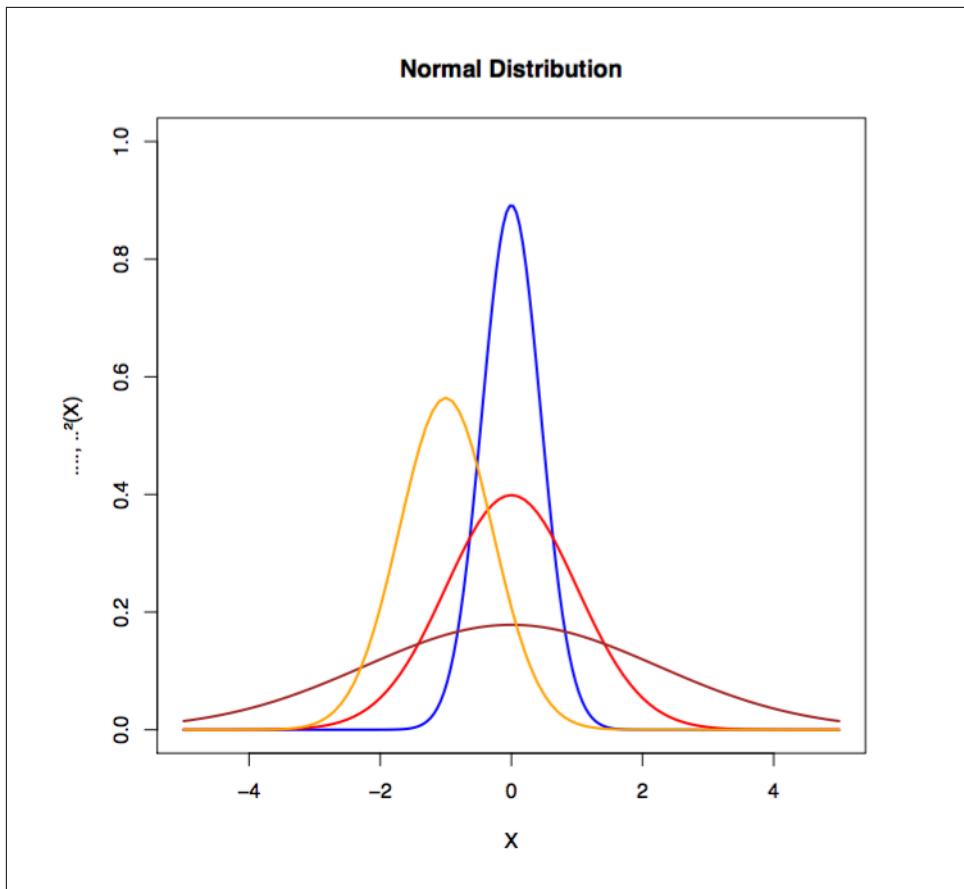


Figure 1-5. Examples of normal distributions

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 approximate 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 population that gradually decreases in asymptotic fashion. These distributions were discovered 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 consider 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 represented 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 seasoned 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 programmers 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 statistics that are useful to machine learning practitioners. In the context of machine learning 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 dataset 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 randomization 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 reference 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 techniques 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 outcomes 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 optimization algorithms as the book progresses. We'll also cover the basics of hyperparameters 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 previously 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 vector). 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:

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

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 dependent variable and the length of the commute is the independent variable. It's reasonable to keep track of these two quantities and then define a function, like so:

$$\text{cost} = f(\text{distance})$$

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](#).

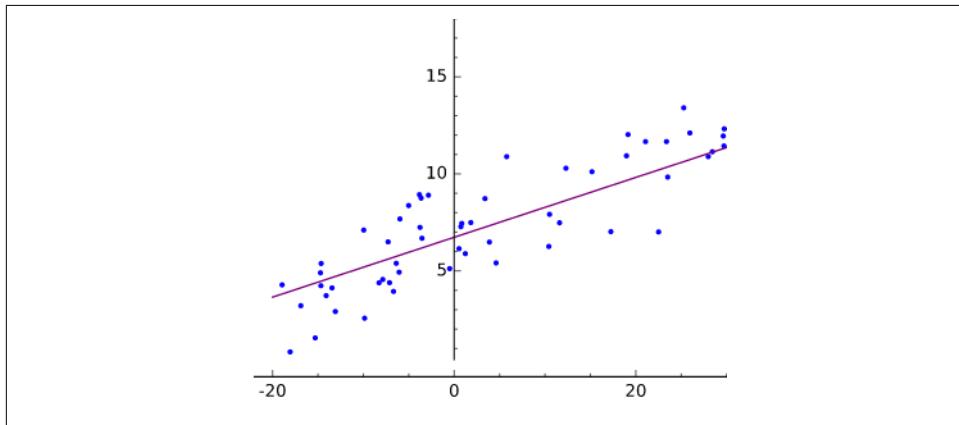


Figure 1-6. Linear regression plotted

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 coordinates, 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 outcomes 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 minimize 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.

Classification

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 certainty. 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 measure and iteratively moving similar items more closely together. At the end of the process, 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.

Underfitting and Overfitting

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 scatterplot would be a good example of underfitting, as is illustrated in [Figure 1-7](#).

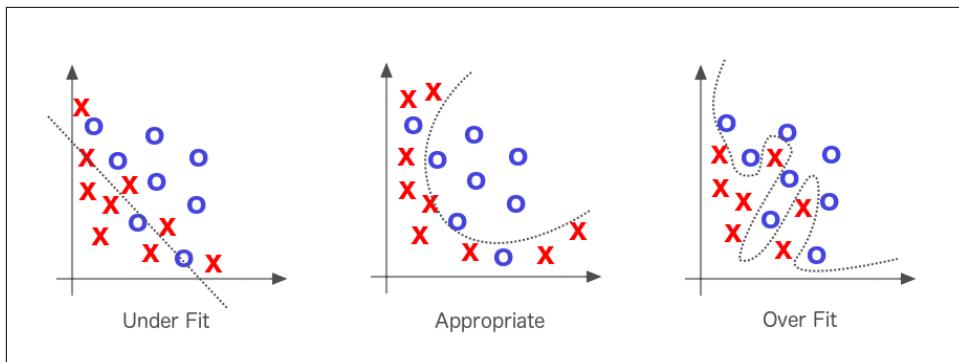


Figure 1-7. Underfitting and overfitting in machine learning

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 generalize 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 process 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 conjectures 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 network predicts (\hat{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](#). The 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 hyperparameters.



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 optimization:

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 optimization.

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.

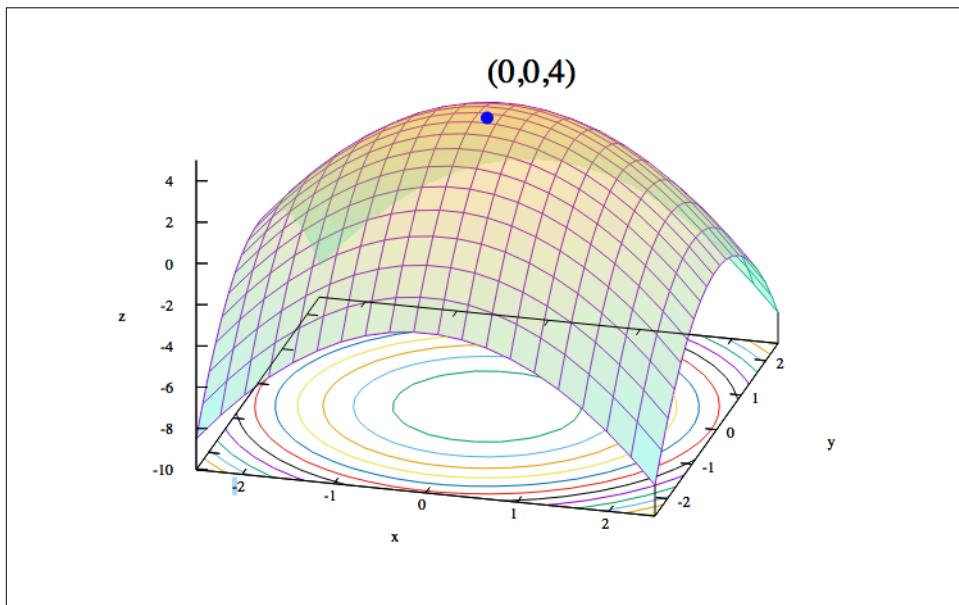


Figure 1-8. Visualizing convex functions

Another way to relate parameters to the data is with a maximum likelihood estimation, or MLE. The MLE traces a parabola whose edges point downward, with likelihood 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 algorithm 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 important, 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. Gradient 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](#).

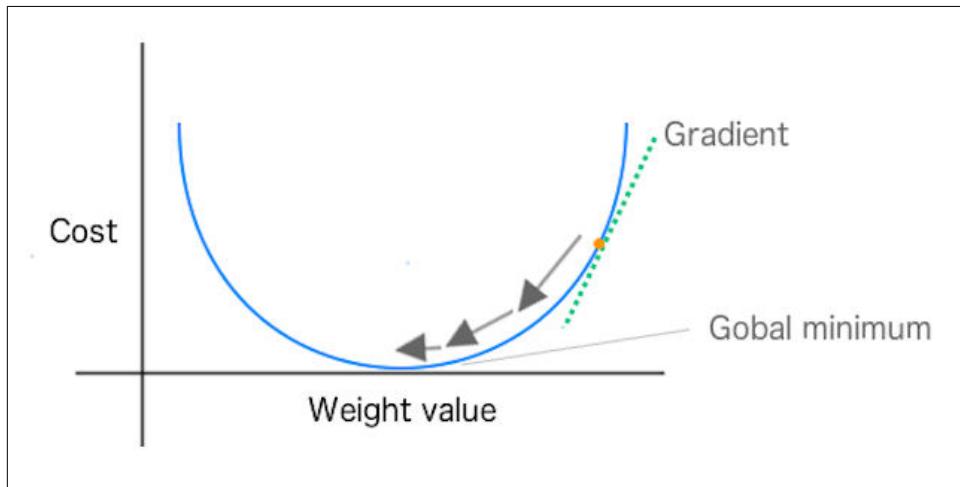


Figure 1-9. Showing weight changes toward global minimum in SGD

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 optimization, 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 dimension 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 slowly 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 prediction 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 parameter 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 gradient descent reaches a local minimum, it is effectively trapped, and this is one drawback 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 compute 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 computed 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 efficiency. 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. Conversely, 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 methods 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 vectors 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 methods 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 matrices 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* models. 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 signal. 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 learning.

A generative model learns the *joint* probability distribution $p(x, y)$, whereas a discriminative model learns the *conditional* probability distribution $p(y|x)$. The distribution $p(y|x)$ 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 components 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 logistic 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.

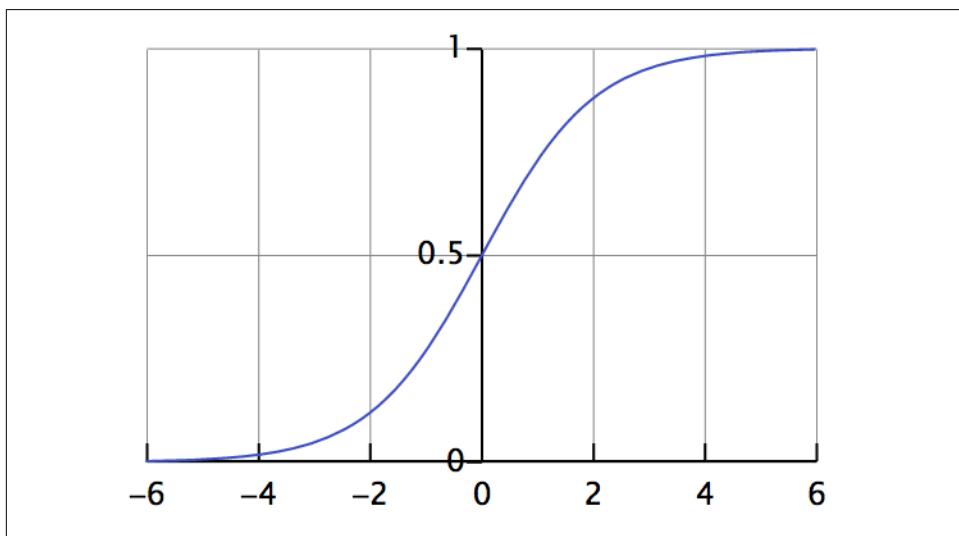


Figure 1-10. Plot of the logistic function

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 number 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 classification and then measuring the value of the prediction in a certain context. Sometimes, 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, unbalanced 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.

	P' (Predicted)	N' (Predicted)
P (Actual)	True Positive	False Negative
N (Actual)	False Positive	True Negative

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 negative 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 evaluation 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 specificity

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 example would be classifying a patient as having a condition who was actually sick. Sensitivity quantifies how well the model avoids false negatives.

$$\text{Sensitivity} = \text{TP} / (\text{TP} + \text{FN})$$

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.

$$\text{Specificity} = \text{TN} / (\text{TN} + \text{FP})$$

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 frequently 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.

$$\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{FP} + \text{FN} + \text{TN})$$

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.

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

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 precision and recall measures (described previously) into a single score, as defined here:

$$F1 = 2\text{TP} / (2\text{TP} + \text{FP} + \text{FN})$$

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 different 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 accuracy 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 calculate as follows:

$$\text{Score} = \text{MIN}(\text{Precision}, \text{Recall})$$

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 predicting 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 imbalance. We take a closer look at the different facets of class imbalance and error distributions in the context of classification and regression.

Building an Understanding of Machine Learning

In this chapter, we introduced the core concepts needed for practicing machine learning. 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 concepts. We’ll see how neural networks and deep learning are based on these fundamentals 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 networks.

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 animal 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](#).

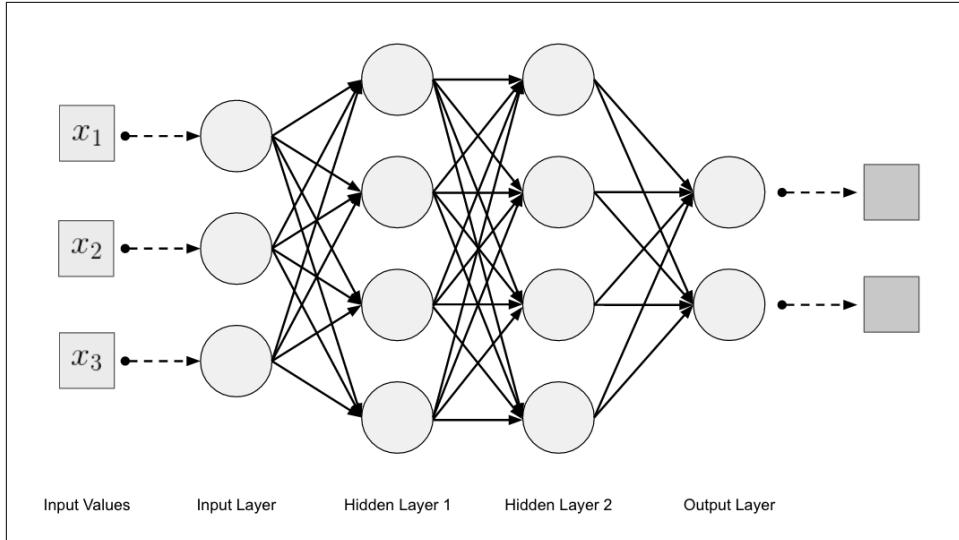


Figure 2-1. Multilayer neural network topology

A feed-forward multilayer neural network can represent any function, given enough artificial neuron units. It is generally trained by a learning algorithm called *backpropagation 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 computational 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 culmination 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 chemicals 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.

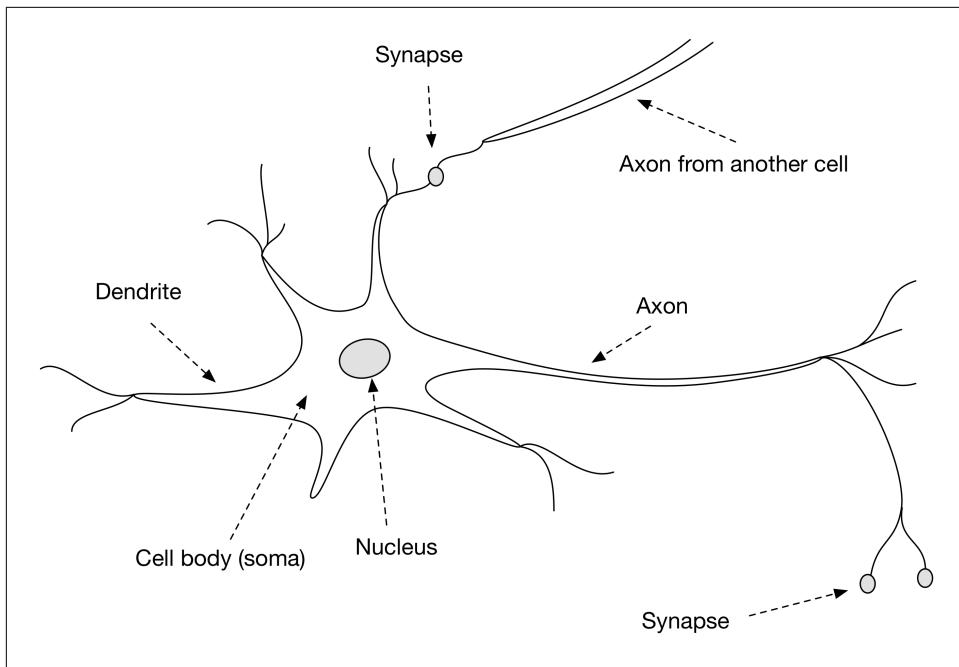


Figure 2-2. The biological neuron

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 neurotransmitters 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 flow across the biological neuron

Synapses that increase the potential are considered *excitatory*, and those that decrease the potential are considered *inhibitory*. *Plasticity* refers to 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 artificial

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) developed by McCulloch and Pitts in 1943, which could learn the AND and OR logic functions. 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 implementation 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 analyzing neural activity in 1943¹ based on thresholds and weighted sums. These concepts 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.

Definition of the perceptron

The perceptron is a linear-model binary classifier with a simple input–output relationship 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.”

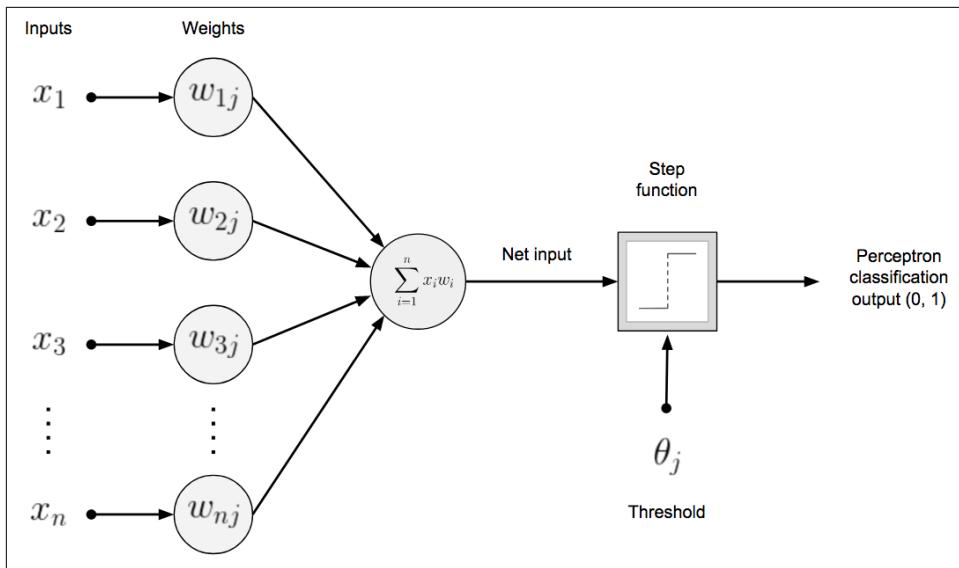


Figure 2-3. Single-layer perceptron

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

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

To produce the net input to the activation function (here, the Heaviside step function) 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

Function parameter	Description
w	Vector of real-valued weights on the connections
w · x	Dot product ($\sum_{i=1}^n w_i x_i$)
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 perceptron learning algorithm.



The Single-Layer Perceptron

The perceptron is more widely known as a “single-layer perceptron” 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 columns (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 diagram. 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 individual 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.

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2-4. The XOR function

A basic perceptron (single-layer variant) cannot solve the XOR logic modeling problem, 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 neurons. These artificial neurons are similar to their perceptron precursor yet have a different 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 chapter. 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 artificial 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.

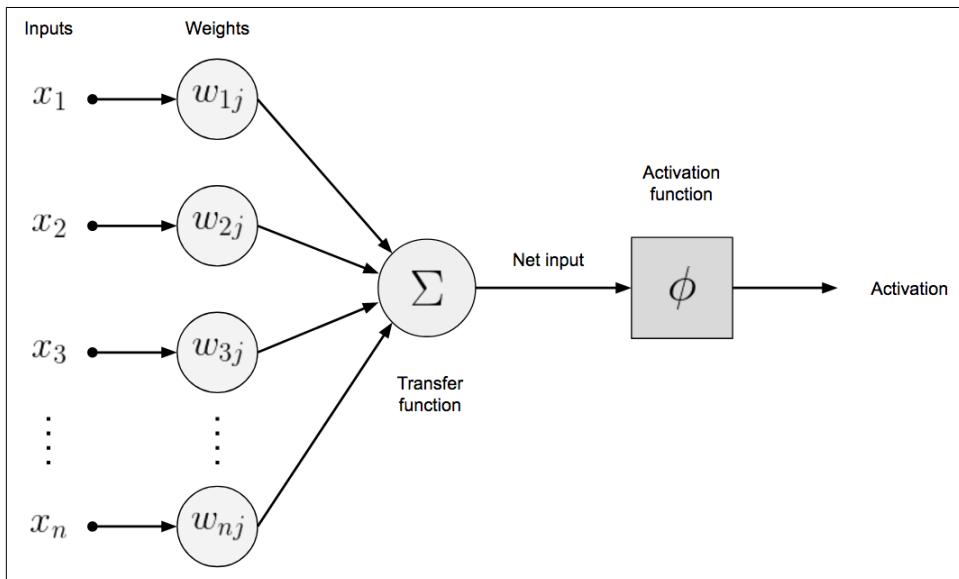


Figure 2-5. Artificial neuron for a multilayer perceptron

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.

Artificial 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 connection) 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.

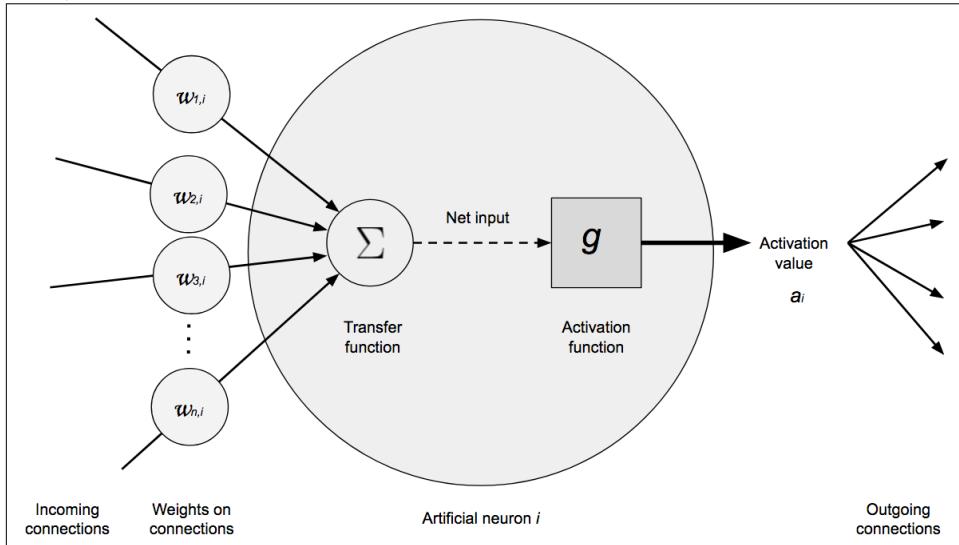


Figure 2-6. Details of an artificial neuron in a multilayer perceptron neural network

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

$$\text{input_sum}_i = \mathbf{W}_i \cdot \mathbf{A}_i$$

where \mathbf{W}_i is the vector of all weights leading into neuron i and \mathbf{A}_i is the vector of activation 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):

$$\text{input_sum}_i = \mathbf{W}_i \cdot \mathbf{A}_i + b$$

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

$$a_i = g(\text{input_sum}_i)$$

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 connections) 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 connection 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 propagation*. 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 artificial 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 functionality 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 perceptron, we can better understand the structure of the full multilayer feed-forward neural 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 connection weights and the activation function type. Let’s now take a look at the specifics of each layer type.

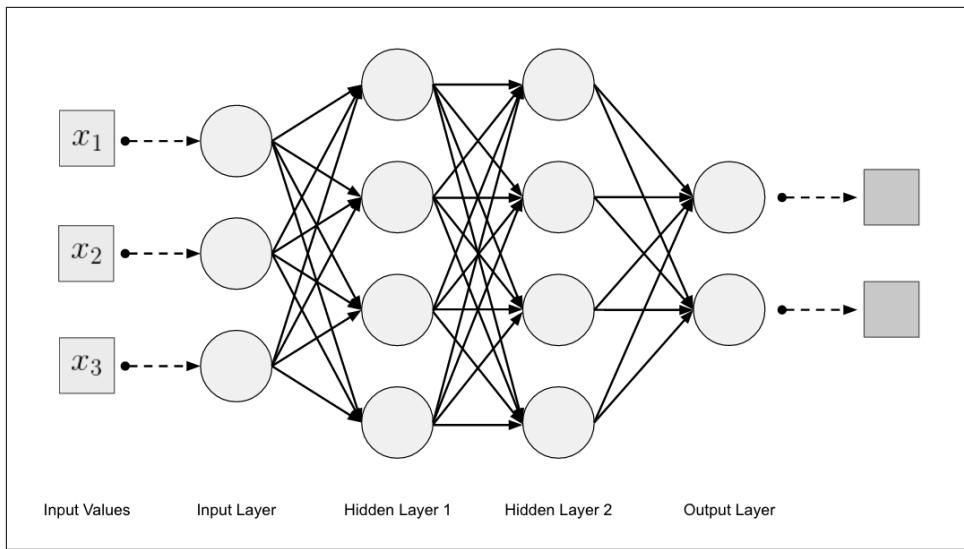


Figure 2-7. Fully connected multilayer feed-forward neural network topology

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 connected 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 output layer typically uses either a *softmax* or *sigmoid activation function* for classification. We'll discuss the difference between these two types of activation functions later in the chapter.

Connections between layers. In a fully connected feed-forward network, the connections 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 algorithm. 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 backpropagation 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 relationships blindly by making a guess based on the inputs and weights and then measuring 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 feature set as evidence. Then, we can make the analogy that the network seeks to establish 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 neural networks: *backpropagation learning*.

Backpropagation Learning

Backpropagation is an important part of reducing error in a neural network model. To explain backpropagation, we'll return to our discussion about how information circulates within a feed-forward neural network. Let's look at how this learning algorithm works before we dive deeper into the mathematical notation and pseudocode for backpropagation learning.



Origins of Backpropagation Learning

Backpropagation learning was first invented by Bryson and Ho in 1969. It was largely ignored in research and practice for neural network training until a resurgence in the mid-1980s.

Algorithm intuition

Backpropagation learning is similar to the perceptron learning algorithm. We want to compute the input example's output with a forward pass through the network. If the output matches the label, we don't do anything. If the output does not match the label, we need to adjust the weights on the connections in the neural network.

To further illustrate general neural network learning, let's take a look at the pseudocode for the algorithm, as shown in [Example 2-1](#).

Example 2-1. General neural network training pseudocode

```
function neural-network-learning( training-records ) returns network
    network <- initialize weights (randomly)
    start loop
        for each example in training-records do
            network-output = neural-network-output( network, example )
            actual-output = observed outcome associated with example
            update weights in network based on
                { example, network-output, actual-output }
        end for
    end loop when all examples correctly predicted or hit stopping conditions
    return network
```

The key is to distribute the blame for the error and divide it between the contributing weights. With the perceptron learning algorithm, it's easy because there is only one weight per input to influence the output value. With feed-forward multilayer networks learning algorithms have a bigger challenge. There are many weights connecting each input to the output, so it becomes more difficult. Each weight contributes to more than one output, so our learning algorithm must be more clever.

Backpropagation is a pragmatic approach to dividing the contribution of error for each weight. It is similar to the perceptron learning algorithm. With backpropagation, we're trying to minimize the error between the label (or “actual”) output associated with the training input and the value generated from the network output. In the next section, we take a look at the mathematical notation that you will see in most literature on neural networks for backpropagation of feed-forward neural networks.



A Caution on Learning Algorithms in Multilayer Networks

Learning algorithms for multilayer neural networks are neither guaranteed to converge to a global optimum nor are they absolutely efficient. This is a direct effect of how learning a general function from training examples is considered, in the worst case, an intractable problem. Learning algorithms do, however, work decently in practice with some decent hyperparameter tuning.

A closer look at backpropagation

For most of this book, we don't intend to throw a lot of math at you. However, for the topic of backpropagation and to better understand a core fundamental concept that much of the book is based on, we feel we should provide a section that illustrates these concepts down to the notation level.



A Note About Notation

This notation is similar to what you would see in a conference paper on machine learning or in a well-known machine learning textbook. Hopefully, we can explain the notation in a way that will feel comfortable for you. Ideally, this will serve as a jump-off point for you to explore many more neural network and deep learning papers down the road.

Let's zoom in on the previous diagram to focus on the input layer and the first hidden layer. [Figure 2-8](#) provides the view.

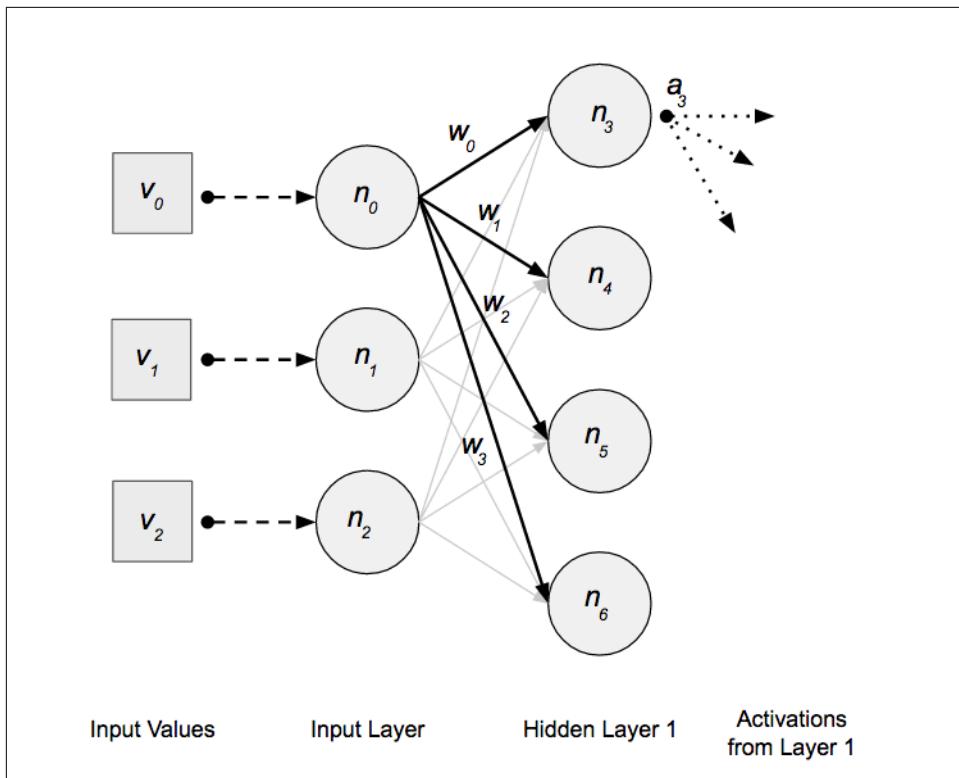


Figure 2-8. Multilayer perceptron network zoomed-in with component labels

In [Figure 2-8](#) we see our previous multilayer neural network diagram zoomed-in and updated with some notation. We'll use this notation for the rest of this section's explanation of backpropagation. [Table 2-2](#) lists the notations in the figure.

Table 2-2. Neural network notation

Notation	Meaning
i	Index of artificial neuron
n_i	Neuron at index i
j	Index of neuron in previous layer connecting to neuron i
a_i	Activation value of neuron i (output of neuron i)
\mathbf{A}_i	Vector of activation values for the inputs into neuron i
g	Activation function
g'	Derivative of the activation function
Err_i	Difference between the network output and the actual output value for the training example
\mathbf{W}_i	Vector of weights leading into neuron i
$W_{j,i}$	Weight on the incoming connection from previous layer neuron j to neuron i

Notation	Meaning
$input_sum_i$	Weighted sum of inputs to neuron i
$input_sum_j$	Weighted sum of inputs for neuron j in previous layer (used in backpropagation)
α	Learning rate
Δ_j	Error term for connected neuron j in previous layer
Δ_i	Error term for neuron i ; $= Err_i \times g'(input_sum_i)$

To further set the stage for explaining this algorithm, let's take a look at the pseudocode of the backpropagation learning algorithm, as shown in [Example 2-2](#).

Example 2-2. Backpropagation algorithm for updating weights pseudocode

```

function backpropagation-algorithm
  ( network, training-records, learning-rate ) returns network
  network <- initialize weights (randomly)
  start loop
    for each example in training-records do

      // compute the output for this input example
      network-output <- neural-network-output( network, example )

      // compute the error and the [delta] for neurons in the output layer
      example_err <- target-output - network-output

      // update the weights leading to the output layer
       $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(input\_sum_i)$ 

      for each subsequent-layer in network do

        // compute the error at each node
         $\Delta_j \leftarrow g'(input\_sum_j) \sum_i W_{j,i} \Delta_i$ 

        // update the weights leading into the layer
         $W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$ 

      end for

    end for
  end loop when network has converged
  return network

```



A Note About Loss Functions in the Pseudocode

The loss function (described later in this chapter) is not explicitly called out in the pseudocode in [Example 2-2](#). We've presented backpropagation in algorithmic form here for readability purposes because it's most approachable to the general practitioner. However, to give consideration to the mathematics-minded practitioner, we also have a version of backpropagation explained mathematically in [Appendix C](#).

In this case, the Err_i term is dependent on the derivative of the loss function. We're using mean squared error (MSE), so the derivative works out to be the difference.

Understanding backpropagation pseudocode

[Example 2-2](#) has the following inputs:

- Network: a multilayer feed-forward neural network
- Training records: a set of training vectors with associated outputs
- Learning rate: the learning rate (sometimes denoted by the Greek alpha symbol)

To start the algorithm, we initialize our neural network and begin looping through the input examples (until we encounter a terminating condition or a maximum number of epochs). First, we compute the output of the current network for the current input example. We compare this output to the actual output associated with the input and compute the error (*example_err*).

Now we're ready to compute the weight updates leading to the output layer.

Updating the output layer weights. The output layer weights update is calculated by using the following:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(input_sum_i)$$

This is the weight update rule for the connection between neuron j and neuron i for all connections in the neural network. We can see these connections leading into the output layer highlighted in [Figure 2-9](#).

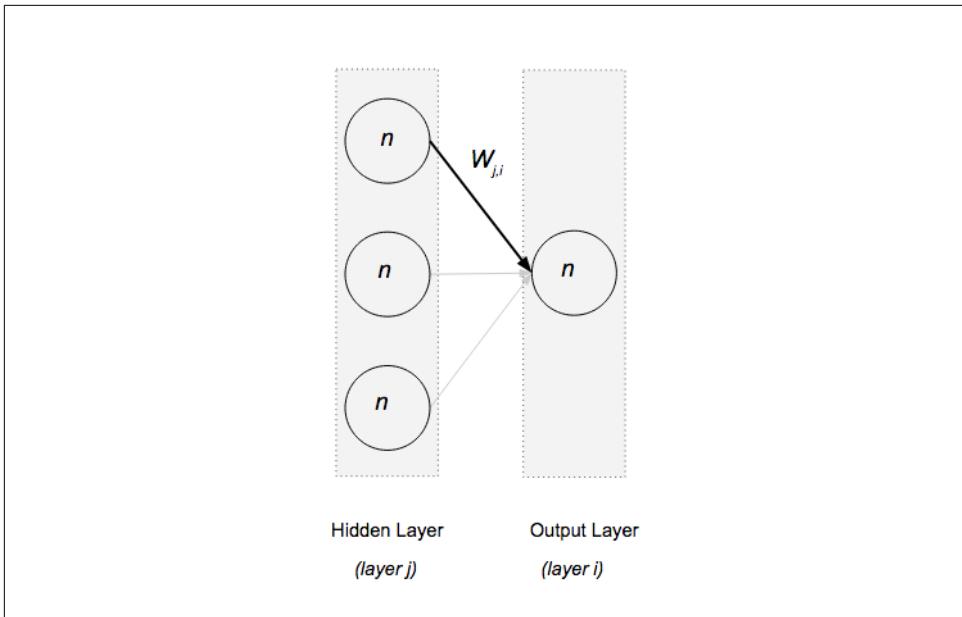


Figure 2-9. Updating connections into the output layer

Breaking this diagram down further, we can see we are working with the neuron j weight from the previous hidden layer connecting to the current neuron i . We're multiplying the learning rate α (discussed later in the chapter) by the incoming activation from the j neuron. This input is calculated by getting the net input to the j neuron and then computing the activation of neuron j .

Computing the total weight input to the activation function for neuron i , we compute the dot product of the incoming weight vector \mathbf{W}_j and the activation vector \mathbf{A}_j and then add in the bias term value:

$$\text{input_sum}_i = \mathbf{W}_j \cdot \mathbf{A}_j + b$$

Here is the equation for computing the activation value of neuron j :

$$a_j = g(\text{input_sum}_j)$$

The error term for example e at neuron i is denoted as Err_i . We denote the derivative of the activation function as $g'(x)$, which is applied to the net input of neuron i with the term:

$$g'(\text{input_sum}_i)$$

This update rule is similar to how we'd update a perceptron except we're using the activations of the previous layers as opposed to their raw input values. This rule also contains a term for the derivative of the activation function to get the gradient of the activation function.

Further expressing the error term. We saw in the previous section that $g'(z)$ gave us the derivative of the activation function. The error term is commonly expressed as Δ_i giving us:

$$\Delta_i = Err_i \times g'(input_sum_i)$$

This gives us a more condensed weight update function, expressed as the following:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

We see this update expression used in the inner loop of the pseudocode, as we'll now discuss.



Gradient Descent in Weight Space

We consider backpropagation to be doing gradient descent in weight space where the gradient is on the error surface. This error surface describes the error of the input features as a function of the weight values in the neural network.

The new propagation rule for the error value. The propagation rule for the delta values now becomes the following:

$$\Delta_j \leftarrow g'(input_sum_j) \Sigma_i W_{j,i} \Delta_i$$

This gives us a new update rule for weights between the inputs and the hidden layer.

Updating the hidden layers. With the backpropagation algorithm, we walk back across the hidden layers, updating the connection between each one until we reach the input layer. [Figure 2-10](#) illustrates connections between the two hidden layers highlighted.

To update these connections we take the input from the fractional error value computed previously and multiply it by the activation (input) from the connection from the previous layer and the learning rate.

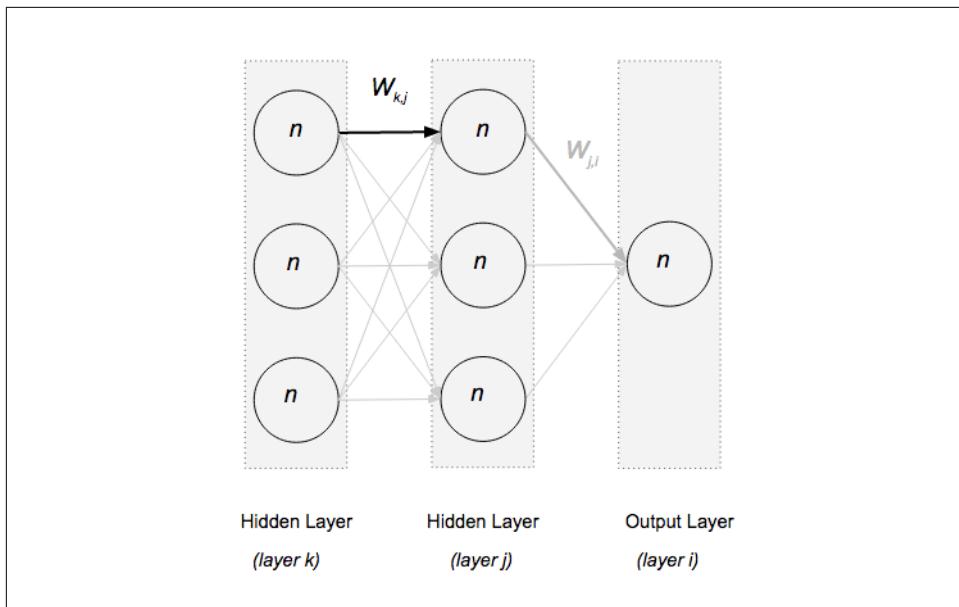


Figure 2-10. Highlighting connections between hidden layers

We then add this to the previous value for the weight, and this is our updated value:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

Weights and biases that have been assigned the blame for error are reduced to limit their signal. Weights and biases that pass along signals supporting correct answers are strengthened. The adjustment of weights to an optimal state is an iterative process, taken one step at a time.

Backpropagation and Fractional Error Responsibility

With backpropagation is how we distribute the “blame” backward through the network. Each hidden node sending input to the current node is somewhat “responsible” for some portion of the error in each of the neurons to which it has a forward connection.

The first hidden layer uses the input from the raw feature vector as input. All subsequent layers use the activation value of the previous layer’s neurons as input. However, for hidden layers before the output layer we must divide up this error appropriately. With error backpropagation, we divide the Δ_i values according to the connection weight between the hidden node and the output node.

To calculate Δ_j in the preceding equation we use the equation:

$$\Delta_j = g'(\text{input_sum}_j) \sum_i W_{j,i} \Delta_i$$

This equation takes a look at each node i in the current layer and takes the current error value Δ_i and multiplies it by the weight of the incoming connection times the derivative of the activation function. This produces the fractional error value for the node in the previous layer, which is used as we update the incoming connections to that layer. We progressively perform this algorithm layer by layer until we've updated every layer in the network.

The length of these learning steps, or the amount the weights that are changed with each iteration, is known as the *learning rate*. The learning rate is a parameter we define (as opposed to being a measurement of the network's performance). We discuss learning rate later in this chapter when we talk more about hyperparameters in general.

Backpropagation and Mini-Batch Stochastic Gradient Descent

In [Chapter 1](#), we learned about a variant of SGD called mini-batch, in which we train the model on multiple examples at once as opposed to a single example at a time. We see mini-batch used with backpropagation and SGD in neural networks as well to improve training.

Under the hood, we're computing the average of the gradient across all the the examples inside the mini-batch. Specifically, we compute the forward pass for all of the examples to get their output scores as a batch linear algebra matrix operation. During the backward pass for each layer, we are computing the average of the gradient (for the layer). By doing backpropagation this way, we're able to get a better gradient approximation and use our hardware more efficiently at the same time.



AI Winter II: Early 1990s

In the late 1980s and early 1990s, we saw an overpromotion of technologies such as expert systems and LISP machines, both of which failed to live up to expectations. The Strategic Computing Initiative cancelled new spending at the end of this cycle. The fifth-generation computer failed its goals.

Activation Functions

We use activation functions to propagate the output of one layer's nodes forward to the next layer (up to and including the output layer). Activation functions are a

scalar-to-scalar function, yielding the neuron's activation. We use activation functions for hidden neurons in a neural network to introduce nonlinearity into the network's modeling capabilities. Many activation functions belong to a logistic class of transforms that (when graphed) resemble an S. This class of function is called *sigmoidal*. The sigmoid family of functions contains several variations, one of which is known as the Sigmoid function. Let's now take a look at some useful activation functions in neural networks.

Linear

A linear transform (see [Figure 2-11](#)) is basically the identity function, and $f(x) = Wx$, where the dependent variable has a direct, proportional relationship with the independent variable. In practical terms, it means the function passes the signal through unchanged.

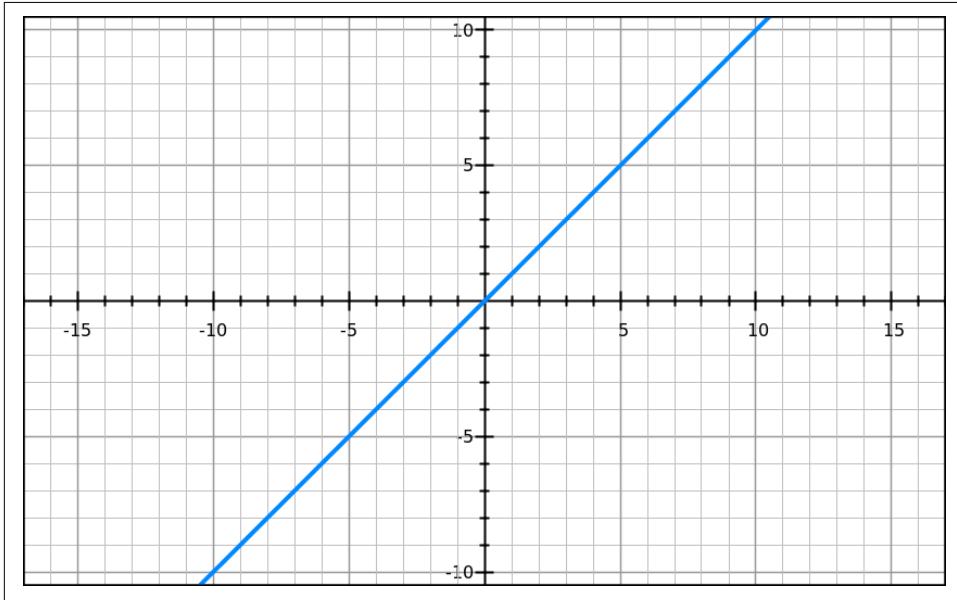


Figure 2-11. Linear activation function

We see this activation function used in the input layer of neural networks.

Sigmoid

Like all logistic transforms, sigmoids can reduce extreme values or outliers in data without removing them. The vertical line in [Figure 2-12](#) is the decision boundary.

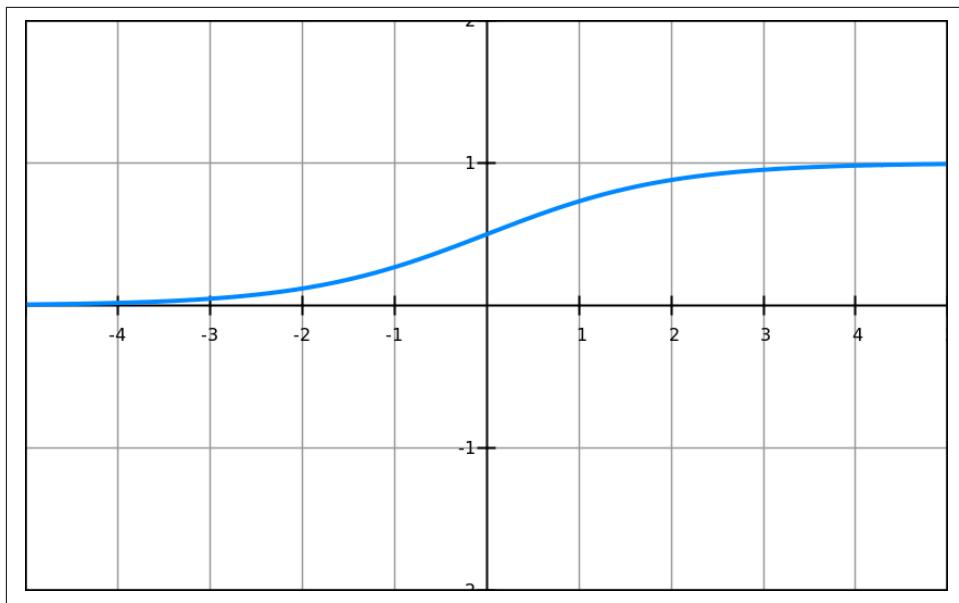


Figure 2-12. Sigmoid activation function

A sigmoid function is a machine that converts independent variables of near infinite range into simple probabilities between 0 and 1, and most of its output will be very close to 0 or 1.



Understanding Sigmoid Output

A sigmoid activation function outputs an independent probability for each class.

Tanh

Pronounced “tanch,” tanh is a hyperbolic trigonometric function (see Figure 2-13). Just as the tangent represents a ratio between the opposite and adjacent sides of a right triangle, tanh represents the ratio of the hyperbolic sine to the hyperbolic cosine: $\tanh(x) = \sinh(x) / \cosh(x)$. Unlike the Sigmoid function, the normalized range of tanh is -1 to 1. The advantage of tanh is that it can deal more easily with negative numbers.

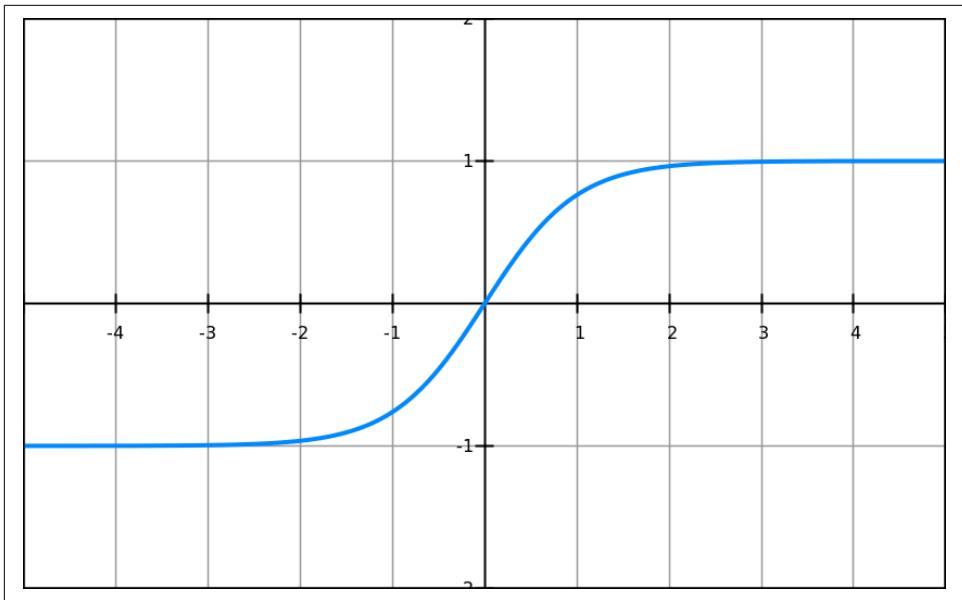


Figure 2-13. Tanh activation function

Hard Tanh

Similar to tanh, hard tanh simply applies hard caps to the normalized range. Anything more than 1 is made into 1, and anything less than -1 is made into -1. This allows for a more robust activation function that allows for a limited decision boundary.

Softmax

Softmax is a generalization of logistic regression inasmuch as it can be applied to continuous data (rather than classifying binary) and can contain multiple decision boundaries. It handles multinomial labeling systems. Softmax is the function you will often find at the output layer of a classifier.



Understanding Softmax Output

The softmax activation function returns the probability distribution over mutually exclusive output classes.

To further illustrate the idea of the softmax output layer and how to use it, let's consider two use cases. If we have a multiclass modeling problem yet we care only about

the best score across these classes, we'd use a softmax output layer with an `argmax()` function to get the highest score of all the classes.



Dealing with Multiple Classifications

If we want to get multiple classifications per output (e.g., “person + car”), we do not want softmax as an output layer. Instead, we'd use the sigmoid output layer giving us a probability for every class independently.

For the case in which we have a large set of labels (e.g., thousands of labels), we'd use the variant of the softmax activation function called the *hierarchical softmax* activation function. This variant decomposes the labels into a tree structure, and the softmax classifier is trained at each node of the tree to direct the branching for classification.

Rectified Linear

Rectified linear is a more interesting transform that activates a node only if the input is above a certain quantity. While the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable $f(x) = \max(0, x)$, as demonstrated in [Figure 2-14](#).

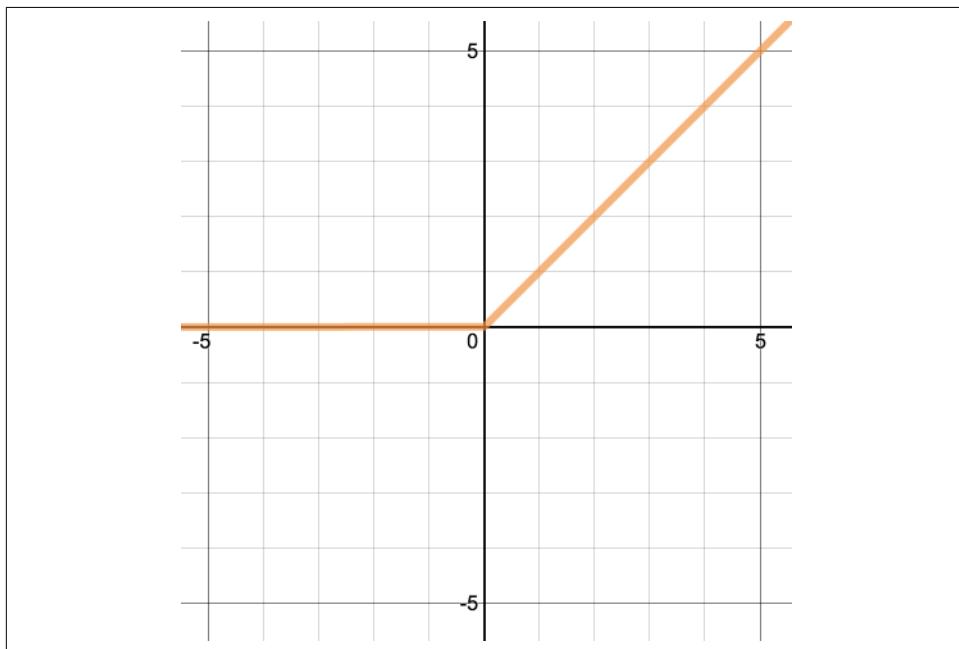


Figure 2-14. Rectified linear activation function

Rectified linear units (ReLU) are the current state of the art because they have proven to work in many different situations. Because the gradient of a ReLU is either zero or a constant, it is possible to reign in the vanishing exploding gradient issue. ReLU activation functions have shown to train better in practice than sigmoid activation functions.



The Unreasonable Effectiveness of ReLU Activation Functions

Compared to the sigmoid and tanh activation functions, the ReLU activation function does not suffer from vanishing gradient issues. If we use hard max as the activation function, we can induce sparsity in the activation output from the layer. Research has shown deep networks using ReLU activation functions to train well without using pretraining techniques.

Leaky ReLU

Leaky ReLUs are a strategy to mitigate the “dying ReLU” issue.² As opposed to having the function being zero when $x < 0$, the leaky ReLU will instead have a small negative slope (e.g., “around 0.01”). Some success has been seen in practice with this ReLU variation but results are not always consistent. The equation is given here:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Softplus

This activation function is considered to be the “smooth version of the ReLU,” as is illustrated in [Figure 2-15](#). Compare this plot to the ReLU in [Figure 2-14](#).

[Figure 2-15](#) shows that the softplus activation function ($f(x) = \ln[1 + \exp(x)]$) has a similar shape to the ReLU. We also notice the differentiability and nonzero derivative of the softplus everywhere on the graph, [in contrast to the ReLU](#).

² Li, Karpathy, “CS231n: Convolutional Neural Networks for Visual Recognition” (Course Notes). See <http://cs231n.stanford.edu> and <http://cs231n.github.io>

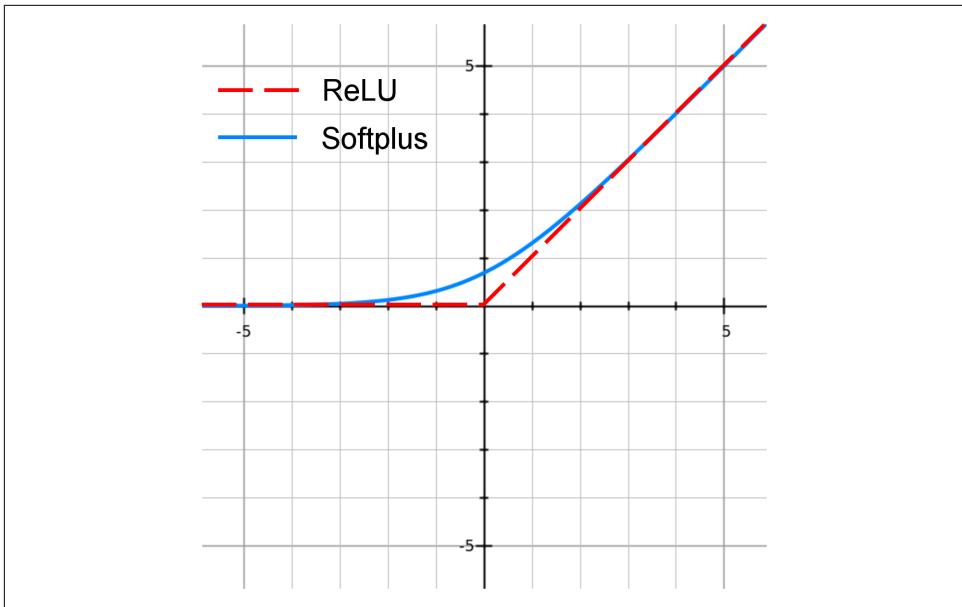


Figure 2-15. Visualizing the ReLU and softplus activation functions

Loss Functions

Loss functions quantify how close a given neural network is to the ideal toward which it is training. The idea is simple. We calculate a metric based on the error we observe in the network's predictions. We then aggregate these errors over the entire dataset and average them and now we have a single number representative of how close the neural network is to its ideal.

Looking for this ideal state is equivalent to finding the parameters (weights and biases) that will minimize the “loss” incurred from the errors. In this way, loss functions help reframe training neural networks as an optimization problem. In most cases, these parameters cannot be solved for analytically, but, more often than not, they can be approximated well with iterative optimization algorithms like gradient descent. The following section provides an overview on commonly seen loss functions, linking them back to their origins in machine learning, as necessary.

Loss Function Notation

Equations in this section will use the notation described here:

- Consider the dataset gathered to train a neural net. Let “ N ” denote the number of samples (set of inputs with corresponding outcomes) that have been gathered.

- Consider the nature of the input and output collected. Each data point records some set of unique input features and output features. Let “ P ” denote the number of input features gathered and “ M ” denote the number of output features that have been observed.
- We will use (X, Y) to denote the input and output data we collected. Note that there will be N such pairs where the input is a collection of P values and the output Y is a collection of M values. We will denote the i th pair in the dataset as X_i and Y_i .
- We will use \hat{Y} to denote the output of the neural net. Of course, \hat{Y} is the network’s guess at Y and therefore it will also have M features.
- We will use the notation $h(X_i) = \hat{Y}_i$ to denote the neural network transforming the input X_i to give the output \hat{Y}_i . We will alter this notation a little later to emphasize its dependence on weights and biases.
- When referring to j th output feature, we will use it as a subscript firmly linking our notation to a matrix where the rows are different data points and the columns are the different unique features. Thus $y_{i,j}$ refers to the j th feature observed in the i th sample collected.
- We will represent the loss function by $L(W, b)$.

Given the data available, the loss function notation indicates that its value depends only on W and b , the weights and the biases of the neural network. This cannot be emphasized enough. In the universe of a given neural network with a set number of layers, configurations, and so on that will be trained on a given set of data, the value of the loss function depends exclusively on the state of the network, as defined by the weights and biases. Wiggle those and our losses wiggle. Wiggle those for a given input and our output wiggles.

So, our notation $h(X) = \hat{Y}$ should be conditioned on a set of weights and biases; thus, we will amend our notation to say $h_{w,b}(X) = \hat{Y}$. We are now ready to tackle loss functions.

Loss Functions for Regression

In this section, we cover loss functions appropriate for regression models.

Mean squared error loss

When working on a regression model that requires a real valued output, we use the squared loss function, much like the case of ordinary least squares in linear regression. Consider the case in which we have to predict only one output feature ($M = 1$). The error in a prediction is squared and is averaged over the number of data points, plain and simple, as we see in the following equation for MSE loss:

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N (\widehat{Y}_i - Y_i)^2$$

What if M is greater than one and we are looking to predict multiple output features for a given set of input features? In this case, the desired and predicted entities, Y and \widehat{Y} , respectively, are an ordered list of numbers, or, in other words, vectors.



A Note About Loss Functions

The loss function boils-down the difference between desired and predicted, be that they are vectors, into a single number.

Let's now look at another variation of the MSE loss function:

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N \frac{1}{M} \sum_{j=1}^M (\widehat{y}_{ij} - y_{ij})^2$$

If you're familiar with linear algebra, you'll recognize the inner sigma in the preceding equation as the square of the Euclidean distance. In fact, the MSE is sometimes referred to by these terms. Note that N , the size of your dataset, and M , the number of features the network has to predict, are constants. So, consider these as simple scaling factors that you can account for in other ways (like by scaling the learning rate). In a lot of use cases (including DL4J), the M is dropped and a division by two is added for mathematical convenience (which will become clearer in the context of its gradient in backpropagation). In the following equation, we see the version of MSE that the DL4J library uses for regression:

$$L(W, b) = \frac{1}{2N} \sum_{i=1}^N \sum_{j=1}^M (\widehat{y}_{ij} - y_{ij})^2$$



Is MSE Really a Convex Loss Function?

In a technical sense, the MSE is a convex loss function. However, when dealing with hidden layers in neural networks, the convex property no longer holds true, because we could have multiple parameter sets of values resulting in the same loss value.



Optimizing MSE

Optimizing the MSE is equivalent to optimizing for the mean.

Other loss functions for regression

Although the MSE is used widely, it is quite sensitive to outliers, and this is something that you should consider when picking a loss function. When picking a stock to invest in, we want to take the outliers into account. But perhaps when buying a house we don't. In this case, what is of most interest is what most people would pay for it. In which case, we are more interested in the median and less so in the mean.

Mean absolute error loss. In a similar vein, an alternative to the MSE loss is the mean absolute error (MAE) loss, as shown in the following equation:

$$L(W, b) = \frac{1}{2N} \sum_{i=1}^N \sum_{j=1}^M |\widehat{y}_{ij} - y_{ij}|$$

This simply averages the absolute error over the entire dataset.

Mean squared log error loss. Another loss function used for regression is the mean squared log error (MSLE):

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M (\log \widehat{y}_{ij} - \log y_{ij})^2$$

Mean absolute percentage error loss. Finally, we have mean absolute percentage error (MAPE) loss:

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M \frac{100 \times |\widehat{y}_{ij} - y_{ij}|}{y_{ij}}$$

Regression loss function discussion

These are all valid choices, and there certainly is no single loss function that will outperform all other loss functions for every scenario. The MSE is very widely used and is a safe bet in most cases. So is the MAE. The MSLE and the MAPE are worth taking into consideration if our network is predicting outputs that vary largely in range. Suppose that a network is to predict two output variables: one in the range of [0, 10] and the other in the range of [0, 100]. In this case, the MAE and the MSE will penalize the error in the second output more significantly than the first. The MAPE makes it a relative error and therefore doesn't discriminate based on the range. The MSLE squishes the range of all the outputs down, simply how 10 and 100 translate to 1 and 2 (in log base 10).



Common Practice for Regression in Neural Networks

Although MSLE and MAPE are approaches to handling large ranges, common practice with neural networks is to normalize inputs to a suitable range and use the MSE or MAE to optimize for either the mean or the median.

Loss Functions for Classification

We can build neural networks to bin data points into different categories; for example, fraud|not fraud. However, when building neural networks for classification problems, the focus is often on attaching probabilities to these classifications (30 percent fraud|70 percent not fraud). These differing scenarios require different loss functions.

Hinge loss

Hinge loss is the most commonly used loss function when the network must be optimized for a hard classification. For example, 0 = no fraud and 1 = fraud, which by convention is called a 0-1 classifier. The 0,1 choice is somewhat arbitrary and -1, 1 is also seen in lieu of 0-1. Hinge loss is also seen in a class of models called maximum-margin classification models (e.g., support vector machines, a somewhat distant cousin to neural networks).

Following is the equation for hinge loss when data points must be categorized as -1 or 1:

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N \max(0, 1 - y_{ij} \times \hat{y}_{ij})$$

The hinge loss is mostly used for binary classifications. There are extensions for multiclass classification (e.g., “one versus all,” “one versus one”) for the hinge loss that are not covered here.



Hinge Loss Is a Convex Function

Note that like the MSE, the hinge loss is known to be a convex function.

Logistic loss

Logistic loss functions are used when probabilities are of greater interest than hard classifications. Great examples of these would be flagging potential fraud, with a human-in-the-loop solution or predicting the “probability of someone clicking on an ad,” which can then be linked to a currency number.

Predicting valid probabilities means generating numbers between 0 and 1. Predicting valid probabilities also means making sure the probability of mutually exclusive outcomes should sum to one. For this reason, it is essential that the very last layer of a neural network used in classification is a softmax. Note that the sigmoid activation function also will give valid values between 0 and 1. However, you cannot use it for scenarios in which the outputs are mutually exclusive, because it does not model the dependencies between the output values.

Now that we have made sure our neural network will produce valid probabilities for the classes we have, we can dive headlong into the loss function and into the idea of what we should be optimizing here. We want to optimize for what is formally called the “maximum likelihood.” In other words, we want to maximize the probability we predict for the correct class AND we want to do so for every single sample we have.

Let us consider the case in which our network predicts a probability for two classes, like the fraud and not fraud 0–1 classifier. Based on the notation described earlier, we can express our output for a given input X_i as $h(X_i)$ and $1 - h(X_i)$, given a set of weights and biases, W and b , expressing the probability of 1 and 0, respectively, as shown here:

$$P(y_i = 1 | X_i; \mathbf{W}, \mathbf{b}) = h_{\mathbf{W}, \mathbf{b}}(X_i)$$

$$P(y_i = 0 | X_i; \mathbf{W}, \mathbf{b}) = 1 - h_{\mathbf{W}, \mathbf{b}}(X_i)$$

We can combine these equations and express them as follows:

$$P(y_i | X_i; \mathbf{W}, \mathbf{b}) = (h_{\mathbf{W}, \mathbf{b}}(X_i))^{y_i} \times (1 - h_{\mathbf{W}, \mathbf{b}}(X_i))^{1 - y_i}$$

The word “AND” from our colloquial definition of maximum likelihood in the context of probability should immediately ring a bell. The “AND” across all available samples translates to the product of the probabilities in question, as shown here:

$$L(W, b) = \prod_{i=1}^N \hat{y}_i^{y_i} \times (1 - \hat{y}_i)^{1 - y_i}$$

Let’s now move on and take a look at negative log likelihood.

Negative log likelihood. For the sake of mathematical convenience, when dealing with the product of probabilities, it is customary to convert them to the log of the probabilities; hence, the product of the probabilities transforms to the sum of the log of the probabilities.



Negative Log Likelihood and Maximizing Probability

The logarithm is a monotonically increasing function. Thus, minimizing the negative log likelihood is equivalent to maximizing the probability.

We also negate the expression so that the equation now corresponds to a “loss.” So, the loss function in question becomes the following, commonly referred to as the negative log likelihood:

$$L(W, b) = - \sum_{i=1}^N y_i \times \log \hat{y}_i + (1 - y_i) \times \log (1 - \hat{y}_i)$$

Extending the loss function from two classes to M classes gives us the following equation for logistic loss:

$$L(W, b) = - \sum_{i=1}^N \sum_{j=1}^M y_{i,j} \times \log \hat{y}_{i,j}$$

This is mathematically equivalent to what is called the cross-entropy between two probability distributions—in this case, what we predict and what we have observed under the same criteria. We will dive into this a bit more in the section that follows.



Unifying Views on Loss Functions

Note that we want to minimize our loss function, which we do by maximizing the likelihood, which we in turn do by minimizing the negative log likelihood. A mouthful? Yes, indeed.

Cross-entropy has its origin in information theory, whereas negative log likelihood for classification has its origin in statistical modeling. These two methods are mathematically the same, so although it might not matter which is used, it tends to confuse people.

Loss Functions for Reconstruction

This set of loss functions relates to what is called *reconstruction*. The idea is simple. A neural network is trained to recreate its input as closely as possible. So, why is this any different from memorizing the entire dataset? The key here is to tweak the scenario so that the network is forced to learn commonalities and features across the dataset.

In one approach, the number of parameters in the network is constrained such that the network is forced to compress the data and then re-create it. Another often-used approach is to corrupt the input with meaningless “noise” and train the network to ignore the noise and learn the data. Examples of these kinds of neural nets are restricted Boltzmann machines, autoencoders, and so on. These neural networks all use loss functions that are rooted in information theory.

Following is the equation for KL divergence:

$$D_{KL}(Y || \hat{Y}) = -\sum_{i=1}^N Y_i \times \log \left(\frac{Y_i}{\hat{Y}_i} \right)$$

Although we briefly touched on the subject of cross-entropy a moment ago and justified taking the log of probabilities to turn our sum into product, we did not mention that taking the log of the probability puts us squarely in the field of informational theory and the concept of entropy.



Different Approaches

Even though the negative log likelihood as framed above is mathematically equivalent to the cross-entropy, they are, however, grounded in different theoretical approaches.

Hyperparameters

In machine learning, we have both model parameters and parameters we tune to make networks train better and faster. These tuning parameters are called *hyperparameters*, and they deal with controlling optimization functions and model selection during training with our learning algorithm. In DL4J, we also refer to the optimization algorithms as updaters, because updates are synonymous with the steps the algorithm takes across the weight space to minimize error.

Hyperparameter selection focuses on ensuring that the model neither underfits nor overfits the training dataset, while learning the structure of the data as quickly as possible.

Learning Rate

The learning rate affects the amount by which you adjust parameters during optimization in order to minimize the error of neural network's guesses. It is a coefficient that scales the size of the steps (updates) a neural network takes to its parameter vector x as it crosses the loss function space.

During backpropagation we multiply the error gradient by the learning rate, and then update a connection weight's last iteration with the product to reach a new weight. The learning rate determines how much of the gradient we want to use for the algorithm's next step. A large error and steep gradient combine with the learning rate to produce a large step. As we approach minimal error and the gradient flattens out, the step size tends to shorten.

A large learning rate coefficient (e.g., 1) will make your parameters take leaps, and small ones (e.g., 0.00001) will make it inch along slowly. Large leaps will save time initially, but they can be disastrous if they lead us to overshoot our minimum. A learning rate too large oversteps the nadir, making the algorithm bounce back and forth on either side of the minimum without ever coming to rest.

In contrast, small learning rates should lead you eventually to an error minimum (it might be a local minimum rather than a global one), but they can take a very long time and add to the burden of an already computationally intensive process. Time matters when neural network training can take weeks on large datasets. If you can't wait another week for the results, choose a moderate learning rate (e.g., 0.1) and experiment with several others in the same ballpark to get the best speed and accuracy at once. Beyond setting a static learning rate, we'll look at ways to vary the learning rate over time to get the best of both worlds later in the book.

Regularization

Regularization helps with the effects of out-of-control parameters by using different methods to minimize parameter size over time.



Controlling Overfitting in Machine Learning

Regularization's main purpose is to control overfitting in machine learning.

In mathematical notation, we see regularization represented by the coefficient lambda, controlling the trade-off between finding a good fit and keeping the value of certain feature weights low as the exponents on features increase.

Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller. Smaller-valued weights lead to simpler hypotheses, and simpler hypotheses are the most generalizable. Unregularized weights with several higher-order polynomials in the feature set tend to overfit the training set.

As the input training set size grows, the effect of regularization decreases and the parameters tend to increase in magnitude. This is appropriate, because an excess of features relative to training set examples leads to overfitting in the first place. Bigger data is the ultimate regularizer.

Momentum

Momentum helps the learning algorithm get out of spots in the search space where it would otherwise become stuck. In the errorscape, it helps the updater find the gulleys that lead toward the minima. Momentum is to the learning rate what the learning rate

is to weights, and it helps us produce better quality models. We'll see the momentum hyperparameter in action in many later chapters.

Sparsity

The sparsity hyperparameter recognizes that for some inputs only a few features are relevant. For example, let's assume that a network can classify a million images. Any one of those images will be indicated by a limited number of features. But to effectively classify millions of images a network must be able to recognize considerably more features, many of which don't appear most of the time. An example of this would be how photos of sea urchins don't contain noses and hooves. This contrasts to how in submarine images the nose and hoof features will be 0.

The features that indicate sea urchins will be few and far between, in the vastness of the neural network's layers. That's a problem, because sparse features can limit the number of nodes that activate and impede a network's ability to learn. In response to sparsity, biases force neurons to activate and the activations stay around a mean that keeps the network from becoming stuck.

Fundamentals of Deep Networks

Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!

—The Red Queen, *Through the Looking Glass*

Defining Deep Learning

In the [Chapter 2](#) we set up the foundations of machine learning and neural networks. In this chapter we'll build on these foundations to give you the core concepts of deep networks. This will help build your understanding of what is going on in different network architectures as we progress into the specific architectures in [Chapter 4](#) and then the practical examples in [Chapter 5](#). Let's begin by restating our definitions of both deep learning and deep networks.

What Is Deep Learning?

Revisiting our definition of deep learning from [Chapter 1](#), the facets that differentiate deep learning networks in general from “canonical” feed-forward multilayer networks are as follows:

- More neurons than previous networks
- More complex ways of connecting layers
- “Cambrian explosion” of computing power to train
- Automatic feature extraction

When we say “more neurons,” we mean that the neuron count has risen over the years to express more complex models. Layers also have evolved from each layer being fully connected in multilayer networks to locally connected patches of neurons between layers in Convolutional Neural Networks (CNNs) and recurrent connections

to the same neuron in Recurrent Neural Networks (in addition to the connections from the previous layer).

More connections means that our networks have more parameters to optimize, and this required the explosion in computing power that occurred over the past 20 years. All of these advances provided the foundation to build next-generation neural networks capable of extracting features for themselves in a more intelligent fashion. This allowed deep networks to model more complex problem spaces (e.g., image recognition advances) than previously possible. As industry demands are ever changing and ever reaching, the capabilities of neural networks have had to charge forward. The Red Queen¹ would have it no other way.

Defining deep networks

To further provide color to our definition of deep learning, here we define the four major architectures of deep networks:

- Unsupervised Pretrained Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

There is continuous research in the domain of neural networks, but for the purposes of this book, we'll focus on these four architectures. These architectures have evolved over the past 20 years. Let's take a quick look at some of the highlights, continuing our history lesson that began in [Chapter 1](#) on the history of feed-forward multilayer neural networks.

Deep Reinforcement Learning

Reinforcement learning is defined in [Sutton's book](#) as follows:

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem.

It goes on to state that any method suitable to solving that problem can be considered to be a reinforcement learning method. In reinforcement learning, we do not tell the learner which actions to take, but let the agent discover the actions yielding the best reward by experimenting in a simulation.

In reinforcement learning, the agent begins with no trained model of the environment, and the utility function is synonymous with the reward or objective our agent is after. The training system gives the agent input from the environment and rewards

¹ Referring to Lewis Carroll's Red Queen character from the book *Through the Looking Glass*; she has to keep running to stay in the same place.

the agent when the outcome (of the cycle, or *frame*) of the simulation (or *game*) being played is positive. Many times, actions will affect not only the immediate reward, but also future rewards. The mechanics of trial-and-error and delayed reward are key features of reinforcement learning.

Deep reinforcement learning is a variant of reinforcement learning in which a neural network is used as the universal function's approximator. The drawback to this approach is that the behavior of a neural network is not bounded, and the proof of convergence doesn't hold anymore. However, using neural networks as the universal function approximator shows good results despite this.

In 2013 a paper was published by the DeepMind team at the NIPS 2013 Deep Learning Workshop on playing ATARI games with Deep Q Learning.² In this paper, the authors used a standard algorithm (Q Learning with function approximation). Their function approximator for this algorithm was a CNN. They demonstrated an agent playing Atari 2600 games based on using the pixels on the screen as input and a CNN as the internal model.

The computer/agent playing the ATARI game was positively rewarded when the outcome of the game was positive based on the actions performed. The algorithm was able to learn some of the games to a level where it performed better than humans.

Deep reinforcement learning has risen in popularity over the course of production of this book, and hopefully we'll see it included in a future edition. For now we'll direct you to an example in [Appendix B](#).

Evolutionary progress and resurgence

When we last left off in [Chapter 2](#), neural networks had entered a “winter period” in the mid-1980s when the promise of AI fell short of what it could deliver. As happens many times when promising technology falls into the Trough of Disillusionment ([Figure 3-1](#)), there were many researchers still doing important work in the realm of neural networks.

² Mnih et al. 2013. “Human-level control through deep reinforcement learning.”

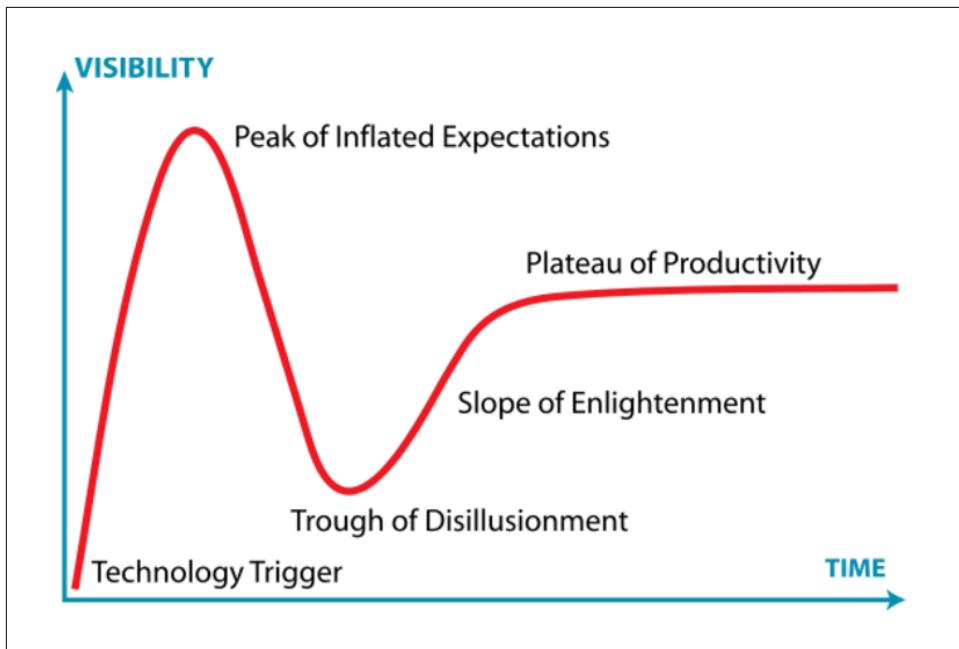


Figure 3-1. Trough of Disillusionment (source: https://en.wikipedia.org/wiki/Hype_cycle)

One important development in neural networks was Yann LeCun's work at AT&T Bell Labs on optical character recognition.³ His lab was focused on check image recognition for the financial services sector. Through this work, LeCun and his team developed the concept of the biologically inspired model of image recognition we know today as CNN. This eventually led to the creation of the **MNIST handwriting benchmark** (we cover this more later in the chapter) and a progressive number of record accuracy marks achieved by deep learning.



Better Labeled Data

Another contributing factor to the evolution and success of deep networks was the creation of better and larger labeled datasets such as MNIST and **ImageNet**.

Advances in modeling sequential data with recurrent neural networks appeared in the late-1980s and early 1990s by researchers such as Sepp Hochreiter. As time went on, the research community created better artificial neuron variants (e.g., Long Short-

³ LeCun et al. 1998. “Gradient-based learning applied to document recognition.”

Term Memory [LSTM] Memory Cell and Memory Cell with Forget Gate) over the course of the late 1990s. The stage for a neural network resurgence was being set quietly in research labs around the world.

During the 2000s, researchers and industry applications began to progressively apply these advances in products such as the following:

- Self-driving cars
- Google Translate
- Amazon Echo
- AlphaGo

Self-driving cars in the 2006 Darpa Grand Challenge used many techniques beyond just deep learning. The top teams (Stanford and Carnegie Mellon University) were able to take advantage of the big improvements of image processing.



Advances in Computer Vision

In 2012 Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton developed a “large, deep convolutional neural network” that won the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge).

AlexNet⁴ was hailed as an advancement in computer vision and some credit it specifically with kicking off the deep learning craze. However, it was largely a scaled-up (e.g., deeper and wider) variant of the CNNs from the 1990s. The recent advances in computer vision were driven less by recent algorithm advances and more by better compute, data, and infrastructure.

Better image analysis allowed the planning systems in the cars to better choose paths through uncertain terrain and avoid obstacles more safely. Other advances in deep learning allowed models to more accurately translate and recognize audio data, driving value in the Google Translate and Amazon Echo line of products. Most recently, we've seen another complex game fall at the master level when the AlphaGo system beat the 9-dan professional Go player Lee Sedol.

Big advances in what machine learning can accomplish are not always easy to see. Public recognition for these advances many times is the culmination of many different lines of work that is exhibited in high-profile demonstrations such as The Darpa Grand Challenge or Watson beating Ken Jennings in Jeopardy. However, behind the scenes, the underpinnings for these advances change slowly but constantly. Just like

⁴ Krizhevsky, Sutskever, and Hinton. 2012. “ImageNet Classification with Deep Convolutional Neural Networks.”

the changing of the seasons, we don't always notice these changes in our daily lives until they've crossed some threshold.

In the near future, we'll continue to see deep learning being applied in unique and innovative ways. This application will be more of the latent intelligence variety (e.g., recommendations or voice recognition) coupled with pragmatic engineering to make them useful in everyday aspects of our lives. What we're unlikely to see (in the near term, at least) are out of control malevolent artificial agents blowing us out of airlocks at inopportune times (think HAL 9000 from *2001: A Space Odyssey*).

HAL 9000

HAL 9000 is the fictional computer who controls the systems of the *Discovery One spaceship* in Arthur C. Clarke's book *2001: A Space Odyssey*. HAL stands for Heuristically programmed ALgorithmic computer. HAL is represented largely in the film as a camera lens with a glowing red dot and is accessed through a conversational voice recognition system. In the film, HAL concludes that it must kill off the crew of the *Discovery One* to complete its mission successfully.

Dave: Open the pod bay doors, HAL.

HAL: I'm sorry, Dave. I'm afraid I can't do that.

Deep learning continues to push the field forward in many domains and on many core machine learning problems. Here are just a few of the benchmark records deep learning has achieved in the last few years:

- Text-to-speech synthesis (Fan et al., Microsoft, Interspeech 2014)
- Language identification (Gonzalez-Dominguez et al., Google, Interspeech 2014)
- Large vocabulary speech recognition (Sak et al., Google, Interspeech 2014)
- Prosody contour prediction (Fernandez et al., IBM, Interspeech 2014)
- Medium vocabulary speech recognition (Geiger et al., Interspeech 2014)
- English-to-French translation (Sutskever et al., Google, NIPS 2014)
- Audio onset detection (Marchi et al., ICASSP 2014)
- Social signal classification (Brueckner & Schulter, ICASSP 2014)
- Arabic handwriting recognition (Bluche et al., DAS 2014)
- TIMIT phoneme recognition (Graves et al., ICASSP 2013)
- Optical character recognition (Breuel et al., ICDAR 2013)
- Image caption generation (Vinyals et al., Google, 2014)
- Video-to-textual description (Donahue et al., 2014)
- Syntactic parsing for natural language processing (Vinyals et al., Google, 2014)
- Photo-real talking heads (Soong and Wang, Microsoft, 2014)

Based on these accomplishments, we can easily project deep learning to impact many applications over the next decade. Some of the more impressive demonstrations of applied deep learning include the following:

- Automated image sharpening
- Automating image upscaling
- WaveNet: generating human speech that can imitate anyone's voice
- WaveNet: generating believable classical music
- Speech reconstruction from silent video
- Generating fonts
- Image autofill for missing regions
- Automated image captioning (see also: <https://github.com/karpathy/neuraltalk2>)
- Turning hand-drawn doodles into stylized artwork

We probably won't realize all of the major commercial applications until they're right in front of our faces. Understanding the advances in deep network architecture is important in order to understand application ideas going forward.

Advances in network architecture

As research pressed the state of the art forward from multilayer feed-forward networks toward newer architectures like CNNs and Recurrent Neural Networks, the discipline saw changes in how layers were set up, how neurons were constructed, and how we connected layers. Network architectures evolved to take advantage of specific types of input data.

Advances in layer types. Layers became more varied with the different types of architectures. Deep Belief Networks (DBNs) demonstrated success with using Restricted Boltzmann Machines (RBMs) as layers in pretraining to build features. CNNs used new and different types of activation functions in layers and changed how we connected layers (from fully connected to locally connected patches). Recurrent Neural Networks explored the use of connections that better modeled the time domain in time-series data.

Advances in neuron types. Recurrent Neural Networks specifically created advancements in the types of neurons (or units) applied in the work around LSTM networks. They introduced new units specific to Recurrent Neural Networks such as the LSTM Memory Cell and Gated Recurrent Units (GRUs).

Hybrid architectures. Continuing the theme of matching input data to architecture type, we have seen hybrid architectures emerge for types of data that has both a time domain and image data involved. For instance, classifying objects in video has been successfully demonstrated by combining layers from both CNNs and Recurrent Neural Networks into a single hybrid network. Hybrid neural network architectures can allow us to take advantage of the best of both worlds in some cases.

From feature engineering to automated feature learning

Although deep networks might have innovated with new units and layers for their internals, they are still fundamentally capped with a discriminatory classifier at the end with the constructed features as input. Automating feature extraction is a common theme among the various architectures. Each architecture does feature construction differently and is specialized such that it is better at certain types of input than others. Yann LeCun hit on this theme describing deep learning when he said it was: “machines that learn to represent the world.”

Geoffrey Hinton talks about this theme in DBNs when he explains how RBMs are used to decompose the data into higher-order features.⁵



Categorizing DBNs

For the purposes of this book, we place DBNs (and autoencoders) in the UPN group of deep networks.

Staying with the image classification theme, we can use the example of face detection. Raw image data of faces as input have issues with how the face is oriented, lighting of the photo, and position of the key features of the face. The key features we'd normally associate with a face are things like the edge of the face, edges of specific features like eyes and nose, and then subtle features we don't consistently see like dimples.

Feature engineering. Handcrafting features has been a hallmark of machine learning for a long time. Practitioners who win competitions in machine learning often study the dataset thoroughly and use many arcane tricks to make the learning process as simple as possible for their learning algorithm. These datasets are often columnar/tabular text data and we can apply domain knowledge to specific columns so feature creation is more direct.

If we think back to [Chapter 1](#) and how we modeled the input data as the A matrix in the equation $Ax = b$, we can see how we had to hand-code the values from the data into those specific columns of A . These handcrafted features tend to produce highly accurate models but take a lot of time and experience to produce. From a knowledge representation perspective, it's like reading a poorly written book versus a book that is well-written and easy to read. The former takes us a lot longer to read and we need to spend more energy to get the same out of it as the latter.

Image classification is an interesting example because handcrafting image features is more difficult than creating features for tabular data. The information in images is

⁵ Hinton, Osindero, and Teh. 2006. “[A Fast Learning Algorithm for Deep Belief Nets](#).”

not constrained to stay in the same column and can be influenced by lighting, angle, and other issues. Feature extraction and creation for images needed a new approach, which in some part drove the evolution of CNNs.

Feature learning. Coming back to our face-detection example, a nose can be located in any set of pixels in an image as opposed to our bank balance always being located in a specific column in tabular data. With CNNs, we train the network to understand the edge of the nose and then the general shape of the nose from lower-level “nose-edge” features. The first layers in the network might pick up those nose-edge features and then pass them on to later layers in the network as larger *feature maps*.

These more granular patches of feature maps eventually are combined into a “face” feature at the latter layers of the CNN. This allows a CNN to take on a task that has been attempted many times before (“Is this a face?”) yet pose the question in a simpler way that takes less energy to answer in a more accurate way.



Automated Feature Learning with Complex Data

Taking complex raw data and creating higher-order features automatically in order to make a simpler classification (or regression) output is a hallmark of deep learning.

As you progress through this book, you’ll get a better sense of how to match input data types to deep network architectures and how to set up these architectures to best model the underlying dataset.

Generative modeling

Generative modeling is not a new concept, but the level to which deep networks have taken it has begun to rival human creativity. From generating art to generating music to even writing beer reviews, we see deep learning applied in creative ways every day. Recent variants of generative modeling to note include the following:

- Inceptionism
- Modeling artistic style
- Generative Adversarial Networks
- Recurrent Neural Networks

Let’s quickly review each of these.

Inceptionism. **Inceptionism** is a technique in which a trained convolutional network is taken with its layers in reverse order and given an input image coupled with a prior constraint. The images are modified iteratively to enhance the output in a manner that could be described as “hallucinative.” In examples for which the input involves images of the sky, we might see fish faces appear in clouds of the output image. This

line of research from Google has shown discriminatory neural network models contain considerable information to generate images.

Modeling artistic style. Variants of convolutional networks have shown to learn the style of specific painters and then generate a new image in this style of arbitrary photographs. [Figure 3-2](#) (which you saw earlier in [Chapter 1](#)) shows the amazing results. Imagine having your family photo painted by Vincent van Gogh. (By the time this book is published, this will probably be a Snapchat filter, so you won't have to wait that long.)

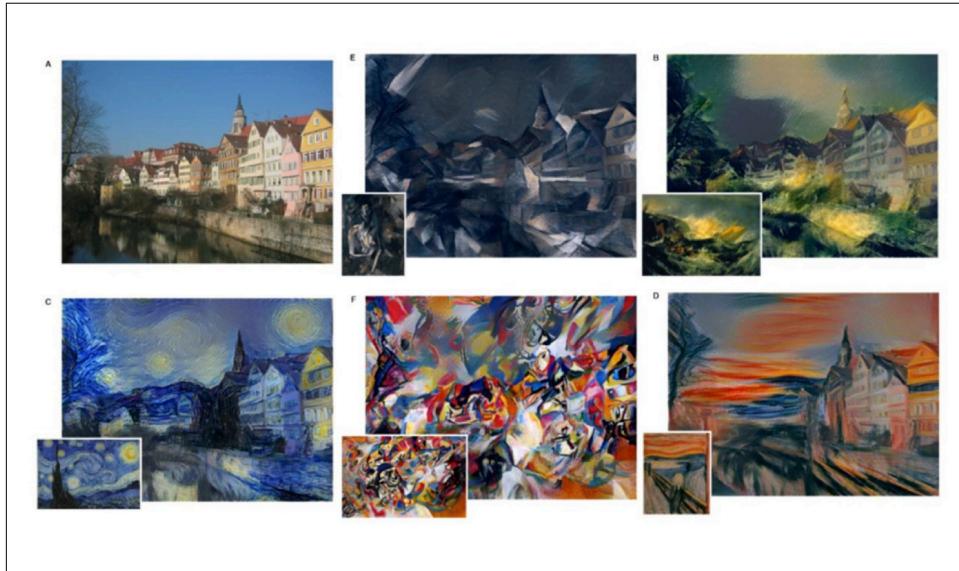


Figure 3-2. Stylized images by Gatys et al., 2015

In 2015, Gatys et al. published a paper titled “A Neural Algorithm of Artistic Style”⁶ in which they separate the style and the content of a painting. The CNN extracts the artist’s style into the network’s parameters, which can later be applied to arbitrary images to be rendered in the same style.

GANs. The generative visual output of a GAN⁷ can best be described as synthesizing novel images by modeling the distributions of input data seen by the network. We cover GANs in more depth in [Chapter 4](#).

Recurrent Neural Networks. Recurrent Neural Networks have been shown to model sequences of characters and generate new sequences that are lucidly coherent. In

⁶ Gatys et al., 2015. “[A Neural Algorithm of Artistic Style](#).”

⁷ Goodfellow et al. 2014. “[Generative Adversarial Networks](#).”

Chapter 5, we take a look at an example of Recurrent Neural Networks in which we generate new lines of Shakespeare by modeling all of Shakespeare's other works.

Another interesting application of Recurrent Neural Networks is the work by Lipton and Elkan in which the network models proper nouns like "Coors Light" and other aspects of beer jargon. The generated beer reviews can be guided with hints (e.g., "give me a 3-star review of a German lager") and are impressive. Here's a sample beer review generated by the program:

On tap at the brewpub. A nice dark red color with a nice head that left a lot of lace on the glass. Aroma is of raspberries and chocolate. Not much depth to speak of despite consisting of raspberries. The bourbon is pretty subtle as well. I really don't know that find a flavor this beer tastes like. I would prefer a little more carbonization to come through. It's pretty drinkable, but I wouldn't mind if this beer was available.⁸

The Tao of deep learning

There is a lot of marketing noise and hype in the realm of deep learning today, some of it justifiably so. However, deep learning is still trying to answer the same fundamental machine learning questions like: "Is this image a face?" The difference is that deep learning has taken the previous generation's neural network techniques and added advanced automated feature construction to make computationally difficult questions on complex data easier to answer.

When you use deep learning as a practitioner, the best way to take advantage of this power is to match the input data to the appropriate deep network architecture. If you do this, you can apply deep learning successfully in new and interesting ways. If you don't, you won't add any new modeling power beyond basic techniques like logistic regression. The remainder of this book is dedicated to giving you, as practitioner, the skills and context necessary to make these decisions and use deep learning well.

Organization of This Chapter

In this chapter, we build on the concepts from **Chapter 1** and dig further into specific architectures for deep networks. We'll differentiate the architectures and break down how their components evolved differently, providing color on how this better extracts features from certain types of data. We close the chapter with some discussion of the practicality of deep learning and alleviate some misconceptions surrounding the domain today. With that, let's continue our discussion of the architecture components relevant to deep networks.

⁸ Source: [IEEE Spectrum](#).

Common Architectural Principles of Deep Networks

Before we get into the specific architectures of the major deep networks, let's extend our understanding of the core components. First, we'll reexamine the core components again as follows and extend their coverage for the purposes of understanding deep networks:

- Parameters
- Layers
- Activation functions
- Loss functions
- Optimization methods
- Hyperparameters

Next, we'll take these concepts and build on them to better understand the building block networks of deep networks, such as the following:

- RBMs
- Autoencoders

We'll then continue to build on these ideas by reviewing these specific deep network architectures:

- UPNs
- CNNs
- Recurrent neural networks
- Recursive neural networks

As we work our way through this chapter, we'll also drop references to how DL4J implements certain aspects of deep networks. For now, let's continue our review of parameters to better understand how they are extended for deep networks.

Parameters

We learned in [Chapter 1](#) that parameters relate to the x parameter vector in the equation $Ax = b$ in basic machine learning. Parameters in neural networks relate directly to the weights on the connections in the network. In [Figure 1-4](#) (introduced first in [Chapter 1](#)) we can see the parameter vector represented by the x column vector. We take the dot product of the matrix A and the parameter vector x to get our current output column vector b . The closer our outcome vector b is to the actual values in the training data, the better our model is. We use methods of optimization such as gradient descent to find good values for the parameter vector to minimize loss across our training dataset.

In deep networks, we still have a parameter vector representing the connection in the network model we're trying to optimize. The biggest change in deep networks with respect to parameters is how the layers are connected in the different architectures. In DBNs, we see two parallel sets of feed-forward connections with two separate networks. One network's layers are composed of RBMs (subnetworks in their own right, which we'll review later in the chapter) used to extract features for the other network. The other network in a DBN is a regular feed-forward multilayer neural network, which uses the features extracted from the RBMs–layer network to initialize its weights. This is just one example of many that we'll see over the course of this chapter in how parameters/weights are specialized in different deep network architectures.



Parameters and NDArrays

In terms of working with the core linear algebra of deep networks, DL4J relies on the ND4J library to represent these linear algebra primitives. NDArrays and linear algebra are key to working with neural networks in DL4J.

Layers

In [Chapter 1](#), we learned how input, hidden, and output layers define feed-forward neural networks. In [Chapter 2](#), we further expanded this architecture with more types of layers and discussed how they relate to specific architectures of deep networks. Layers also can be represented by subnetworks in certain architectures, as well. In the previous section, we used the example of DBNs having layers composed of RBMs.

Layers are a fundamental architectural unit in deep networks. In DL4J we customize a layer by changing the type of activation function it uses (or subnetwork type in the case of RBMs). We'll also look at how you can use combinations of layers to achieve a goal (e.g., classification or regression). Finally, we'll also explore how each type of layer requires different hyperparameters (specific to the architecture) to get our network to learn initially. Further hyperparameter tuning can then be beneficial through reducing overfitting.

Activation Functions

In [Chapter 1](#), we reviewed the basic activation functions used in feed-forward neural networks. In this chapter, we begin to illustrate how activation functions are used in specific architectures to drive feature extraction. The higher-order features learned from the data in deep networks are a nonlinear transform applied to the output of the previous layer. This allows the network to learn patterns in the data within a constrained space.

Activation functions for general architecture

Depending on the activation function you pick, you will find that some objective functions are more appropriate for different kinds of data (e.g., dense versus sparse). We group these design decisions for network architecture into two main areas across all architectures:

- Hidden layers
- Output layers

Hidden layers are concerned with extracting progressively higher-order features from the raw data. Depending on the architecture we're working with, we tend to use certain subsets of layer activation functions. As we work through this chapter, we'll illustrate these patterns more across DBNs, CNNs, and Recurrent Neural Networks. In [Chapter 4](#), we take a deeper look at the impacts of different activation functions on different network architectures in the context of tuning deep networks.



A Note About Input Layers

Typically for the input layer, we want to pass on the raw input vector features so that in practice we don't express an activation function for the input layer.

Hidden layer activation functions. Commonly used functions include:

- Sigmoid
 - Tanh
 - Hard tanh
- Rectified linear unit (ReLU) (and its variants)

A more continuous distribution of input data is generally best modeled with a ReLU activation function. Optionally, we'd suggest using the tanh activation function (if the network isn't very deep) in the event that the ReLU did not achieve good results (with the caveat that there could be other hyperparameter-related issues with the network).



Sigmoid Activation Functions in Practice

In recent years, we've seen the sigmoid activation function fall out of favor for hidden layers in practice and research.

As we progress further in this book, we'll see how these activation functions tend to be arranged in the different architectures.



The Evolution of Activation Functions in Practice

We're also seeing an entire family of ReLUs emerge in deep learning research, such as the "leaky ReLU." You can read more on this topic in [Chapter 6](#).

Output layer for regression. This design decision is motivated by what type of answer we expect our model to output. If we want to output a single real-valued number from our model, we'll want to use a linear activation function.

Output layer for binary classification. In this case, we'd use a sigmoid output layer with a single neuron to give us a real value in the range of 0.0 to 1.0 (excluding those values) for the single class. This real-valued output is typically interpreted as a probability distribution.

Output layer for multiclass classification. If we have a multiclass modeling problem yet we only care about the best score across these classes, we'd use a softmax output layer with an `arg-max()` function to get the highest score of all the classes. The softmax output layer gives us a probability distribution over all the classes.



Getting Multiple Classifications

If we want to get multiple classifications per output (e.g., person + car), we do not want softmax as an output layer. Instead, we'd use the sigmoid output layer with n number of neurons, giving us a probability distribution (0.0 to 1.0) for every class independently.

Loss Functions

In [Chapter 2](#) we introduced loss functions and their role in machine learning. Loss functions quantify the agreement between the predicted output (or label) and the ground truth output. We use loss functions to determine the penalty for an incorrect classification of an input vector. So far, we've introduced the following loss functions:

- Squared loss
- Logistic loss
- Hinge loss
- Negative log likelihood

Previously we described loss functions as falling into one of three camps:

- Regression
- Classification
- Reconstruction

We covered the first two types in [Chapter 1](#). The third, reconstruction, is involved in unsupervised feature extraction and is an important reason why deep learning networks have achieved their record-breaking accuracy. In certain architectures of deep networks reconstruction loss functions help the network extract features more effectively when paired with the appropriate activation function. An example of this would be using the multiclass cross-entropy as a loss function in a layer with a softmax activation function for classification output. We cover a specialized loss function in the next section.

Reconstruction cross-entropy

With the reconstruction entropy loss function, we first apply “Gaussian noise” (a kind of statistical white noise), and then the loss function punishes the network for any result that is less similar to the original input data. This feedback drives the network to learn different features in an attempt to reconstruct the input more effectively and minimize error. In deep learning, reconstruction entropy loss is used for feature engineering in the pretrain phase that involves RBMs.

Optimization Algorithms

~~Training a model in machine learning involves finding the best set of values for the parameter vector of the model. We can think of machine learning as an optimization problem in which we minimize the loss function with respect to the parameters of our prediction function (based on our model).~~



Defining “Best” in Terms of the Loss Function

In optimization algorithms, we define “best set of values” for the parameter vector as the values with the lowest loss function value.

In [Chapter 1](#), we introduced the basic concepts of optimization, gradient descent, and parameter vectors. In this section, we take a look at more advanced methods of optimization and how can we use them to train deep networks. In this book, we divide optimization algorithms into two camps:

- First-order
- Second-order

First-order optimization algorithms calculate the *Jacobian* matrix.



The Jacobian

The Jacobian is a matrix of partial derivatives of loss function values with respect to each parameter.

The Jacobian has one partial derivative per parameter (to calculate partial derivatives, all other variables are momentarily treated as constants). The algorithm then takes one step in the direction specified by the Jacobian.

Second-order algorithms calculate the derivative of the Jacobian (i.e., the derivative of a matrix of derivatives) by approximating the *Hessian*. Second-order methods take into account interdependencies between parameters when choosing how much to modify each parameter.



Second-Order Methods

Second-order methods can take “better” steps; however, each step will take longer to calculate.



Practical Use of Optimization Algorithms

We provide much of the details on optimization algorithms so that you are aware of the mechanics involved for reference. In later chapters in this book, we simplify the discussion around optimization algorithms to give you a rule of thumb for when to use which algorithm and in what context.



Other Optimization Algorithms

There are other variations of optimization algorithms (such as “meta heuristics”) that we won’t cover in this book. They include the following:

- Genetic algorithms
- Particle swarm optimization
- Ant colony optimization
- Simulated annealing

First-order methods

The Jacobian, as mentioned, is a matrix of partial derivatives of the loss function with respect to the parameters in the network. In practice, we calculate it at a specific point —the current values of the parameters.

If we think about taking one step at a time to reach an objective, first-order methods calculate a gradient (Jacobian) at each step to determine which direction to go in next. This means that at each iteration, or step, we are trying to find the next best possible direction to go, as defined by our objective function. This is why we consider optimization algorithms to be a “search.” They are finding a path toward minimal error.

Gradient descent is a member of this path-finding class of algorithms. Variations of gradient descent exist, but at its core, it finds the next step in the right direction with respect to an objective at each iteration. Those steps move us toward a global minimum error or maximum likelihood.

Stochastic gradient descent (SGD) is machine learning’s workhorse optimization algorithm. SGD trains several orders of magnitude faster than methods such as batch gradient decent, with no loss of model accuracy.



Why Is SGD Considered “Stochastic”?

This is due to how we calculate the gradient for a single input training example (or mini-batch of training examples). The computed gradient is a “noisy” approximation of the true gradient yet allows SGD to converge faster.

The strengths of SGD are easy implementation and the quick processing of large datasets. You can adjust SGD by adapting the learning rate (e.g., with methods such as Adagrad, discussed shortly) or using second-order information (i.e., the Hessian), as we’ll see next. SGD is also a popular algorithm for training neural networks due to its robustness in the face of noisy updates. That is, it helps you build models that generalize well.



Other Factors in Learning Rate Adjustment

It’s relevant to note that other techniques such as momentum and RMSProp can affect learning rates.

Second-order methods

All second-order methods calculate or approximate the Hessian. As described earlier, we can think of the Hessian as the derivative of the Jacobian. That is, it is a matrix of second-order partial derivatives, analogous to “tracking acceleration rather than speed.” The Hessian’s job is to describe the curvature of each point of the Jacobian. Second-order methods include:

- Limited-memory BFGS (L-BFGS)⁹
- Conjugate gradient¹⁰
- Hessian-free¹¹

Think of these optimization algorithms as a black-box search algorithm that determines the best way to minimize error, given an objective and a defined gradient relative to each layer.



Making Trade-offs in Optimization

A major difference in first- and second-order methods is that second-order methods converge in fewer steps yet take more computation per step.

L-BFGS. L-BFGS is an optimization algorithm and a so-called quasi-Newton method. As its name indicates, it's a variation of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, and it limits how much gradient is stored in memory. By this, we mean the algorithm does not compute the full Hessian matrix, which is more computationally expensive.

L-BFGS approximates the inverse Hessian matrix to direct weight adjustments search toward more promising areas of parameter space. Whereas BFGS stores the gradient's full $n \times n$ inverse matrix, Hessian L-BFGS stores only a few vectors that represent a local approximation of it. L-BFGS performs faster because it uses approximated second-order information. L-BFGS and conjugate gradient in practice can be faster and more stable than SGD methods.



L-BFGS in Practice

Although L-BFGS has some interesting properties, they are not commonly used in practice for deep networks.

Conjugate gradient. Conjugate gradient guides the direction of the line search process based on conjugacy information. Conjugate gradient methods focus on minimizing the conjugate L2 norm. Conjugate gradient is very similar to gradient descent in that it performs line search. The major difference is that conjugate gradient requires each

⁹ Le et al. 2011. “On Optimization Methods for Deep Learning.”

¹⁰ LeCun et al. 1998. “Efficient BackProp.”

¹¹ Martens. 2010. “Deep learning via Hessian-free optimization.”

successive step in the line search process to be conjugate to one another with respect to direction.

Hessian-free. Hessian-free optimization is related to Newton's method, but it better minimizes the quadratic function we get. It is a powerful optimization method adapted to neural networks by James Martens in 2010. We find the minimum of the quadratic function with an iterative method called conjugate gradient.

Hyperparameters

Here we define a hyperparameter as any configuration setting that is free to be chosen by the user that might affect performance.

Hyperparameters fall into several categories:

- Layer size
- Magnitude (momentum, learning rate)
- Regularization (dropout, drop connect, L1, L2)
- Activations (and activation function families)
- Weight initialization strategy
- Loss functions
- Settings for epochs during training (mini-batch size)
- Normalization scheme for input data (vectorization)

In this section, we expand on the concepts from [Chapter 1](#) with some new hyperparameters relevant to deep learning training.



A Few Cautionary Notes About Hyperparameters

Some hyperparameters apply only some of the time. We cover the details of this more in Chapters [6](#) and [7](#). Moreover, changing a specific hyperparameter might affect the best settings for other hyperparameters. We'd also like to point out that some hyperparameters are incompatible with one another (e.g., Adagrad + momentum).

Layer size

Layer size is defined by the number of neurons in a given layer. Input and output layers are relatively easy to figure out because they correspond directly to how our modeling problem handles input and output. For the input layer, this will match up to the number of features in the input vector. For the output layer, this will either be a single output neuron or a number of neurons matching the number of classes we are trying to predict.

Deciding on neuron counts for each hidden layer is where hyperparameter tuning becomes a challenge. We can use an arbitrary number of neurons to define a layer

and there are no rules about how big or small this number can be. However, how complex of a problem we can model is directly correlated to how many neurons are in the hidden layers of our networks. This might push you to begin with a large number of neurons from the start but these neurons come with a cost.

Depending on the deep network architecture, the connection schema between layers can vary. However, the weights on the connections, as we saw in [Chapter 1](#), are the parameters we must train. As we include more parameters in our model, we increase the amount of effort needed to train the network. Large parameter counts can lead to long training times and models that struggle to find convergence.



Large Parameter Counts and Overfitting

There are also cases in which a larger model will sometimes converge easier because it will simply “memorize” the training data. In [Chapter 6](#), we further discuss ways to deal with overfitting.

In [Chapter 6](#), we look at heuristics for determining neuron count per layer and how to iteratively find good layer hyperparameters.

Magnitude hyperparameters

Hyperparameters in the magnitude group involve the gradient, step size, and momentum.

Learning rate. The learning rate in machine learning is how fast we change the parameter vector as we move through search space. If the learning rate becomes too high, we can move toward our goal faster (least amount of error for the function being evaluated), but we might also take a step so large that we shoot right past the best answer to the problem, as well.



High Learning Rates and Stability

Another side effect of learning rates that are large is that we run the risk of having unstable training that does not converge over time.

If we make our learning rate too small, it might take a lot longer than we'd like for our training process to complete. A low learning rate can make our learning algorithm inefficient. Learning rates are tricky because they end up being specific to the dataset and even to other hyperparameters. This creates a lot of overhead for finding the right setting for hyperparameters.

We also can schedule learning rates to decrease over time according to some rule. We'll learn more about this in Chapters [6](#) and [7](#).



The Importance of Learning Rate as a Hyperparameter

Learning rate is considered one of the key hyperparameters in neural networks.

Nesterov's momentum. The “vanilla” version of SGD uses gradient directly, and this can be problematic because gradient can be nearly zero for any parameter. This causes SGD to take tiny steps in some cases, and steps that are too big for situations in which the gradient is too large. To alleviate these issues, we can use techniques such as the following:

- Nesterov’s momentum
- RMSProp
- Adam
- AdaDelta



DL4J and Updaters

Nesterov’s momentum, RMSProp, Adam, and AdaDelta are known as “updaters” in the terminology of DL4J. Most of the terms used in this book are universal for most of all deep learning literature; we just wanted to note this variation for DL4J specifically.

We can speed up our training by increasing momentum, but we might lower the chance that the model will reach minimal error by overshooting the optimal parameter values. Momentum is a factor between 0.0 and 1.0 that is applied to the change rate of the weights over time. Typically, we see the value for momentum between 0.9 and 0.99.

AdaGrad. AdaGrad¹² is one technique that has been developed to help augment finding the “right” learning rate. AdaGrad is named in reference to how it “adaptively” uses **subgradient methods to dynamically control** the learning rate of an optimization algorithm. AdaGrad is monotonically decreasing and never increases the learning rate above whatever the base learning rate was set at initially.

AdaGrad is the square root of the sum of squares of the history of gradient computations. AdaGrad speeds our training in the beginning and slows it appropriately toward convergence, allowing for a smoother training process.

¹² Duchi, Hazan, and Singer. 2011. [“Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.”](#)

RMSProp. RMSprop is a very effective, but currently unpublished adaptive learning rate method. Amusingly, everyone who uses this method in their work currently cites [slide 29 of Lecture 6 of Geoff Hinton's Coursera class](#).

AdaDelta. AdaDelta¹³ is a variant of AdaGrad that keeps only the most recent history rather than accumulating it like AdaGrad does.

ADAM. ADAM (a more recently developed updating technique from the University of Toronto) derives learning rates from estimates of first and second moments of the gradients.

Regularization

Let's dig deeper into the idea of regularization that we touched on in [Chapter 2](#). Regularization is a measure taken against overfitting. Overfitting occurs when a model describes the training set but cannot generalize well over new inputs. Overfitted models have no predictive capacity for data that they haven't seen. Geoffery Hinton described the best way to build a neural network model:

Cause it to overfit, and then regularize it to death.

Regularization for hyperparameters helps modify the gradient so that it doesn't step in directions that lead it to overfit. Regularization includes the following:

- Dropout
- DropConnect
- L1 penalty
- L2 penalty

Dropout and DropConnect mute parts of the input to each layer, such that the neural network learns other portions. Zeroing-out parts of the data causes a neural network to learn more general representations. Regularization works by adding an extra term to the normal gradient computed.

Dropout. Dropout¹⁴ is a mechanism used to improve the training of neural networks by omitting a hidden unit. It also speeds training. Dropout is driven by randomly dropping a neuron so that it will not contribute to the forward pass and backpropagation.

¹³ Zeiler. 2012. “ADADELTA: An Adaptive Learning Rate Method.”

¹⁴ Bengio et al. 2015. *Deep Learning* (In Preparation).



Dropout Related to Model Averaging

We also can relate dropout to the concept of averaging the output of multiple models. If we use a dropout coefficient of 0.5, we have the mean of the model. A random Dropout of features is a sampling from 2^N possible architectures, where N is the number of parameters.

DropConnect. DropConnect¹⁵ does the same thing as Dropout, but instead of choosing a hidden unit, it mutes the connection between two neurons.

L1. The penalty methods L1 and L2, in contrast, are a way of preventing the neural network parameter space from getting too big in one direction. They make large weights smaller.

L1 regularization is considered computationally inefficient in the nonsparse case, has sparse outputs, and includes built-in feature selection. L1 regularization multiplies the absolute value of weights rather than their squares. This function drives many weights to zero while allowing a few to grow large, making it easier to interpret the weights.

L2. In contrast, L2 regularization is computationally efficient due to it having analytical solutions and nonsparse outputs, but it does not do feature selection automatically for us. The “L2” regularization function, a common and simple hyperparameter, adds a term to the objective function that decreases the squared weights. You multiply half the sum of the squared weights by a coefficient called the weight-cost. L2 improves generalization, smooths the output of the model as input changes, and helps the network ignore weights it does not use.

Mini-batching

With mini-batching,¹⁶ we send more than one input vector (a group or batch of vectors) to be trained in the learning system. This allows us to use hardware and resources more efficiently at the computer-architecture level. This method also allows us to compute certain linear algebra operations (specifically matrix-to-matrix multiplications) in a vectorized fashion. In this scenario we also have the option of sending the vectorized computations to GPUs if they are present.

¹⁵ *Ibid.*

¹⁶ Bengio. 2012. “Practical recommendations for gradient-based training of deep architectures.”

Summary

In [Chapter 2](#), we learned about some of the basic regularization tools for feed-forward multilayer neural networks. In this chapter, we expanded this definition to some newer techniques and options for hyperparameters to find better parameter vectors. Let's now put some of these ideas together to construct building blocks for deep networks.

Building Blocks of Deep Networks

Building deep networks goes beyond basic feed-forward multilayer neural networks. In some cases, deep networks combine smaller networks as building blocks into larger networks; in other cases, they use a specialized set of layers. Here are the specific building blocks we want to highlight:

- Feed-forward multilayer neural networks
- RBMs
- Autoencoders

In [Chapter 2](#), we introduced the canonical feed-forward networks. Inspired by networks of biological neurons, feed-forward networks are the simplest artificial neural networks. They are composed of an input layer, one or many hidden layers, and an output layer. In this section, we introduce networks that are considered building blocks of larger deep networks:

- RBMs
- Autoencoders

Both RBMs and autoencoders are characterized by an extra layer-wise step for training. They are often used for the pretraining phase in other larger deep networks.



Unsupervised Layer-Wise Pretraining

Unsupervised layer-wise pretraining¹⁷ can help in some training circumstances. Over time, better optimization methods, activation functions, and weight initialization methods have lessened the importance of pretraining-based deep networks. A case for which pretraining becomes interesting is when we have a lot of unlabeled data yet only a relatively smaller set of labeled training data. Pre-training does, however, add extra overhead regarding tuning and extra training time.

Layer-wise pretraining works by performing unsupervised pre-training of the first layer (e.g., RBMs) based on the input data. This gives us the first layer of weights for our main neural network (e.g., feed-forward multilayer perceptron). We perform this process for each layer progressively in the network, using the output of previous layers based on the training input as the input to the successive layers. This pretraining process allows us to initialize the main neural network's parameters with good initial values.

RBM^s model probability and are great at feature extraction. They are feed-forward networks in which data is fed through them in one direction with two biases rather than one bias as in traditional backpropagation feed-forward networks.

Autoencoders are a variant of feed-forward neural networks that have an extra bias for calculating the error of reconstructing the original input. After training, autoencoders are then used as a normal feed-forward neural network for activations. This is an unsupervised form of feature extraction because the neural network uses only the original input for learning weights rather than backpropagation, which has labels. Deep networks can use either RBMs or autoencoders as building blocks for larger networks (it is, however, rare that a single network would use both). In the following sections, we take a closer look at both networks.

RBM^s

RBM^s are used in deep learning for the following:

- Feature extraction
- Dimensionality reduction

The “restricted” part of the name “Restricted Boltzmann Machines” means that connections between nodes of the same layer are prohibited (e.g., there are no visible-visible or hidden-hidden connections along which signal passes). Geoff Hinton, the

¹⁷ Bengio et al. 2007. “Greedy Layer-Wise Training of Deep Networks.”

deep learning pioneer who popularized RBM use almost a decade ago, describes the more general Boltzmann machine as follows:

A network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off.

RBMs are also a type of autoencoder, which we'll talk about in a following section. RBMs are used for pretraining layers in larger networks such as Deep Belief Networks.

Geoff Hinton

Geoff Hinton is a distinguished researcher at Google (part-time) and also works for the University of Toronto as a **Distinguished Emeritus Professor**. Dr. Hinton's team is credited for key work on RBMs and Deep Belief Networks.

Dr. Hinton's team has produced results that in no small part helped drive the wide-scale interest in the field of deep learning we see today. In 2012, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton developed a “large, deep convolutional neural network” that won the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) known as “AlexNet.” AlexNet¹⁸ was hailed as an advancement in computer vision, and some point to it specifically as the beginning of the deep learning craze.

Dr. Hinton has been a proponent of basic research and the long-term persistence it takes to get results:

...it took 17 years after Terry Sejnowski and I invented the Boltzmann machine learning algorithm before I found a version of it that worked efficiently. If you really believe in an idea, you just have to keep trying.

Network layout

There are five main parts of a basic RBM:

- Visible units
- Hidden units
- Weights
- Visible bias units
- Hidden bias units

A standard RBM has a visible layer and a hidden layer, as shown in [Figure 3-3](#). We can also see a graph of weights (connections) between the hidden and visible units in

¹⁸ Krizhevsky, Sutskever, and Hinton. 2012. “ImageNet Classification with Deep Convolutional Neural Networks.”

the figure. Think of these weights in the same way you think of weights in the classical neural network sense.

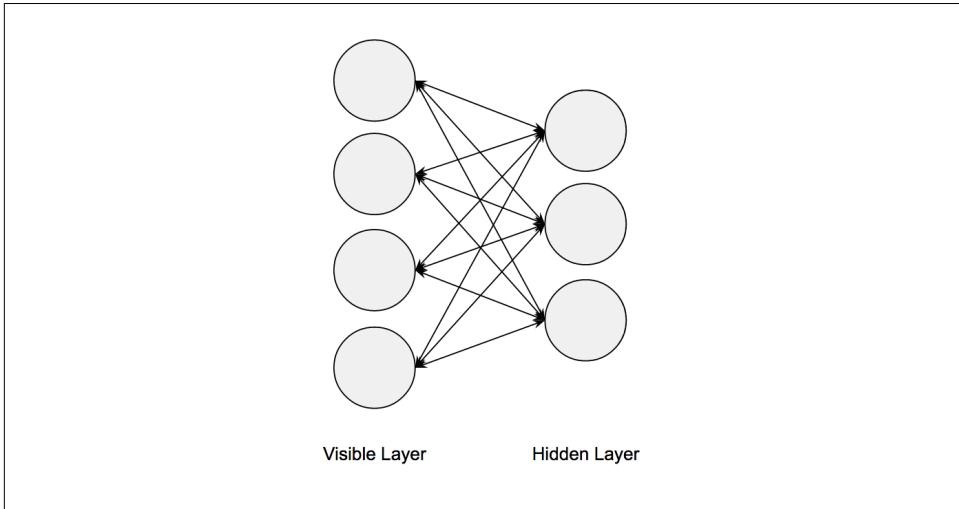


Figure 3-3. RBM network

With RBMs, every visible unit is connected to every hidden unit, yet no units from the same layer are connected. Each layer of an RBM can be imagined as a row of nodes. The nodes of the visible and hidden layers are connected by connections with associated weights.

Visible and hidden layers. In an RBM, every single node of the input (visible) layer is connected by weights to every single node of the hidden layer, but no two nodes of the same layer are connected. The second layer is known as the “hidden” layer. Hidden units are *feature detectors*, learning features from the input data. Nodes in each layer are biologically inspired as with the feed-forward multilayer neural network from [Chapter 1](#). Units (nodes) in the visible layer are “observable” in that they take training vectors as input. Each layer has a bias unit with its state always set to on.

Each node performs computation based on the input to the node and outputs a result based on a stochastic decision whether or not to transmit data through an activation. Just as with the artificial neuron from [Chapter 1](#), the activation computation is based on weights on the connections and the input values. The initial weights are randomly generated.

Connections and weights. All connections are visible-hidden; none are visible-visible or hidden-hidden. The edges represent connections along which signals are passed. Loosely speaking, those circles, or nodes, act like human neurons. They are decision units. They make decisions about whether to be on or off through acts of computa-

tion. “On” means that they pass a signal further through the net; “off” means that they don’t.

Usually, being “on” means the data passing through the node is valuable; it contains information that will help the network make a decision. Being “off” means the network thinks that particular input is irrelevant noise. A network comes to know which features/signals correlate with which labels (which code contains which messages) by being trained. With training, networks learn to make accurate classifications of the input they receive.

Biases. There is a set of bias weights (“parameters”) connecting the bias unit for each layer to every unit in the layer. Bias nodes help the network better triage and model cases in which an input node is always on or always off.

Training

The technique known as *pretraining* using RBMs means teaching it to reconstruct the original data from a limited sample of that data. That is, given a chin, a trained network could approximate (or “reconstruct”) a face. RBMs learn to reconstruct the input dataset. We’ll review the concept of reconstruction in the next section.



Contrastive Divergence

RBM^s calculate gradients by using an algorithm called **contrastive divergence**. Contrastive divergence is the name of the algorithm used in sampling for the layer-wise pretraining of a RBM. Also called CD-k, contrastive divergence minimizes the KL divergence (the delta between the real distribution of the data and the guess) by sampling k steps of a Markov chain to compute a guess.

Reconstruction

Deep neural networks with unsupervised pretraining (RBMs, autoencoders) perform feature engineering from unlabeled data through reconstruction. In pretraining, the weights learned through unsupervised pretrain learning are used for weight initialization in networks such as Deep Belief Networks.



Reconstruction as Matrix Factorization

Reconstruction is a matrix factorization problem (also known as matrix decomposition).

Figure 3-4 presents a visual explanation of the network involved in reconstruction in RBMs.

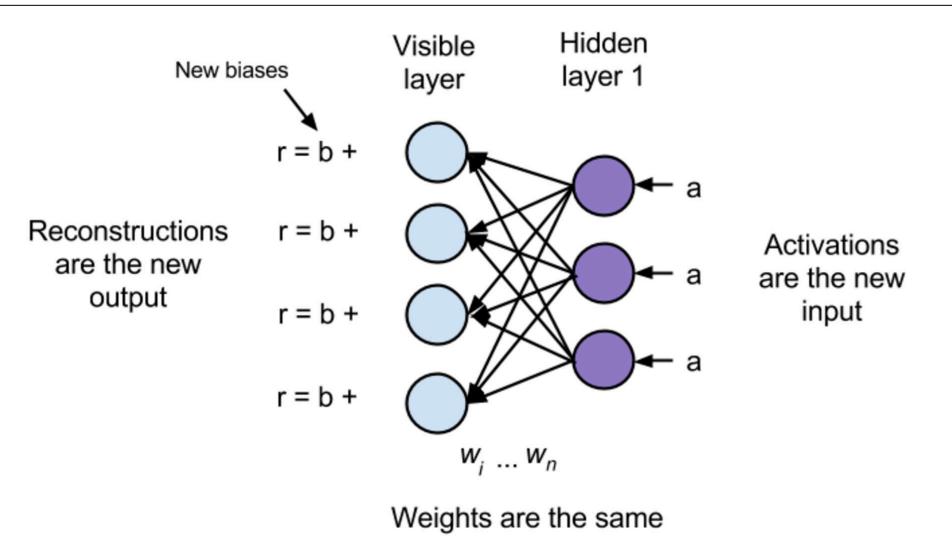


Figure 3-4. Reconstruction in RBMs

We can visually explain reconstruction in RBMs by looking at the [MNIST dataset](#). MNIST stands for the *mixed National Institute of Standards and Technology* dataset that contains the images. The MNIST dataset is a collection of images representing the handwritten numerals 0 through 9. [Figure 3-5](#) depicts a sample of some of the handwritten digits in MNIST.



Figure 3-5. Sample of MNIST digits

The training dataset in MNIST has 60,000 records and the test dataset has 10,000 records. If we use a RBM to learn the MNIST dataset, we can sample from the trained

network to see¹⁹ how well it can reconstruct the digits. Figure 3-6 shows renderings of MNIST digits being progressively reconstructed with a RBM.

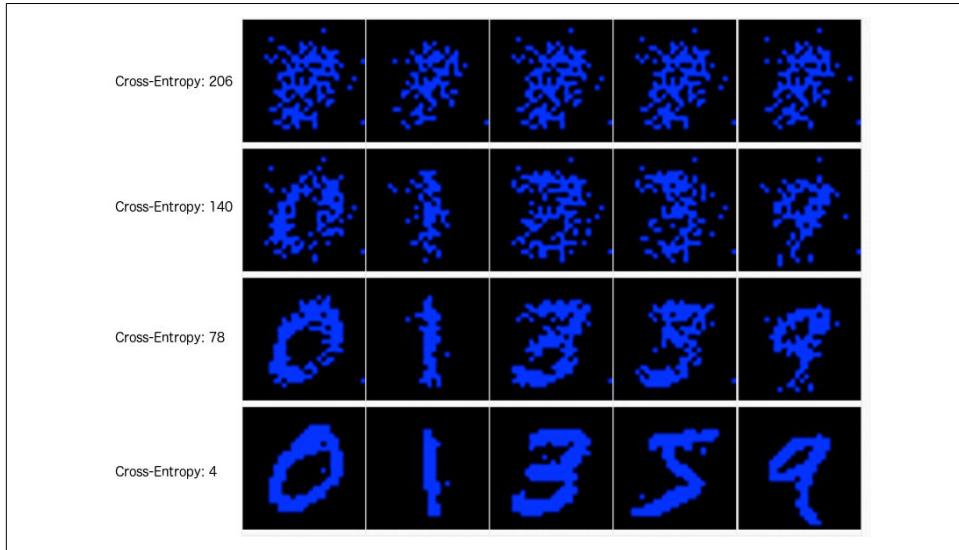


Figure 3-6. Reconstructing MNIST digits with RBMs

If the training data has a normal distribution, most of them cluster around a central *mean*, or average, and become scarcer the further you stray from that average. It looks like a bell curve. If we know the mean and the variance, or sigma, of normal data, we can reconstruct that curve. But suppose that we don't know the mean and variance. Those are parameters we then need to guess. Picking them randomly and contrasting the curve they produce with the original data can operate similarly to a loss function. We measure the difference between two probability distributions much like we measure erroneous classifications, adjust our parameters, and try again.



Reconstruction Cross-Entropy

The objective function here is usually reconstruction cross-entropy, or KL divergence (the mathematicians and cryptanalysts Solomon Kullback and Richard Leibler first published a paper on the technique in 1951). “Cross” refers to the comparison between two distributions. “Entropy” is a term from information theory that refers to uncertainty. For example, a normal curve with a wide spread, or variance, also implies more uncertainty about where data points will fall. That uncertainty is called entropy.

¹⁹ Yosinski and Lipson. 2012. “Visually Debugging Restricted Boltzmann Machine Training with a 3D Example.”

Other uses of RBMs

Here are some other places we see RBMs used:

- Dimensionality reduction
- Classification
- Regression
- Collaborative filtering
- Topic modeling

Autoencoders

We use autoencoders to learn compressed representations of datasets. Typically, we use them to reduce a dataset's dimensionality. The output of the autoencoder network is a reconstruction of the input data in the most efficient form.

Similarities to multilayer perceptrons

Autoencoders share a strong resemblance with multilayer perceptron neural networks in that they have an input layer, hidden layers of neurons, and then an output layer. The key difference to note between a multilayer perceptron network diagram (from earlier chapters) and an autoencoder diagram is the output layer in an autoencoder has the same number of units as the input layer does.

Figure 3-7 presents an example of an autoencoder network.

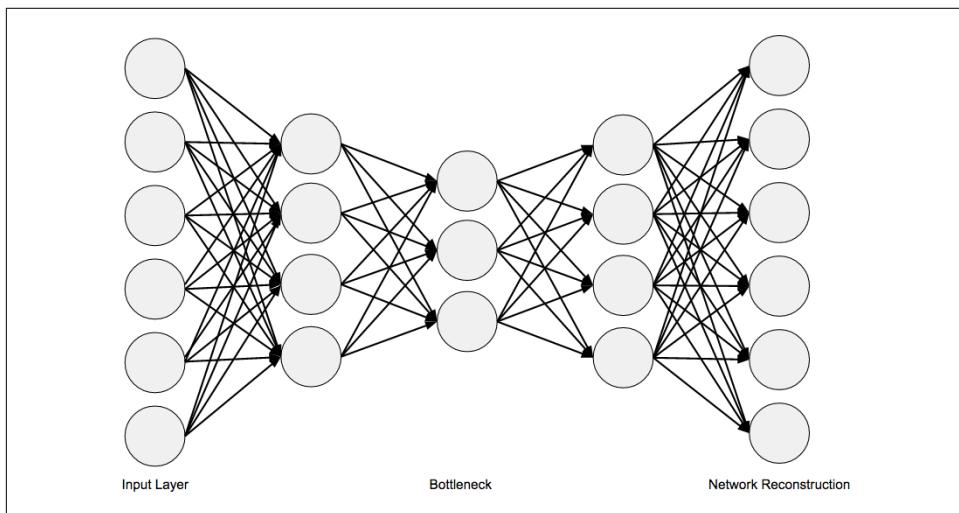


Figure 3-7. Autoencoder network architecture

Beyond the output layer, there are a few other differences, which we outline in the next section.

Defining features of autoencoders

Autoencoders differ from multilayer perceptrons in a couple of ways:

- They use unlabeled data in unsupervised learning.
- They build a compressed representation of the input data.

Unsupervised learning of unlabeled data. The autoencoder learns directly from unlabeled data. This is connected to the second major difference between multilayer perceptrons and autoencoders.

Learning to reproduce the input data. The goal of a multilayer perceptron network is to generate predictions over a class (e.g., fraud versus not fraud). An autoencoder is trained to reproduce its own input data.

Training autoencoders

Autoencoders rely on backpropagation to update their weights. The main difference between RBMs and the more general class of autoencoders is in how they calculate the gradients.

Common variants of autoencoders

Two important variants of autoencoders to note are *compression autoencoders* and *denoising autoencoders*.

Compression autoencoders. This is the architecture depicted in [Figure 3-7](#). The network input must pass through a bottleneck region of the network before being expanded back into the output representation.

Denoising autoencoders. The denoising autoencoder²⁰ is the scenario in which the autoencoder is given a corrupted version (e.g., some features are removed randomly) of the input and the network is forced to learn the uncorrupted output.

Applications of autoencoders

Building a model to represent the input dataset might not sound useful on the surface. However, we're less interested in the output itself and more interested in the difference between the input and output representations. If we can train a neural

²⁰ Vincent et al. 2010. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion.”

network to learn data it commonly “sees,” then this network can also let us know when it’s “seeing” data that is unusual, or anomalous.



Autoencoders as Anomaly Detectors

Autoencoders are commonly used in systems in which we know what the normal data will look like, yet it's difficult to describe what is anomalous. Autoencoders are good at powering anomaly detection systems.

Variational Autoencoders

A more recent type of autoencoder model is the variational autoencoder (VAE) introduced by Kingma and Welling²¹ (see Figure 3-8). The VAE is similar to compression and denoising autoencoders in that they are all trained in an unsupervised manner to reconstruct inputs.

However, the mechanisms that the VAEs use to perform training are quite different. In a compression/denoising autoencoder, activations are mapped to activations throughout the layers, as in a standard neural network; comparatively, a VAE uses a probabilistic approach for the forward pass.

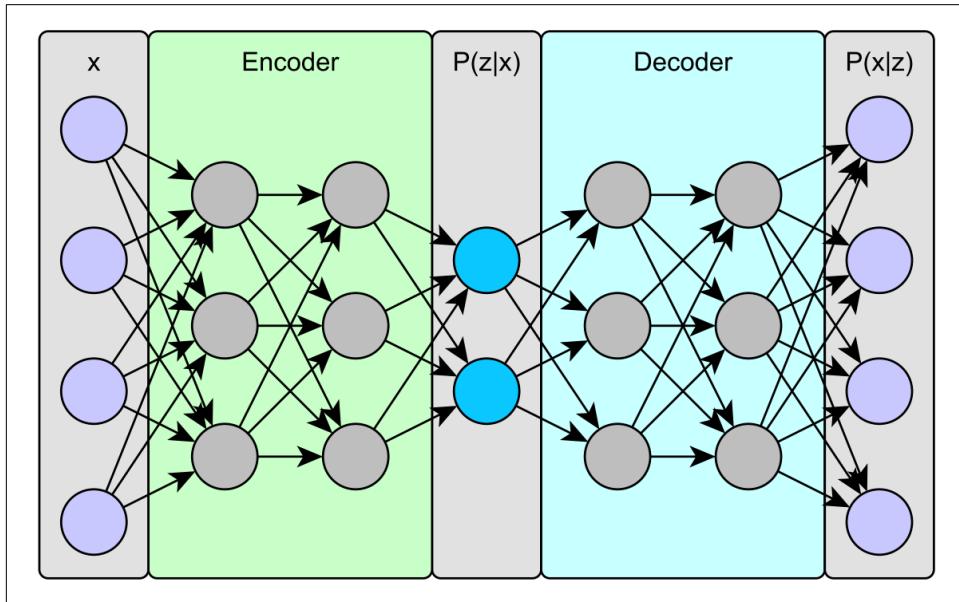


Figure 3-8. VAE network architecture

²¹ Kingma and Welling, 2013. “Auto-Encoding Variational Bayes.”

The VAE model assumes that the data x is generated in two steps: (a) a value $z^i \sim p(z)$ is generated from a prior distribution, and (b) the data instance is generated according to some conditional distribution $x^i \sim p(x|z)$. Of course, we don't actually know the values of z , and inferring $p(z|x)$ exactly is generally intractable. To handle this, we let both distributions, $p(z|x)$ and $p(x|z)$, be approximated by neural networks—the encoder and decoder, respectively. For example, if the $P(z|x)$ is Gaussian, the encoder forward-pass activations provide the Gaussian distribution parameters μ and σ^2 .

Similarly, the distribution parameters for $P(x|z)$ are provided by the decoder forward pass.²² Overall, the network is trained by backpropagation to maximize a lower bound on the marginal likelihood of the training data, $\log p(x^1, \dots, x^N)$. The VAE model also has been extended to allow unsupervised learning on time series with the variational recurrent autoencoder.²³ In [Chapter 5](#), we see the VAE in a practical example generating MNIST digits.

²² These distribution parameters should not be confused with the trainable network parameters; in practice, they are just network activations used to specify (for example) the mean and variance values for a Gaussian distribution, or the mean value for Bernoulli distribution.

²³ Fabius and van Amersfoort. 2014. “[Variational Recurrent Auto-Encoders](#).”

CHAPTER 4

Major Architectures of Deep Networks

The mother art is architecture. Without an architecture of our own we have no soul of our own civilization.

—Frank Lloyd Wright

Now that we've seen some of the components of deep networks, let's take a look at the four major architectures of deep networks and how we use the smaller networks to build them. Earlier in the book, we introduced four major network architectures:

- Unsupervised Pretrained Networks (UPNs)
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks
- Recursive Neural Networks

In this chapter, we take a look in more detail at each of these architectures. In [Chapter 2](#), we gave you a deeper understanding of the algorithms and math that underlie neural networks in general. In this chapter, we focus more on the higher-level architecture of different deep networks so as to build an understanding appropriate for applying these networks in practice.

Some networks we'll cover more lightly than others, but we'll mostly focus on the two major architectures that you will see in the wild: CNNs for image modeling and Long Short-Term Memory (LSTM) Networks (Recurrent Networks) for sequence modeling.

Unsupervised Pretrained Networks

In this group, we cover three specific architectures:

- Autoencoders
- Deep Belief Networks (DBNs)
- Generative Adversarial Networks (GANs)



A Note About the Role of Autoencoders

As we previously covered in [Chapter 3](#), autoencoders fundamental structures in deep networks because they're often used as part of larger networks. Like many other networks, they serve that role and then are used as a standalone network, as well.

Since we've already covered autoencoders in depth, we'll move on to looking closer at DBNs and GANs.

Deep Belief Networks

DBNs are composed of layers of Restricted Boltzmann Machines (RBMs) for the pre-train phase and then a feed-forward network for the fine-tune phase. [Figure 4-1](#) shows the network architecture of a DBN.

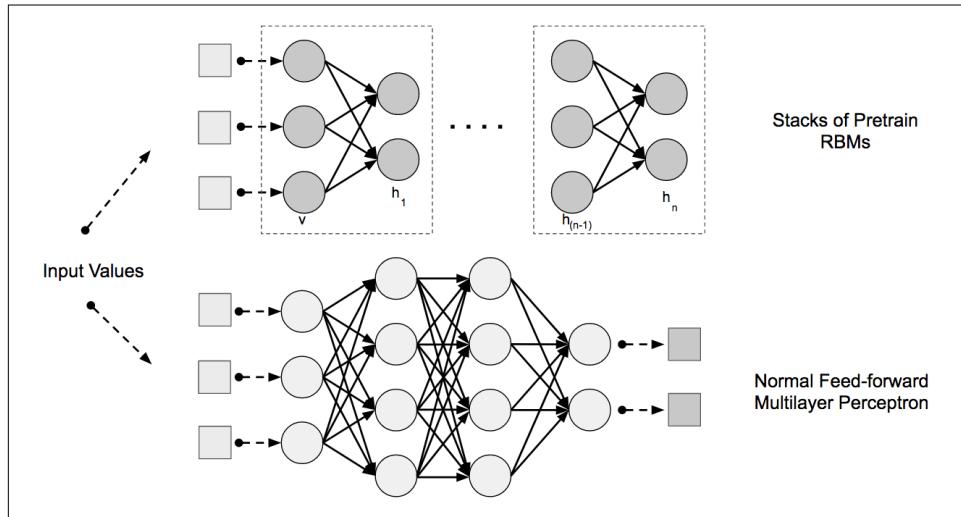


Figure 4-1. DBN architecture

In the sections that follow, we explain more about how DBNs take advantage of RBMs to better model training data.

Feature Extraction with RBM Layers

We use RBMs to extract higher-level features from the raw input vectors. To do that, we want to set the hidden unit states and weights such that when we show the RBM an input record and ask the RBM to reconstruct the record, it generates something pretty close to the original input vector. Hinton talks about this effect in terms of how machines “dream about data.”

The fundamental purpose of RBMs in the context of deep learning and DBNs is to learn these higher-level features of a dataset in an unsupervised training fashion. It was discovered that we could train better neural networks by letting RBMs learn progressively higher-level features using the learned features from a lower level RBM pretrain layer as the input to a higher-level RBM pretrain layer.

Learning higher-order features automatically. Learning these features in an unsupervised fashion is considered the pretrain phase of DBNs. Each hidden layer of the RBM in the pretrain phase learns progressively more complex features from the distribution of the data. These higher-order features are progressively combined in non-linear ways to do elegant automated feature engineering.

To visually understand feature construction with layers of RBMs, take a look at Figures 4-2, 4-3, and 4-4, which show the progression of activation renders on a RBM as it learns MNIST digits.

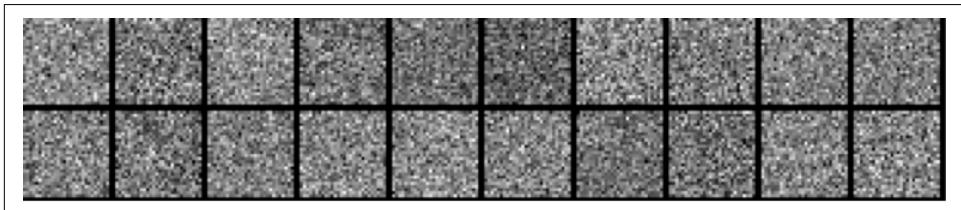


Figure 4-2. Activation render at the beginning of training

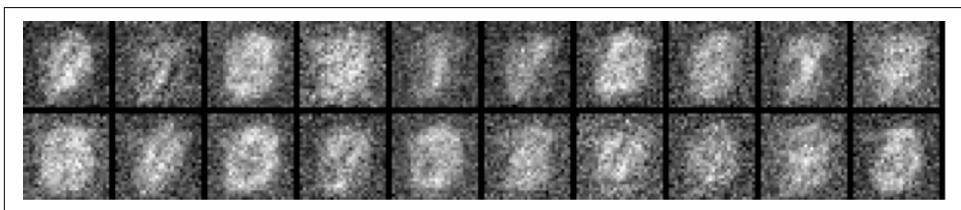


Figure 4-3. Features emerge in later activation render

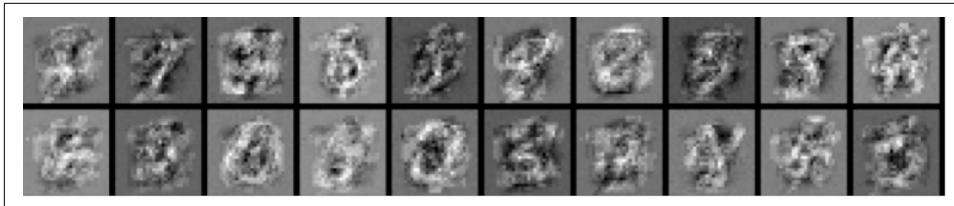


Figure 4-4. Portions of MNIST digits emerge towards end of training

Chapter 6 goes into more detail about how these renders are produced. We can see how a layer of RBM extracts fragments of digits as it trains. These features are then combined in higher-level layers to build progressively more complex (and nonlinear) features.

As this raw data is modeled in the generative modeling process with each layer of the RBM, the system is able to extract increasingly higher-level features from the raw input data produced by our baseline input vectorization process. These features move forward through these layers of RBMs in one direction, producing more elaborate features at the top layer.

Initializing the feed-forward network. We then use these layers of features as the initial weights in a traditional backpropagation driven feed-forward neural network.

These initialization values help the training algorithm guide the parameters of the traditional neural network toward better regions of parameter search space. This phase is known as the *fine-tune phase* of DBNs.

Fine-tuning a DBN with a feed-forward multilayer neural network

In the fine-tune phase of a DBN we use normal backpropagation with a lower learning rate to do “gentle” backpropagation. We consider the pretraining phase to be a general search of the parameter space in an unsupervised fashion based on raw data. In contrast, the fine-tune phase is about specializing the network and its features for the task we actually care about (such as classification).

Gentle backpropagation. The pretrain phase with RBM learns higher-order features from the data, which we use as good initial starting values for our feed-forward network. We want to take these weights and tune them a bit more to find good values for our final neural network model.

The output layer. The normal goal of a deep network is to learn a set of features. The first layer of a deep network learns how to reconstruct the original dataset. The subsequent layers learn how to reconstruct the probability distributions of the activations of the previous layer. The output layer of a neural network is tied to the overall objective. This is typically logistic regression, with the number of features equal to the

number of inputs of the final layer, and the number of outputs equal to the number of classes.

Current state of DBNs

We don't cover DBNs as extensively as the other network architectures in this book. This is because the field has largely seen CNNs take over the image modeling space and thus we chose to emphasize that architecture more, as you'll see in the next section.



The Role of DBNs in the Rise of Deep Learning

Although we don't emphasize DBNs as much in this book, this network played a nontrivial role in the rise of deep learning. Geoff Hinton's team at the University of Toronto persisted over a long period of time in advancing techniques in the image modeling space to produce great advances. We felt it important to note the role that DBNs have played in the evolution of deep networks.

Generative Adversarial Networks

A network of note is the GAN.¹ GANs have been shown to be quite adept at synthesizing novel new images based on other training images. We can extend this concept to model other domains such as the following:

- Sound
- Video²
- Generating images from text descriptions³

GANs are an example of a network that uses unsupervised learning to train two models in parallel. A key aspect of GANs (and generative models in general) is how they use a parameter count that is significantly smaller than normal with respect to the amount of data on which we're training the network. The network is forced to efficiently represent the training data, making it more effective at generating data similar to the training data.

Training generative models, unsupervised learning, and GANs

If we had a large corpus of training images (such as the [ImageNet](#) dataset), we could build a generative neural network that outputs images (as opposed to classifications).

¹ Goodfellow et al. 2014. “[Generative Adversarial Networks](#).”

² Vondrick, Pirsavash, and Torralba. 2016. “[Generating Videos with Scene Dynamics](#).”

³ Zhang et al. 2016. “[StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks](#).”

We'd consider these generated output images to be samples from the model. The generative model in GANs generates such images while a secondary "discriminator" network tries to classify these generated images.

This secondary discriminator network attempts to classify the output images as real or synthetic. When training GANs, we want to update the parameters such that the network will generate more believable output images based on the training data. The goal here is to make images realistic enough that the discriminator network is fooled to the point that it cannot distinguish the difference between the real and the synthetic input data.

An example of efficient model representation in GANs is how they typically have around 100 million parameters when modeling a dataset such as ImageNet. In the course of training, an input dataset such as ImageNet (200 GB) becomes close to 100 MB of parameters. This learning process tries to find the most efficient way to represent the features in the data, such as similar groups of pixels, edges, and other patterns (as we'll see in more detail in ["Convolutional Neural Networks \(CNNs\)" on page 125](#)).

The discriminator network. When modeling images, the discriminator network is typically a standard CNN. Using a secondary neural network as the discriminator network allows the GAN to train both networks in parallel in an unsupervised fashion. These discriminator networks take images as input, and then output a classification.

The gradient of the output of the discriminator network with respect to the synthetic input data indicates how to make small changes to the synthetic data to make it more realistic.

The generative network. The generative network in GANs generates data (or images) with a special kind of layer called a *deconvolutional layer* (read more about deconvolutional networks and layers in the following sidebar).

During training, we use backpropagation on both networks to update the generating network's parameters to generate more realistic output images. The goal here is to update the generating network's parameters to the point at which the discriminating network is sufficiently "fooled" by the generating network because the output is so realistic as compared to the training data's real images.

Deconvolutional Networks

This type of network was developed by Matthew Zeiler and Rob Fergus from New York University as part of the development of ZF Net in the paper “[Visualizing and Understanding Convolutional Neural Networks](#)” (2013). A deconvolutional network helps us examine different feature activations and their relation to the input space ([Figure 4-5](#)).

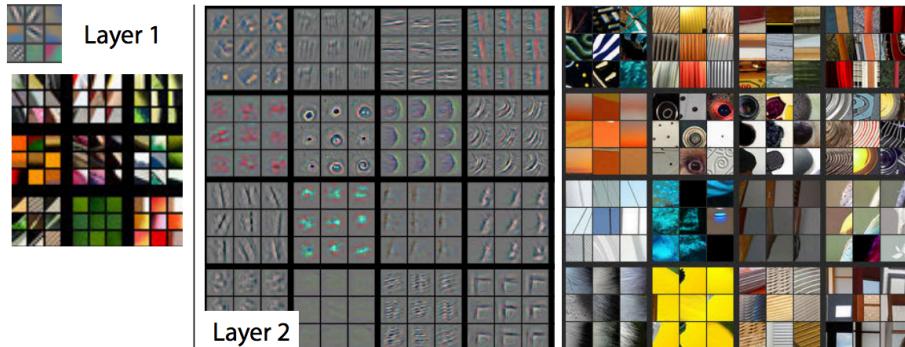


Figure 4-5. Visualizing deconvolutional layers

The deconvolutional layers in a deconvolutional network (“deconvnet” for short) map features to pixels when modeling images, as seen in [Figure 4-5](#), which is the opposite of what a normal convolutional layer does. This aspect of deconvolutional networks is what enables us to generate images as output from neural networks. Deconvolutional networks are unsupervised and trained in a layer-wise fashion, similar to a DBN. The network has multiple stacked deconvolutional layers where each layer is trained on the input of the previous layer. The general idea with the information passing through the layers is that the output from each layer is a sparse representation of the input to the layer.

Building generative models and Deep Convolutional Generative Adversarial Networks

One variant of GANs is the [Deep Convolutional Generative Adversarial Network](#) (DCGAN). [Figure 4-6](#) depicts images of bedrooms generated from a DCGAN.



Figure 4-6. Generated images of bedrooms from a DCGAN network⁴

This network takes random numbers (from a uniform distribution) and generates an image from the network model as output. As the input random numbers change we see the DCGAN generate different types of images.

Conditional GANs

Conditional GANs⁵ also can use class label information, allowing them to conditionally generate data of a specific class.

Comparing GANs and variational autoencoders

GANs focus on trying to classify training records as being from the model distribution or the real distribution. When the discriminator model makes a prediction in which there is a difference between the two distributions, the generator network adjusts its parameters. Eventually the generator converges on parameters that reproduce the real data distribution, and the discriminator is unable to detect the difference.

With variational autoencoders (VAEs) we're setting up this same problem with probabilistic graphical models to reconstruct the input in an unsupervised fashion, as seen previously in [Chapter 3](#). VAEs attempt to maximize a lower bound on the log likelihood of the data such that the generated images look more and more real.

Another interesting difference between GANs and VAEs is how the images are generated. With basic GANs the image is generated with arbitrary code and we don't have a way to generate a picture with specific features. VAEs, in contrast, have a specific encode/decode scheme with which we can compare the generated image to the origi-

⁴ Image is from the [DCGAN authors' GitHub repository](#).

⁵ Mirza and Osindero. 2014. “Conditional Generative Adversarial Nets.”

nal image. This gives us the side effect of being able to code for specific types of images to be generated.



Issues with Generative Models

Sometimes with generated images we also get a different type of noise in the generated output image. The downside of VAE-generated images is that as a result of how they are generated, the images are sometimes slightly blurry. GAN-generated images tend to capture the style of the input data yet sometimes don't compose the scene in a coherent manner (e.g., it's an image of a dog, but the dog doesn't look quite right).

Convolutional Neural Networks (CNNs)

The goal of a CNN is to learn higher-order features in the data via convolutions. They are well suited to object recognition with images and consistently top image classification competitions. They can identify faces, individuals, street signs, platypuses, and many other aspects of visual data. CNNs overlap with text analysis via optical character recognition, but they are also useful when analyzing words⁶ as discrete textual units. They're also good at analyzing sound.

The efficacy of CNNs in image recognition is one of the main reasons why the world recognizes the power of deep learning. As Figure 4-7 illustrates, CNNs are good at building position and (somewhat) rotation invariant features from raw image data.



Figure 4-7. CNNs and computer vision

⁶ Kalchbrenner et al. 2016. “Neural Machine Translation in Linear Time.”

CNNs are powering major advances in machine vision, which has obvious applications for self-driving cars, robotics, drones, and treatments for the visually impaired.



CNNs and Structure in Data

CNNs tend to be most useful when there is some structure to the input data. An example would be how images and audio data that have a specific set of repeating patterns and input values next to each other are related spatially. Conversely, the columnar data exported from a relational database management system (RDBMS) tends to have no structural relationships spatially. Columns next to one another just happen to be materialized that way in the database exported materialized view.

CNNs also have been used in other tasks such as natural language translation/generation⁷ and sentiment analysis.⁸ A convolution is a powerful concept for helping to build a more robust feature space based on a signal.

Biological Inspiration

The biological inspiration for CNNs is the visual cortex in animals.⁹ The cells in the visual cortex are sensitive to small subregions of the input. We call this the *visual field* (or *receptive field*). These smaller subregions are tiled together to cover the entire visual field. The cells are well suited to exploit the strong spatially local correlation found in the types of images our brains process, and act as local filters over the input space. There are two classes of cells in this region of the brain. The simple cells activate when they detect edge-like patterns, and the more complex cells activate when they have a larger receptive field and are invariant to the position of the pattern.

Intuition

Feed-forward multilayer neural networks take input as a single one-dimensional vector and transform the data with one or more hidden layers (fully connected). The network then gives a result from the output layer. The issue we run into with traditional multilayer neural networks and image data is that these networks don't scale well with image data as input. An example would be modeling the CIFAR-10 dataset (see the upcoming sidebar). The images to train on are only 32 pixels wide by 32

⁷ Gehring et al. 2016. “A Convolutional Encoder Model for Neural Machine Translation.”

⁸ Nogueira dos Santos and Gatti. 2014. “Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts.”

⁹ Eickenberg et al. 2017. “Seeing it all: Convolutional network layers map the function of the human visual system.”

pixels in height with 3 channels of RGB information. This creates 3,072 weights per neuron in the first hidden layer, however, and we'll probably want more than one neuron in that hidden layer. In many cases, we'll want multiple hidden layers in our multilayer neural network, which will multiply those weights, as well.

What Is the CIFAR-10 Dataset?

The CIFAR-10 dataset is a well-known [image classification benchmark dataset](#) compiled by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset has 60,000 color images with 10 different classes comprising 6,000 images per class. The images are 32×32 pixels each. There are 50,000 training images and 10,000 test images. [Figure 4-8](#) shows the image classes in the dataset.

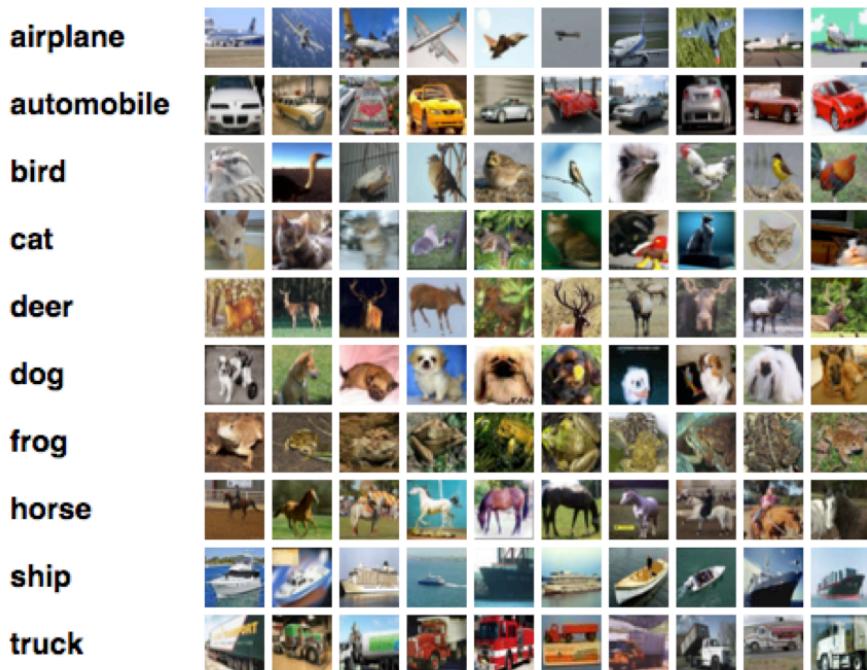


Figure 4-8. The CIFAR-10 dataset

The classes do not overlap, meaning that an image of a truck will contain only an image of a truck. The dataset is around 170 MB in size.

A normal image could easily be 300 pixels in width by 300 pixels in height with 3 channels of RGB information. This would create 270,000 connection weights per hidden neuron. This shows how quickly a fully connected multilayer network creates

a massive number of connections when modeling image data. The structure of image data allows us to change the architecture of a neural network in a way that we can take advantage of this structure. With CNNs, we can arrange the neurons in a three-dimensional structure for which we have the following:

- Width
- Height
- Depth

These attributes of the input match up to an image structure for which we have:

- Image width in pixels
- Image height in pixels
- RGB channels as the depth

We can consider this structure to be a three-dimensional volume of neurons. A significant aspect to how CNNs evolved from previous feed-forward variants is how they achieved computational efficiency with new layer types. We'll cover this arrangement in more depth momentarily. Let's now take a look at the high-level general architecture of CNNs.

CNN Architecture Overview

CNNs transform the input data from the input layer through all connected layers into a set of class scores given by the output layer. There are many variations of the CNN architecture, but they are based on the pattern of layers, as demonstrated in [Figure 4-9](#).

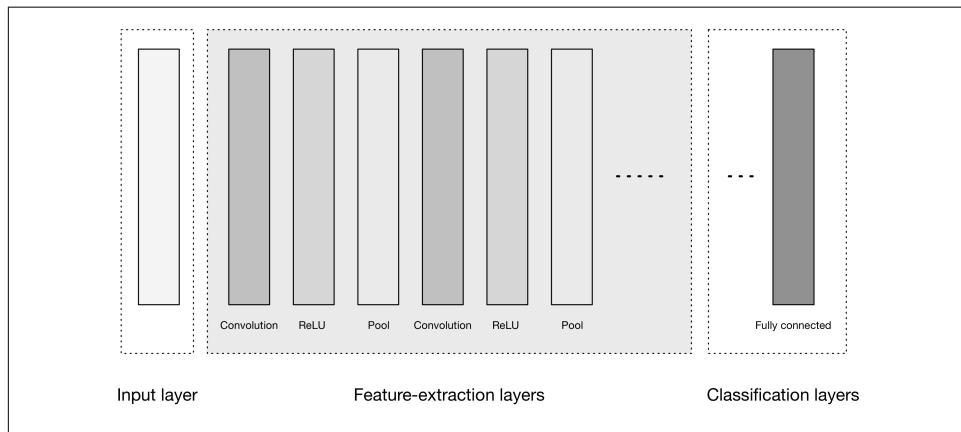


Figure 4-9. High-level general CNN architecture

Figure 4-9 depicts three major groups:

1. Input layer
2. Feature-extraction (learning) layers
3. Classification layers

The input layer accepts three-dimensional input generally in the form spatially of the size (width \times height) of the image and has a depth representing the color channels (generally three for RGB color channels).



Mini-Batch as the Fourth Dimension

When we batch examples together into a mini-batch for training, we end up with four dimensions—one more dimension to index the example within the mini-batch. Thus, in DL4J, image training data arrays have four dimensions, not just three.

The feature-extraction layers have a general repeating pattern of the sequence:

1. Convolution layer

We express the Rectified Linear Unit (ReLU) activation function as a layer in the diagram here to match up to other literature.

2. Pooling layer

These layers find a number of features in the images and progressively construct higher-order features. This corresponds directly to the ongoing theme in deep learning by which features are automatically learned as opposed to traditionally hand engineered.

Finally we have the classification layers in which we have one or more fully connected layers to take the higher-order features and produce class probabilities or scores. These layers are fully connected to all of the neurons in the previous layer, as their name implies. The output of these layers produces typically a two-dimensional output of the dimensions $[b \times N]$, where b is the number of examples in the mini-batch and N is the number of classes we're interested in scoring.

Neuron spatial arrangements

Recall how in traditional multilayer neural networks, the layers are fully connected and every neuron in a layer is connected to every neuron in the next layer. The neurons in the layers of a CNN are arranged in three dimensions to match the input volumes. Here, depth means the third dimension of the activation volume, not the number of layers, as in a multilayer neural network.

Evolution of the connections between layers

Another change is how we connect layers in a convolutional architecture. Neurons in a layer are connected to only a small region of neurons in the layer before it. CNNs retain a layer-oriented architecture, as in traditional multilayer networks, but have different types of layers. Each layer transforms the 3D input volume from the previous layer into a 3D output volume of neuron activations with some differentiable function that might or might not have parameters, as demonstrated in [Figure 4-10](#).

Input Layers

Input layers are where we load and store the raw input data of the image for processing in the network. This input data specifies the width, height, and number of channels. Typically, the number of channels is three, for the RGB values for each pixel.

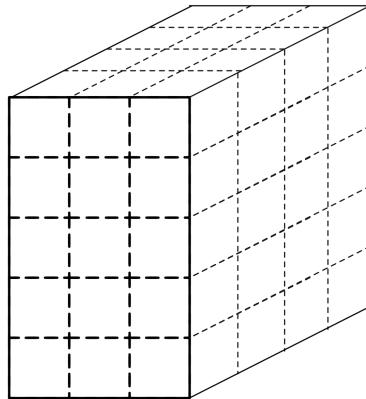


Figure 4-10. Input layer 3D volume

Convolutional Layers

Convolutional layers are considered the core building blocks of CNN architectures. As [Figure 4-11](#) illustrates, convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer. The layer will compute a dot product between the region of the neurons in the input layer and the weights to which they are locally connected in the output layer.

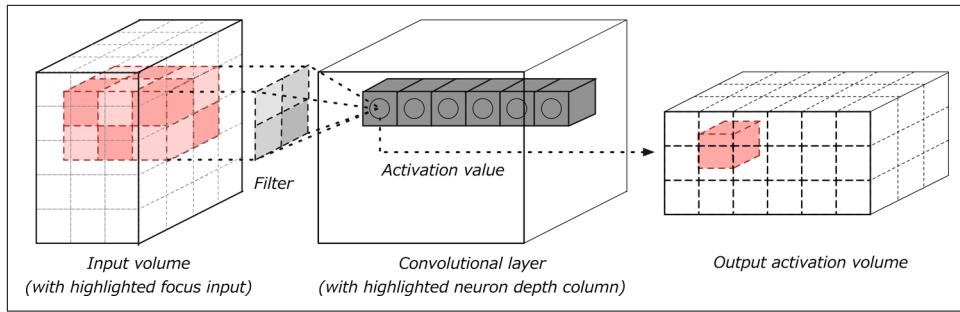


Figure 4-11. Convolution layer with input and output volumes

The resulting output generally has the same spatial dimensions (or smaller spatial dimensions) but sometimes increases the number of elements in the third dimension of the output (depth dimension). Let's take a closer look at a key concept in these layers, called a *convolution*.

Convolution

A convolution is defined as a mathematical operation describing a rule for how to merge two sets of information. It is important in both physics and mathematics and defines a bridge between the space/time domain and the frequency domain through the use of Fourier transforms. It takes input, applies a convolution kernel, and gives us a feature map as output.

The convolution operation, shown in [Figure 4-12](#), is known as the *feature detector* of a CNN. The input to a convolution can be raw data or a feature map output from another convolution. It is often interpreted as a filter in which the kernel filters input data for certain kinds of information; for example, an edge kernel lets pass through only information from the edge of an image.

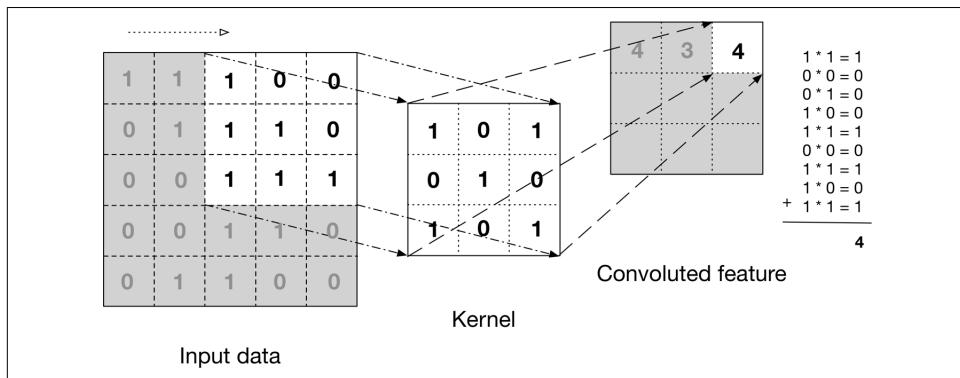


Figure 4-12. The convolution operation

The figure illustrates how the kernel is slid across the input data to produce the convoluted feature (output) data. At each step, the kernel is multiplied by the input data values within its bounds, creating a single entry in the output feature map. In practice the output is large if the feature we're looking for is detected in the input.

We commonly refer to the sets of weights in a convolutional layer as a filter (or kernel). This filter is convolved with the input and the result is a feature map (or activation map). Convolutional layers perform transformations on the input data volume that are a function of the activations in the input volume and the parameters (weights and biases of the neurons). The activation map for each filter is stacked together along the depth dimension to construct the 3D output volume.

Convolutional layers have parameters for the layer and additional hyperparameters. Gradient descent is used to train the parameters in this layer such that the class scores are consistent with the labels in the training set. Following are the major components of convolutional layers:

- Filters
- Activation maps
- Parameter sharing
- Layer-specific hyperparameters

Let's take a look of the specifics of each component.

Filters

The parameters for a convolutional layer configure the layer's set of filters. Filters are a function that has a width and height smaller than the width and height of the input volume.



Filter Sizes in Natural Language Processing Applications

We can have a filter size equal to the input volume, but typically only in one dimension, not both. This would be something to be aware of for cases in which we'd apply CNNs to use cases in Natural Language Processing (NLP).

Filters (e.g., convolutions) are applied across the width and height of the input volume in a sliding window manner, as demonstrated in [Figure 4-12](#). Filters are also applied for every depth of the input volume. We compute the output of the filter by producing the dot product of the filter and the input region.



Filter Count and Activation Maps

The output of applying a filter to the input volume is known as the *activation map* (sometimes referred to as a *feature map*) of that filter. In many CNN diagrams, we often see lots of small activation maps; how these are produced can sometimes be confusing.

The filter count is a hyperparameter value for each convolutional layer. This hyperparameter also controls how many activation maps are produced from the convolutional layer as input into the next layer and is considered the third dimension (number of activation maps) in the 3D layer output activation volume. The filter count hyperparameter can be chosen freely yet some values will work better than others.

The architecture of CNNs is set up such that the learned filters produce the strongest activation to spatially local input patterns. This means that filters are learned that will activate on patterns (or features) only when the patterns occur in the training data in their respective field. As we move farther along in layers in a CNN, we encounter filters that can recognize nonlinear combinations of features and are increasingly global in how they can detect patterns. High-performing convolutional architectures (which we'll see later in this section) have shown network depth to be an important factor in CNNs.

Activation maps

Recall from [Chapter 1](#) that an activation is a numerical result if a neuron decided to let information pass through. This is a function of the inputs to the activation function, the weights on the connections (for the inputs, and the type of activation function itself). When we say the filter “activates,” we mean that the filter lets information pass through it from the input volume into the output volume.

We slide each filter across the spatial dimensions (width, height) of the input volume during the forward pass of information through the CNN. This produces a two-dimensional output called an activation map for that specific filter. [Figure 4-13](#) depicts how this activation map relates to our previously introduced concept of a convoluted feature.

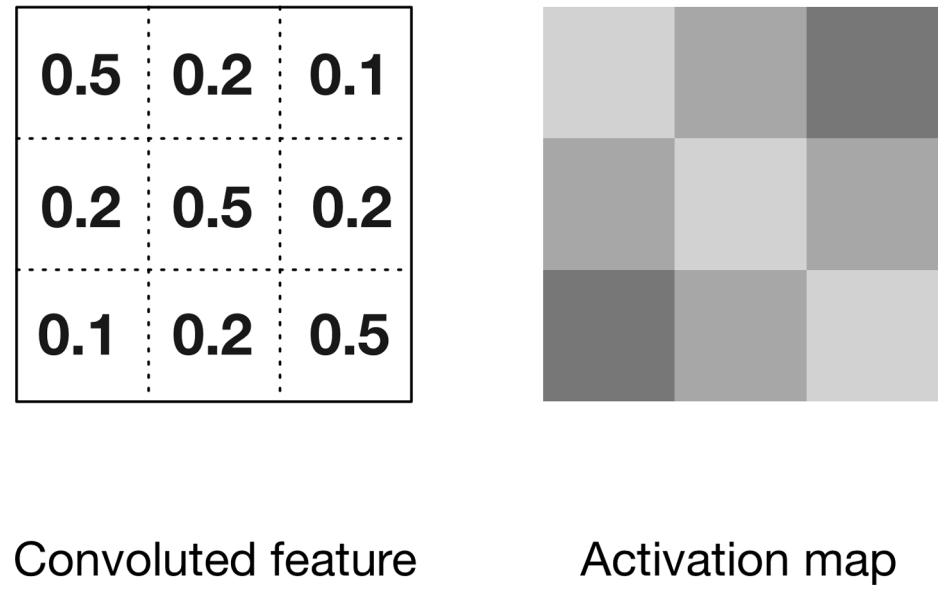


Figure 4-13. Convolutions and activation maps

The activation map on the right in Figure 4-13 is rendered differently to illustrate how convolutional activation maps are commonly rendered in the literature.



Activation Maps

In some literature, the activation map output is called a *feature map*, but for this text, we'll refer to it as an activation map.

To compute the activation map, we slide the filter across the input volume depth slice. We calculate the dot product between the entries in the filter and the input volume. The filter represents the weights that are being multiplied by the moving window (subset) of input activations. Networks learn filters that activate when they see certain types of features in the input data in a specific spatial position.

We create the three-dimensional output volume for the convolution layer by stacking these activation maps along the depth dimension in the output, as shown in Figure 4-14. The output volume will have entries that we consider the output of a neuron that look at only a small window of the input volume.

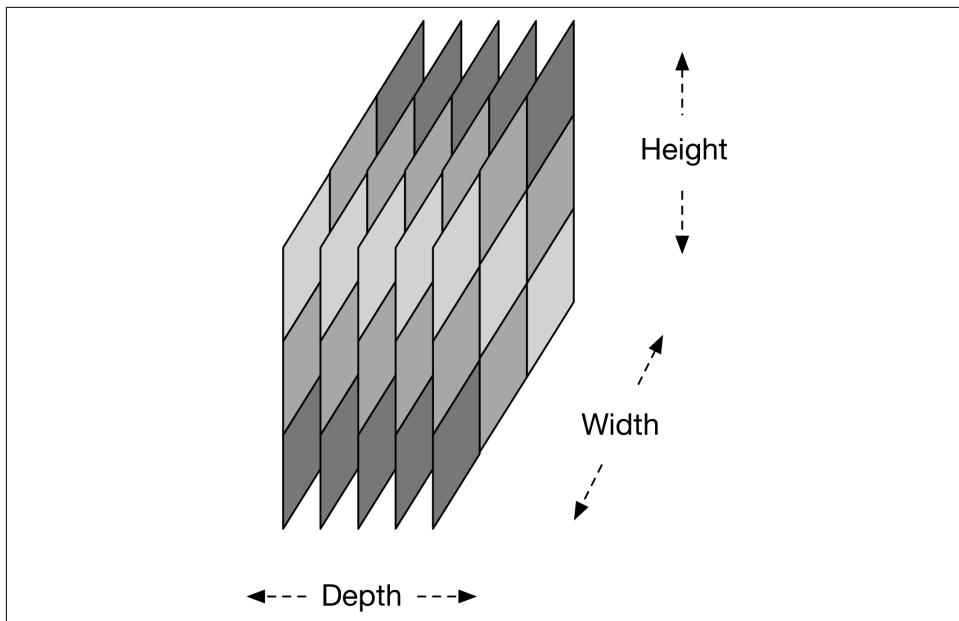


Figure 4-14. Activation volume output of convolutional layer

In some cases, this output will be the result of parameters shared with neurons in the same activation map. Each neuron generating the output volume is connected to only a local region of the input volume, as depicted in [Figure 4-15](#).

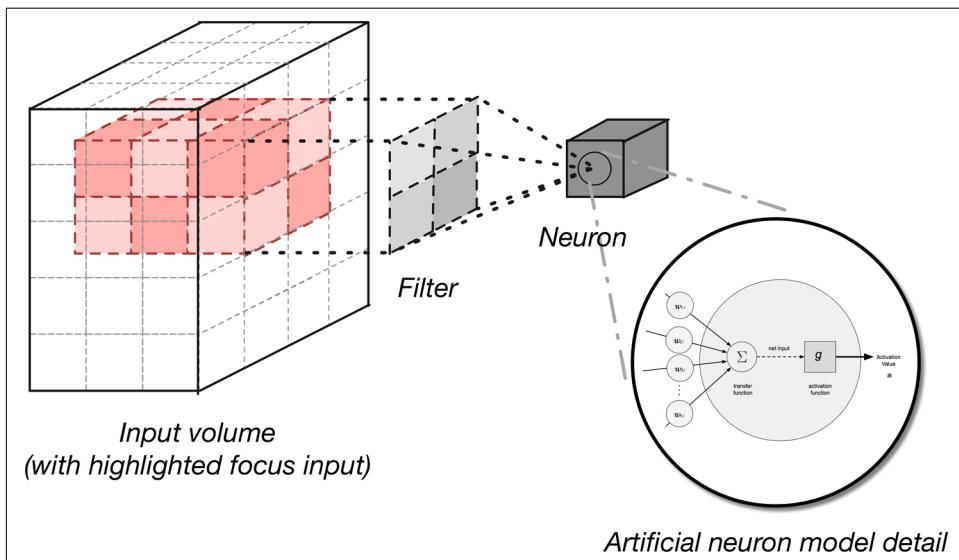


Figure 4-15. Generating an activation output volume

We control local connectivity of this process with the hyperparameter called the *receptive field*, which controls how much of the width and height of the input volume against which our filter maps.

Controlling Local Connectivity with the Receptive Field

Neurons in a layer are connected to a smaller region of the input volume along the spatial dimensions (width and height), but connectivity along the depth axis is always equal to the depth of the input volume. This means that we always have full connectivity along the depth of the input volume. Let's look at an example of this based on the CIFAR-10 input data (RGB image) you saw in “[What Is the CIFAR-10 Dataset?](#)” on page 127.

In this situation, the input volume has the size $32 \times 32 \times 3$ and our receptive field hyperparameter is set as 5×5 . Each neuron in the convolutional layer will have weights to a $5 \times 5 \times 3$ region in the input volume. This gives us $5 \times 5 \times 3 = 75$ weights for each neuron in our convolution layer.

We note that the depth of the input volume is 3, which is always the depth of the weight connectivity on the convolution layer. The region the neuron is connecting to is smaller than the width and height of the image but the depth always stays the same.

Our connectivity in convolutional layers might be local but the neurons themselves remain unchanged. We're still computing a dot product of the weights with the input with a nonlinear function.

The only difference now is the neuron is connected to only a subset of the input and not every neuron from the previous layer, as in traditional multilayer neural networks. We'd consider this connectivity to be full depth but spatially local.

Filters define a smaller bounded region to generate activation maps from the input volumes. They are connected to only a subset of the input volume through the dynamics of local connectivity described earlier. This allows us to still have quality feature extraction while reducing the number of parameters per layer we need to train. Convolutional layers reduce the parameter count further by using a technique called *parameter sharing*.

Parameter sharing

CNNs use a parameter-sharing scheme to control the total parameter count. This helps training time because we'll use fewer resources to learn the training dataset. To implement parameter sharing in CNNs, we first denote a single two-dimensional slice of depth as a “depth slice.” We then constrain the neurons in each depth slice to use the same weights and bias. This gives us significantly fewer parameters (or weights) for a given convolutional layer.

We are not able to take advantage of parameter sharing when the input images we're training on have a specific centered structure. We see this effect in faces when we always expect a specific feature to appear in a specific place (for centered faces). In this case we'd probably not use parameter sharing. Parameter sharing is what gives CNNs invariance to translation/position, as well.

Learned filters and renders

Figure 4-16 presents an example of the learned 96 filters of size $11 \times 11 \times 3$. With the parameter-sharing scheme, we see that detecting a horizontal edge is useful in many places in the image due to the translationally invariant nature of images. This means that we're able to learn the horizontal edge in one place and then not worry about learning it as a feature in all positions in the image.

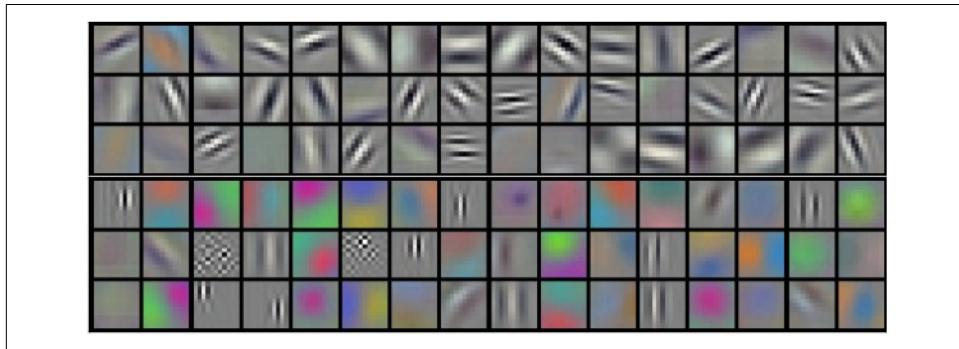


Figure 4-16. Example filters learned by Krizhevsky et al.¹⁰ (96 filters, $11 \times 11 \times 3$)

Breaking this up a bit, let's think about a 2D image. If we subdivide an image into four sections, the neural network will then learn position-invariant features of the image. The reason these are position invariant is because of how the network subdivides the data into quadrants. It then learns portions of the image at a time and pools the results. This allows the neural network to learn an overall representation that isn't local to any particular set of features. We'll talk more about generating filter renders in Chapter 6.



Rotation Invariance in Features for CNNs

The learned features are position invariant by design—but they aren't (in general) rotation invariant. You can achieve some level of rotation invariance with appropriate data augmentation, however.

¹⁰ Krizhevsky et al. 2012. “ImageNet Classification with Deep Convolutional Neural Networks.”

ReLU activation functions as layers

With CNNs, we often see ReLU layers used. The ReLU layer will apply an element-wise activation function over the input data thresholding—for example, $\max(0, x)$ —at zero, giving us the same dimension output as the input to the layer.



DL4J, layer types, and activation functions

In DL4J, we identify layers with their neuron activation function type (but this is not always reflected in the layer class names). DL4J has activation functions built into the layers themselves. Other libraries such as Caffe use separate activation layers.

Running this function over the input volume will change the pixel values but will not change the spatial dimensions of the input data in the output. ReLU layers do not have parameters nor additional hyperparameters.

Convolutional layer hyperparameters

Following are the hyperparameters¹¹ that dictate the spatial arrangement and size of the output volume from a convolutional layer are:

- Filter (or kernel) size (field size)
- Output depth
- Stride
- Zero-padding



More on Sizing Convolutional Layers in Chapter 7

In this section, we explain how these hyperparameters work. In [Chapter 7](#), we lay out the tuning mechanics for CNN layers.

Filter size. Every filter is small spatially with respect to the width and height of the filter size. An example of this is how the first convolutional layer might have a $5 \times 5 \times 3$ -sized filter. This would mean the filter is 5 pixels wide by 5 pixels tall, with 3 representing the color channels, assuming that the input image was in 3-channel RGB color.

¹¹ Another hyperparameter commonly seen in CNNs is called dilation. DL4J currently does not support dilation as of version 0.7.

Output depth. We can manually pick the depth of the output volume. The depth hyperparameter controls the neuron count in the convolutional layer that is connected to the same region of the input volume.



Edges and Activation

Different neurons along the depth dimension learn to activate when stimulated with created input data (e.g., color or edges).

We consider a set of neurons that all look at the same region of input volume as a depth column.

Stride. Stride configures how far our sliding filter window will move per application of the filter function. Each time we apply the filter function to the input column, we create a new depth column in the output volume. Lower settings for stride (for example, 1 specifies only a single unit step) will allocate more depth columns in the output volume. This also will yield more heavily overlapping receptive fields between the columns, leading to larger output volumes. The opposite is true when we specify higher stride values. These higher stride values give us less overlap and smaller output volumes spatially.

Zero-padding. The last hyperparameter is zero-padding, with which we can control the spatial size of the output volumes. We'd want to do this for cases in which we want to maintain the spatial size of the input volume in the output volume.

Batch normalization and layers

To accelerate training in CNNs we can normalize the activations of the previous layer at each batch.¹² This technique applies a transformation that keeps the mean activation close to 0.0 while also keeping the activation standard deviation close to 1.0.

Batch normalization in CNNs has shown to **speed up training** by making normalization part of the network architecture. By applying normalization for each training mini-batch of input records, we can use much higher learning rates. **Batch normalization** also reduces the sensitivity of training toward weight initialization and acts as a regularizer (reducing the need for other types of regularization). Batch normalization has also been applied to LSTM networks,¹³ which is another type of deep network we'll discuss later in the chapter.

¹² Ioffe and Szegedy. 2015. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.”

¹³ Cooijmans et al. 2016. “Recurrent Batch Normalization.”

Pooling Layers

Pooling layers are commonly inserted between successive convolutional layers. We want to follow convolutional layers with pooling layers to progressively reduce the spatial size (width and height) of the data representation. Pooling layers reduce the data representation progressively over the network and help control overfitting. The pooling layer operates independently on every depth slice of the input.



Common DownSampling Operations

The most common downsampling operation is the *max operation*. The next most common operation is *average pooling*.

The pooling layer uses the `max()` operation to resize the input data spatially (width, height). This operation is referred to as *max pooling*. With a 2×2 filter size, the `max()` operation is taking the largest of four numbers in the filter area. This operation does not affect the depth dimension.

Pooling layers use filters to perform the downsampling process on the input volume. These layers perform downsampling operations along the spatial dimension of the input data. This means that if the input image were 32 pixels wide by 32 pixels tall, the output image would be smaller in width and height (e.g., 16 pixels wide by 16 pixels tall). The most common setup for a pooling layer is to apply 2×2 filters with a stride of 2. This will downsample each depth slice in the input volume by a factor of two on the spatial dimensions (width and height). This downsampling operation will result in 75 percent of the activations being discarded.

Pooling layers do not have parameters for the layer but do have additional hyperparameters. This layer does not involve parameters, because it computes a fixed function of the input volume. It is not common to use zero-padding for pooling layers.

Fully Connected Layers

We use this layer to compute class scores that we'll use as output of the network (e.g., the output layer at the end of the network). The dimensions of the output volume is $[1 \times 1 \times N]$, where N is the number of output classes we're evaluating. In the case of the CIFAR dataset we discussed above, N would be 10 for the 10 classes of objects in the dataset. This layer has a connection between all of its neurons and every neuron in the previous layer.

Fully connected layers have the normal parameters for the layer and hyperparameters. Fully connected layers perform transformations on the input data volume that are a function of the activations in the input volume and the parameters (weights and biases of the neurons).



Multiple Fully Connected Layers

There are some CNN architectures that will use multiple fully connected layers at the end of the network. [AlexNet](#) is an example of this; it has two fully connected layers followed by a softmax layer at the end.

Other Applications of CNNs

Beyond normal two-dimensional image data, we also see CNNs applied to three-dimensional datasets. Here are some examples of these alternative uses:

- MRI data¹⁴
- 3D shape data¹⁵
- Graph data¹⁶
- NLP applications¹⁷

The position-invariant nature of CNNs has proven useful in these domains because we're not limited to hand-coding our features to appear in certain "spots" in the feature vector.

CNNs of Note

Following is a list of some of the more popular architectures of CNNs.

¹⁴ Milletari, Navab, and Ahmadi. 2016. "[V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation](#)."

¹⁵ Maturana and Scherer. 2015. "[VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition](#)."

¹⁶ Henaff, Bruna, and LeCun. 2015. "[Deep Convolutional Networks on Graph-Structured Data](#)."

¹⁷ Conneau et al. 2016. "[Very Deep Convolutional Networks for Text Classification](#)."

- LeNet¹⁸
 - One of the earliest successful architectures of CNNs
 - Developed by Yann Lecun
 - Originally used to read digits in images
- AlexNet¹⁹
 - Helped popularize CNNs in computer vision
 - Developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton
 - Won the ILSVRC 2012
- ZF Net²⁰
 - Won the ILSVRC 2013
 - Developed by Matthew Zeiler and Rob Fergus
 - Introduced the visualization concept of the Deconvolutional Network
- GoogLeNet²¹
 - Won the ILSVRC 2014
 - Developed by Christian Szegedy and his team at Google
 - Codenamed “Inception,” one variation has 22 layers
- VGGNet²²
 - Runner-Up in the ILSVRC 2014
 - Developed by Karen Simonyan and Andrew Zisserman
 - Showed that depth of network was a critical factor in good performance
- ResNet²³
 - Trained on very deep networks (up to 1,200 layers)
 - Won first in the ILSVRC 2015 classification task

Summary

CNNs evolved due to the need for specialized feature extraction from image data. We saw layers that are good at finding features no matter where they “roam” across columns. We saw how convolutional layers, pooling layers, and regular fully connected layers worked together to do image classification. Now, let’s move on to a neural net-

¹⁸ LeCun et al. 1998. “Gradient-based learning applied to document recognition.”

¹⁹ Krizhevsky, Sutskever, and Hinton. 2012. “ImageNet Classification with Deep Convolutional Neural Networks.”

²⁰ Zeiler and Fergus. 2013. “Visualizing and Understanding Convolutional Networks.”

²¹ Szegedy et al. 2015. “Going Deeper with Convolutions.”

²² Simonyan and Zisserman. 2015. “Very Deep Convolutional Networks for Large-Scale Image Recognition.”

²³ He et al. 2015. “Deep Residual Learning for Image Recognition.”

work architecture focused on modeling the temporal domain: Recurrent Neural Networks.

Recurrent Neural Networks

Recurrent Neural Networks are in the family of feed-forward neural networks. They are different from other feed-forward networks in their ability to send information over time-steps. Here's an interesting explanation of Recurrent Neural Networks from leading researcher Juergen Schmidhuber:

[Recurrent Neural Networks] allow for both parallel and sequential computation, and in principle can compute anything a traditional computer can compute. Unlike traditional computers, however, Recurrent Neural Networks are similar to the human brain, which is a large feedback network of connected neurons that somehow can learn to translate a lifelong sensory input stream into a sequence of useful motor outputs. The brain is a remarkable role model as it can solve many problems current machines cannot yet solve.

Historically, these networks have been difficult to train, but more recently, advances in research (optimization, network architectures, parallelism, and graphics processing units [GPUs]) have made them more approachable for the practitioner.

Recurrent Neural Networks take each vector from a sequence of input vectors and model them one at a time. This allows the network to retain state while modeling each input vector across the window of input vectors. Modeling the time dimension is a hallmark of Recurrent Neural Networks.

Modeling the Time Dimension

Recurrent Neural Networks are considered Turing complete and can simulate arbitrary programs (with weights). If we view neural networks as optimization over functions, we can consider Recurrent Neural Networks as "optimization over programs." Recurrent neural networks are well suited for modeling functions for which the input and/or output is composed of vectors that involve a time dependency between the values. Recurrent neural networks model the time aspect of data by creating cycles in the network (hence, the "recurrent" part of the name).

Lost in time

Many classification tools (support vector machines, logistic regression, and regular feed-forward networks) have been applied successfully without modeling the time dimension, assuming independence. Other variations of these tools capture the time dynamic by modeling a sliding window of the input (e.g., the previous, current, and next input together as a single input vector).

A drawback of these tools is that assuming independence in the time connection between model inputs does not allow our model to capture long-range time dependencies. Sliding window techniques have a limited window width and will fail to capture any effects larger than the fixed window size. A great example of this is modeling how conversations work and having machines understand how to reply in a coherent fashion as the conversation evolves over time. A well-trained Recurrent Neural Network could compete in Alan Turing's famed *Turing Test*, for instance, which attempts to fool a human into thinking he is speaking with a real person.

Temporal feedback and loops in connections

Recurrent Neural Networks can have loops in the connections. This allows them to model temporal behavior gain accuracy in domains such as time-series, language, audio, and text.



A Note About Connections from Output to Hidden Layers

In practice, we see this connectivity scheme less often than others. DL4J does not use this scheme either. We felt it important to note this variant, however, for context in the material. Most often we see connections between neurons in one time step to the next in each recurrent layer.

Data in these domains are inherently ordered and context sensitive where later values depend on previous ones. The wiring for a Recurrent Neural Network allows for feedback in ways that make it possible to capture these temporal effects. We primarily see Recurrent Neural Network architectures applied in the time-series domains for applications.

A Recurrent Neural Network includes a feedback loop that it uses to learn from sequences, including sequences of varying lengths. Recurrent Neural Networks contain an extra parameter matrix for the connections between time-steps, which are used/trained to capture the temporal relationships in the data.

Recurrent Neural Networks are trained to generate sequences, in which the output at each time-step is based on both the current input and the input at all previous time steps. Normal Recurrent Neural Networks compute a gradient with an algorithm called *backpropagation through time* (BPTT). We go into detail about BPTT later in the chapter.

Sequences and time-series data

We find sequential data in many problem domains in industry for which our model needs to output a sequence of vectors:

- **Image captioning**
- Speech synthesis²⁴
- Music generation²⁵
- Playing video games
- **Language modeling**
- **Character-level text generation models**

In other domains, we need a sequence of input vectors:

- Time-series prediction
- Video analysis
- Music information retrieval

And then we have domains that require both a series of input and output vectors:

- Natural language translation²⁶
- Engaging in dialogue
- Robotic control

Recurrent Neural Networks contrast with other deep networks in what type of input they can model (nonfixed input):

- Nonfixed computation steps
- Nonfixed output size
- It can operate over sequences of vectors, such as frames of video

An important facet of Recurrent Neural Networks is how we can work with input and output in unique ways.

Understanding model input and output

Traditional machine learning operates on the concept of a single fixed-sized input vector. In traditional modeling activities, we typically would see an input-to-output relationship of fixed input size to fixed output size.

This is commonly the pattern for modeling in building classifiers for image classification or classifying columnar data.

Recurrent Neural Networks change this input dynamic to include multiple input vectors, one for each time-step, and each vector can have many columns. The following

²⁴ Graves and Jaitly. 2014. “Towards End-to-End Speech Recognition with Recurrent Neural Network.”

²⁵ Nayebi and Vitelli. 2015. “GRUV: Algorithmic Music Generation using Recurrent Neural Networks.”

²⁶ Sutskever, Vinyals, and Le. 2014. “Sequence to Sequence Learning with Neural Networks.”

list gives examples of how Recurrent Neural Networks operate on sequences of input and output vectors:

- One-to-many: sequence output. For example, image captioning takes an image and outputs a sequence of words.
- Many-to-one: sequence input. For example, sentiment analysis where a given sentence is input.
- Many-to-many: For example, video classification: label each frame.

Now that we've looked at variations of input and output data, let's take a look at how these input data are represented.

3D Volumetric Input

Input into Recurrent Neural Networks involves more dimensions than standard machine learning modeling input. This is similar conceptually to CNNs. We have three dimensions for the input:

1. Mini-batch Size
2. Number of columns in our vector per time-step
3. Number of time-steps

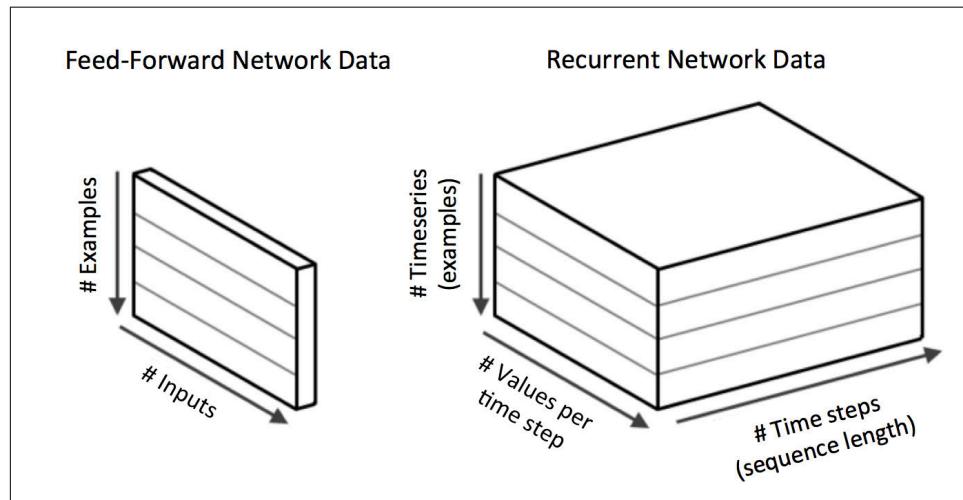


Figure 4-17. Normal input vectors compared to recurrent neural networks input

Mini-batch size is the number of input records (collections of time-series points for a single source entity) we want to model per batch. The number of columns matches up to the traditional feature column count found in a normal input vector. The number of time-steps is how we represent the change in the input vector over time. This is the time-series aspect to the input data. In the terminology from the previous section, we'd consider any time-step count above 1 to be “many-to-” in terms of input and output architecture.

Uneven time-series and masking

We previously described how with Recurrent Neural Network input we have the concept of time-steps in addition to features in our input vector. [Figure 4-18](#) provides a visual representation.

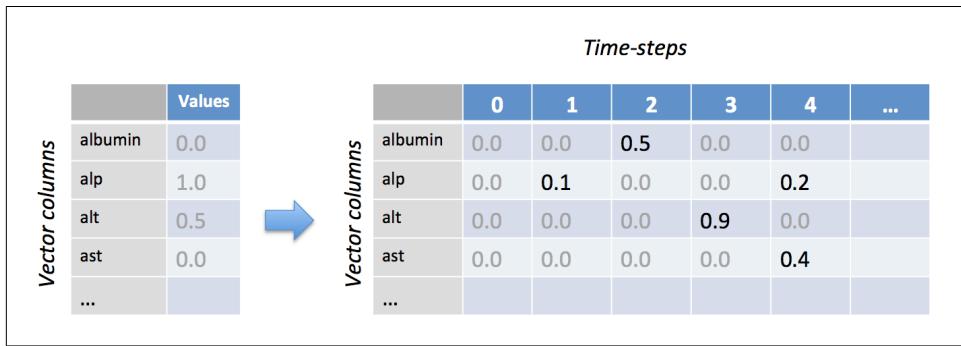


Figure 4-18. The time-step aspect of Recurrent Neural Network input

Every column value likely will not occur at every time-step, especially for the case in which we're mixing descriptor data (e.g., columns from a static database table) with time-series data (e.g., measurements of an ICU patient's heartrate every minute). For cases in which we have “jagged” time-step values, we need to use *masking* to let DL4J know where our real data is located in the vector. We do this by providing an extra matrix for the mask indicating the time-steps that have input data for at least one column, as demonstrated in [Figure 4-19](#).

	0	1	2	3	4	...
albumin	0.0	0.0	0.5	0.0	0.0	
alp	0.0	0.1	0.0	0.0	0.0	
alt	0.0	0.0	0.0	0.9	0.0	
ast	0.0	0.0	0.0	0.0	0.0	
...						

Input Mask (only time-steps)	1.0	1.0	1.0	1.0	0.0	0.0
--	------------	------------	------------	------------	-----	-----

Figure 4-19. Masking specific time-steps

Chapter 3 provides real code examples that show how we set up these masks.

Why Not Markov Models?

When dealing with the time dimension in our models, we naturally could consider **Markov models** as an option. Markov models are another class of machine learning models that have been widely used for modeling sequences. The limiting factor with Markov models is that as their context window grows in size, eventually these models become computationally impractical for modeling long-range dependencies.

Recurrent Neural Networks (connectionist models) are better than Markov models (and other time-window limited models) because they can capture the long-range time dependencies in the input data. Recurrent Neural Networks accomplish this because their hidden state captures information from an arbitrarily long context window and does not have the limitation of the other techniques. Moreover, the number of states they can model is represented by the hidden layer of nodes, and these states grow exponentially with the number of nodes in the layer. This makes Recurrent Neural Networks exceptional at capturing a lot of time-dimension relevant information across many input vectors.

Recurrent Neural Networks, Hidden Layers, and Number of States

If the input was only binary values (0,1), the network could represent 2^N states, where N is the number of nodes in the hidden layer. If the output were real-valued 64-bit numbers, a single hidden layer of these nodes can represent 2^{64^N} different states.

Training of these networks grows only quadratically with hidden node count, whereas the expressive power of the network grows exponentially with the number hidden nodes.

General Recurrent Neural Network Architecture

Recurrent Neural Networks are a superset of feed-forward neural networks but they add the concept of recurrent connections. These connections (or recurrent edges) span adjacent time-steps (e.g., a previous time-step), giving the model the concept of time. The conventional connections do not contain cycles in recurrent neural networks. However, recurrent connections can form cycles including connections back to the original neurons themselves at future time-steps.

Recurrent Neural Networks architecture and time-steps

At each time-step of sending input through a recurrent network, nodes receiving input along recurrent edges receive input activations from the current input vector and from the hidden nodes in the network's previous state.

The output is computed from the hidden state at the given time-step. The previous input vector at the previous time step can influence the current output at the current time-step through the recurrent connections.

We can chain layers of these specialized recurrent neurons together to build better models. We connect the output of the previous layer to the input of the next layer similar to how we'd connect feed-forward multilayer neural networks.

Vanishing Gradient Problem

Recurrent Neural Networks are known to have issues with the “vanishing gradient problem.” This issue occurs when the gradients become too large or too small and make it difficult to model long-range dependencies (10 time-steps or more) in the structure of the input dataset. The most effective way to get around this issue is to use the LSTM variant of Recurrent Neural Networks which DL4J supports.

LSTM Networks

LSTM networks are the most commonly used variation of Recurrent Neural Networks. LSTM networks were introduced in 1997 by Hochreiter and Schmidhuber.²⁷

The critical component of the LSTM²⁸ is the memory cell and the gates (including the forget gate,²⁹ but also the input gate). The contents of the memory cell are modulated by the input gates and forget gates.³⁰ Assuming that both of these gates are closed, the contents of the memory cell will remain unmodified between one time-step and the next. The gating structure allows information to be retained across many time-steps, and consequently also allows gradients to flow across many time-steps. This allows the LSTM model to overcome the vanishing gradient problem that occurs with most Recurrent Neural Network models.

Properties of LSTM networks

LSTMs are known for the following:

- Better update equations
- Better backpropagation

Here are some example use cases of LSTMs:

- Generating sentences (e.g., character-level language models)
- Classifying time-series
- Speech recognition
- Handwriting recognition
- Polyphonic music modeling

LSTM and Bidirectional Recurrent Neural Networks (BRNN) architectures have shown industry-leading benchmarks in recent years on tasks such as the following:

- Image captioning
- Language translation
- Handwriting recognition

LSTM networks consist of many connected LSTM cells and perform well in how efficient they are during learning.

²⁷ Hochreiter and Schmidhuber. 1997. “Long short-term memory.”

²⁸ Graves. 2012. “Supervised Sequence Labelling with Recurrent Neural Networks.”

²⁹ Gers, Schmidhuber, and Cummins. 1999. “Learning to Forget: Continual Prediction with LSTM.”

³⁰ Graves. 2012. “Supervised Sequence Labelling with Recurrent Neural Networks.”



A Note About Training Complexity in LSTMs

The computational complexity of the forward and backward pass operations scale linearly with the number of time-steps in the input sequence.

In the following sections, we provide an overview of the architecture and components of the LSTM.

LSTM network architecture

To better understand the complex arrangement of connections between units and layers in LSTM networks, let's build on some earlier concepts.

Earlier in this book, we developed the concept of the feed-forward multilayer neural network, as depicted in [Figure 4-20](#).

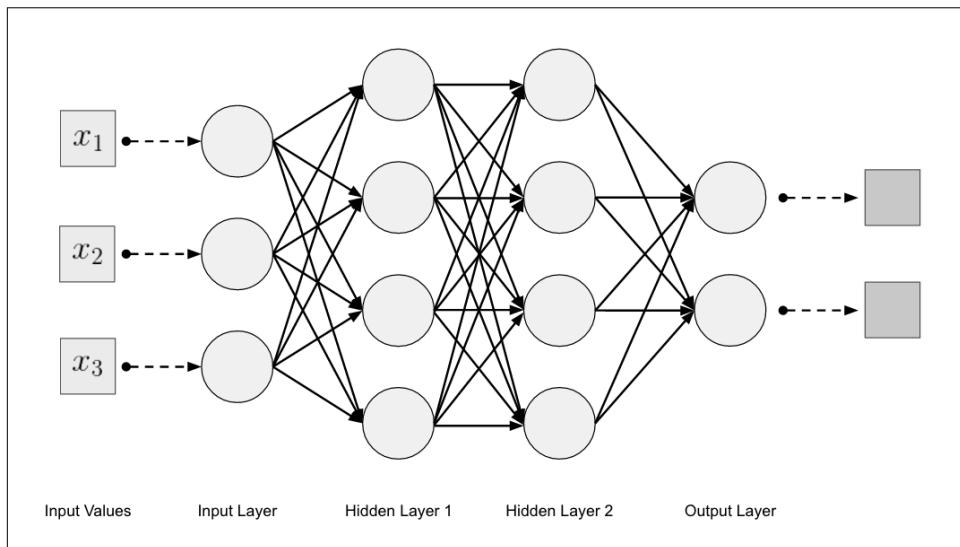


Figure 4-20. Feed-forward multilayer neural network architecture

If we were to show each layer from the network shown in [Figure 4-20](#) as a single node in a flattened representation, it would look like [Figure 4-21](#).

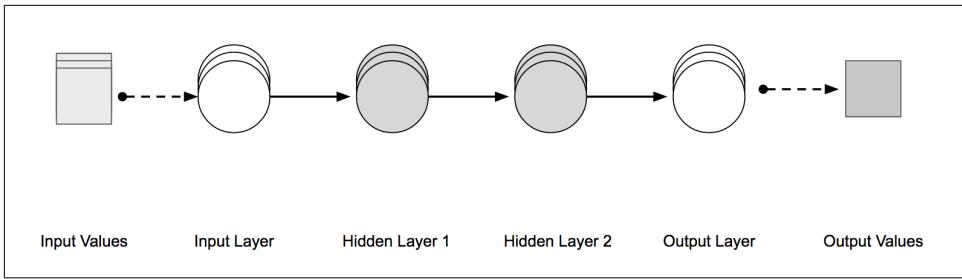


Figure 4-21. Visually feed-forward multilayer network

With Recurrent Neural Networks, we introduce the idea of a type of connection that connects the output of a hidden-layer neuron as an input to the same hidden-layer neuron. With this recurrent connection, we can take input from the previous time-step into the neuron as part of the incoming information.

[Figure 4-22](#) demonstrates that by flattening the network shown in [Figure 4-21](#), we can more easily visualize the recurrent connections in an LSTM.

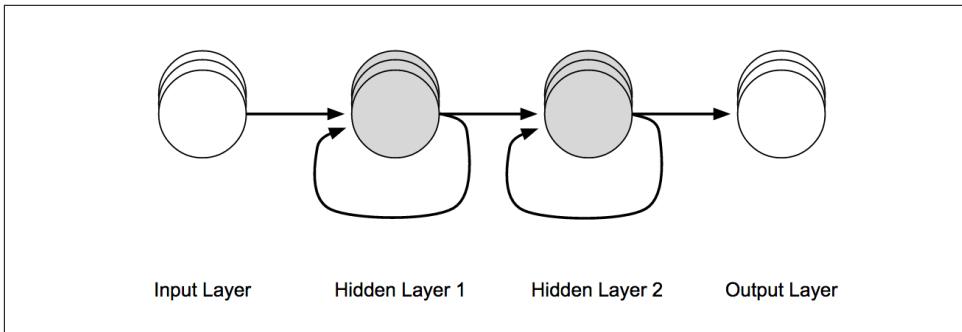


Figure 4-22. Showing recurrent connections on hidden-layer nodes

[Figure 4-23](#) illustrates how we can further visually understand this by “unrolling” the network diagram in [Figure 4-22](#) to show how the information “flows” through the network in a feed-forward manner and “across time.”

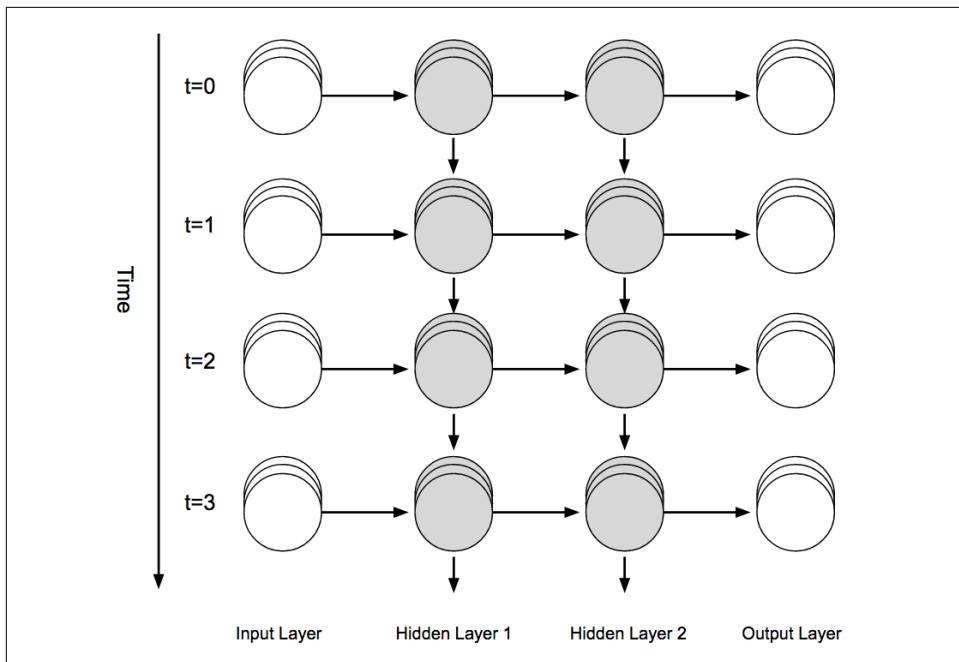


Figure 4-23. Recurrent Neural Network unrolled along the time axis

LSTM networks pass more information across the recurrent connection than the traditional Recurrent Neural Networks, as we'll see in the next section.

LSTM units

The units in the layers of Recurrent Neural Networks are a variation on the classic artificial neuron.

Each LSTM unit has two types of connections:

- Connections from the previous time-step (outputs of those units)
- Connections from the previous layer

The memory cell in an LSTM network is the central concept that allows the network to maintain state over time. The main body of the LSTM unit is referred to as the *LSTM block*, as shown in Figure 4-24.

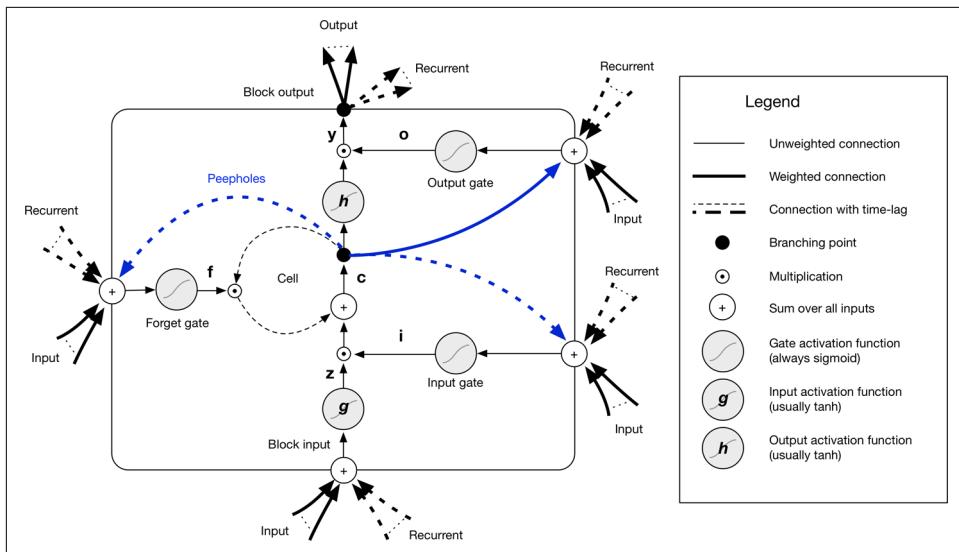


Figure 4-24. LSTM block diagram

Here are the components in an LSTM unit:

- Three gates
 - input gate (input modulation gate)
 - forget gate
 - output gate
- Block input
- Memory cell (the constant error carousel)
- Output activation function
- Peephole connections

There are three gate units, which learn to protect the linear unit from misleading signals:

- The input gate protects the unit from irrelevant input events.
- The forget gate helps the unit forget previous memory contents.
- The output gate exposes the contents of the memory cell (or not) at the output of the LSTM unit.

The output of the LSTM block is recurrent connected back to the block input and all of the gates for the LSTM block. The input, forget, and output gates in an LSTM unit have sigmoid activation functions for $[0, 1]$ restriction. The LSTM block input and output activation function (usually) is a tanh activation function.



A Note About the Forget Gate

An activation output of 1.0 means “remember everything” and activation output of 0.0 means “forget everything.” From a different perspective a better name for the forget gate might be the “remember gate”!

With this in mind we usually initialize the forget gate bias to a large value to enable learning of long-term dependencies (where large value == 1.0 by default in DL4J).

Using the notation from Greff et al.,³¹ Figure 4-25 presents the vector formulas for a LSTM layer forward pass.

$\mathbf{z}^t = g(\mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z)$	<i>block input</i>
$\mathbf{i}^t = \sigma(\mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i)$	<i>input gate</i>
$\mathbf{f}^t = \sigma(\mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f)$	<i>forget gate</i>
$\mathbf{c}^t = \mathbf{i}^t \odot \mathbf{z}^t + \mathbf{f}^t \odot \mathbf{c}^{t-1}$	<i>cell state</i>
$\mathbf{o}^t = \sigma(\mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o)$	<i>output gate</i>
$\mathbf{y}^t = \mathbf{o}^t \odot h(\mathbf{c}^t)$	<i>block output</i>

Figure 4-25. Vector formulas for LSTM layer forward pass

Table 4-1 lists the individual variables in the equations in Figure 4-25.

Table 4-1. Description of variables for LSTM vector formulas

Variable name	Description
x^t	Input vector at time t
\mathbf{W}	Rectangular input weight matrices
\mathbf{R}	Square recurrent weight matrices
\mathbf{p}	Peephole weight vectors
\mathbf{b}	Bias vectors

The self-recurrent connection has a fixed weight of 1.0 (except when modulated) to overcome issues with vanishing gradients. This core unit enables LSTM units to dis-

³¹ Greff et al. 2015. “LSTM: A Search Space Odyssey.”

cover longer-range events in sequences. These events can span up to 1,000 discrete time-steps compared to older recurrent architectures, which could model events across around only 10 time-steps.



For More Variants of LSTMs...

Check out the paper “[LSTM: A Search Space Odyssey](#).”

Gated Recurrent Units (GRU)

Another recurrent unit similar to the LSTM is the Gated Recurrent Unit (GRU).³² The GRU has a reset gate and an update gate, similar to the forget/input gates in the LSTM unit. The major difference is that the GRU fully exposes its memory content using only leaky integration (but with an adaptive time constant controlled by the update gate). The GRU was inspired by the LSTM unit but is considered simpler to compute and implement.

LSTM layers

A basic layer accepts an input vector x (nonfixed) and gives output y . The output y is influenced by the input x and the history of all inputs. The layer is influenced by the history of inputs through the recurrent connections. The RNN has some internal state that is updated every time we input a vector to the layer. The state consists of a single hidden vector.

Training

LSTM networks use supervised learning to update the weights in the network. They train on one input vector at a time in a sequence of vectors. Vectors are real-valued and become sequences of activations of the input nodes. Every noninput unit computes its current activation at any given time-step. This activation value is computed as the nonlinear function of the weighted sum of the activations of all units from which it receives connections.

For each input vector in the sequence of input, the error is equal to the sum of the deviations of all target signals from corresponding activations computed by the network. Let’s now take a look at the BPTT variant of backpropagation used with Recurrent Neural Networks, including LSTMs.

³² Cho et al. 2014. “[Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation](#).”

BPTT and truncated BPTT

Recurrent Neural Network training can be computationally expensive. The traditional option is to use BPTT.



BPTT is fundamentally the same as standard backpropagation: we apply the chain rule to work out the derivatives (gradients) based on the connection structure of the network. It's *through time* in the sense that some of those gradients/error signals will also flow backward from future time-steps to current time-steps, not just from the layer above (as occurs in standard backpropagation).

When our recurrent network is dealing with long sequences with many time-steps, we recommend alternatively using truncated BPTT. Truncated BPTT reduces the computational complexity of each parameter update in a Recurrent Neural Network.

Recurrent Neural Networks and Backpropagation

Computing the gradient for a Recurrent Neural Network on a sequence of length 1,000 has the same computational cost as doing a forward- and backward-pass on a multilayer perceptron network that has 1,000 layers.

Performing more frequent parameter updates speeds up recurrent neural network training. We recommend truncated BPTT when your input sequences are more than a few hundred time-steps.

To better understand the concepts involved with truncated BPTT, let's consider what happens when we train a network with time-series input of length 12 (time-steps). In this scenario, we need to do a forward pass of 12 steps and then calculate the error of the network. Then, do a backward pass of the 12 time-steps, as demonstrated in Figure 4-26.

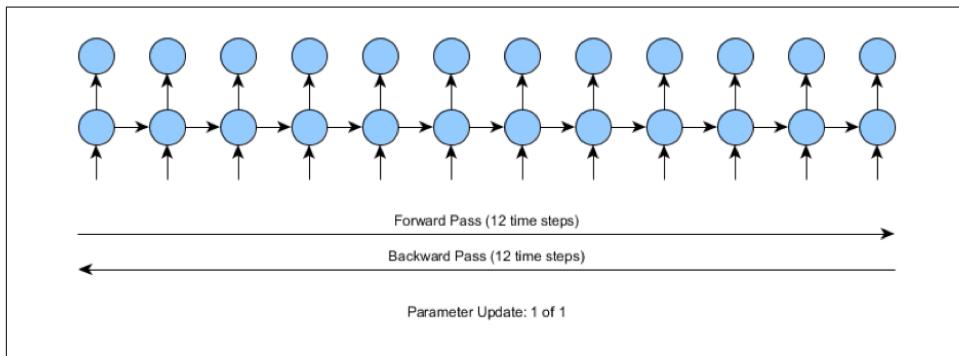


Figure 4-26. Standard BPTT

In the figure, the 12 time-steps are not that difficult for the training process. As we move into models that train on time-series data of a few hundred steps or more, we find training to be more difficult. If the number of steps in the time-series input were 1,000 steps, the standard backpropagation training would require 1,000 time-steps for each forward and backward pass (for each individual parameter update). This quickly becomes computationally expensive and is why we look at alternative training methods such as BPTT and truncated BPTT.

Truncated BPTT separates the forward and backward passes into smaller operations. The truncated BPTT shown in Figure 4-27 takes a smaller forward pass, makes an equally small backward pass, and then updates the intended parameter. This smaller pass size is a hyperparameter configured by the user. In the figure we can see that the truncated BPTT pass size is four time-steps.

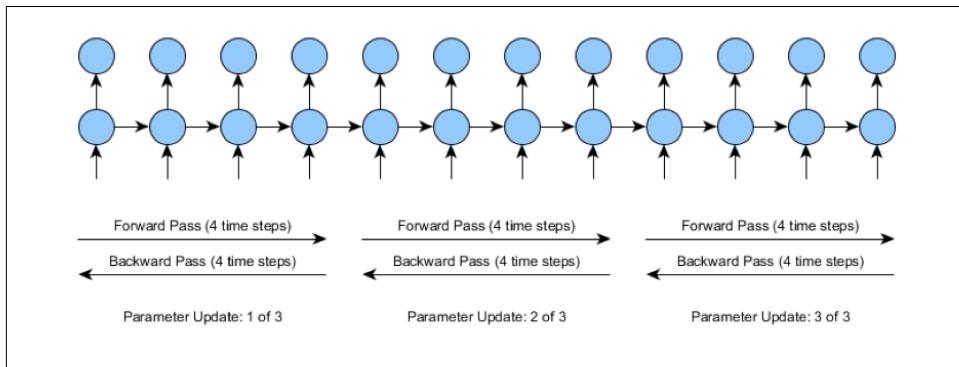


Figure 4-27. Truncated BPTT

Truncated BPTT is considered the current most practical method for training Recurrent Neural Networks. With truncated BPTT, we can capture long-term dependencies with less computational burden than regular BPTT.

The overall complexity for standard BPTT and truncated BPTT is similar and requires the same number of time-steps during training. However, we get more parameter updates for about the same amount of computational effort (although there is some overhead for each parameter update). As with all approximations, we see a slight downside when using truncated BPTT; the length of the dependencies learned in truncated BPTT can end up being shorter than the full BPTT algorithm's results. In practice, this speed trade-off is generally worth it as long as truncated BPTT lengths are set correctly.

Domain-Specific Applications and Blended Networks

As mentioned earlier, Recurrent Neural Networks perform many domain-specific applications such as speech transcription to text, machine translation, and generation of handwritten text. Recurrent Neural Networks have proven to be adept in the world of computer vision and can do the following:

- Frame-level video analysis³³
- Image captioning
- Video captioning³⁴
- Visual question answering³⁵

Another emerging area of research in computer vision with Recurrent Neural Networks is a network that can extract information from an image by processing only a small region at a time and is called Recurrent Models of Visual Attention.³⁶ These models are effective at working with images that are cluttered with multiple objects and are more difficult for CNNs to classify. These networks blend together CNNs for raw perception and Recurrent Neural Networks for the time-domain modeling.

Another hybrid CNN + RNN network of note is the work by Andrej Karpathy and Li Fei-Fei, in which the network generates natural-language descriptions of images and their regions.³⁷ The model is able to generate captions for images by using datasets of images and their corresponding sentence, as depicted in [Figure 4-28](#).

³³ Srivastava, Mansimov, and Salakhutdinov. 2015. “Unsupervised Learning of Video Representations using LSTMs.”

³⁴ Venugopalan et al. 2014. “Translating Videos to Natural Language Using Deep Recurrent Neural Networks.”

³⁵ Wu et al. 2016. “Image Captioning and Visual Question Answering Based on Attributes and External Knowledge.”

³⁶ Mnih et al. 2014. “Recurrent Models of Visual Attention.”

³⁷ Karpathy and Fei-Fei. 2014. “Deep Visual-Semantic Alignments for Generating Image Descriptions.”

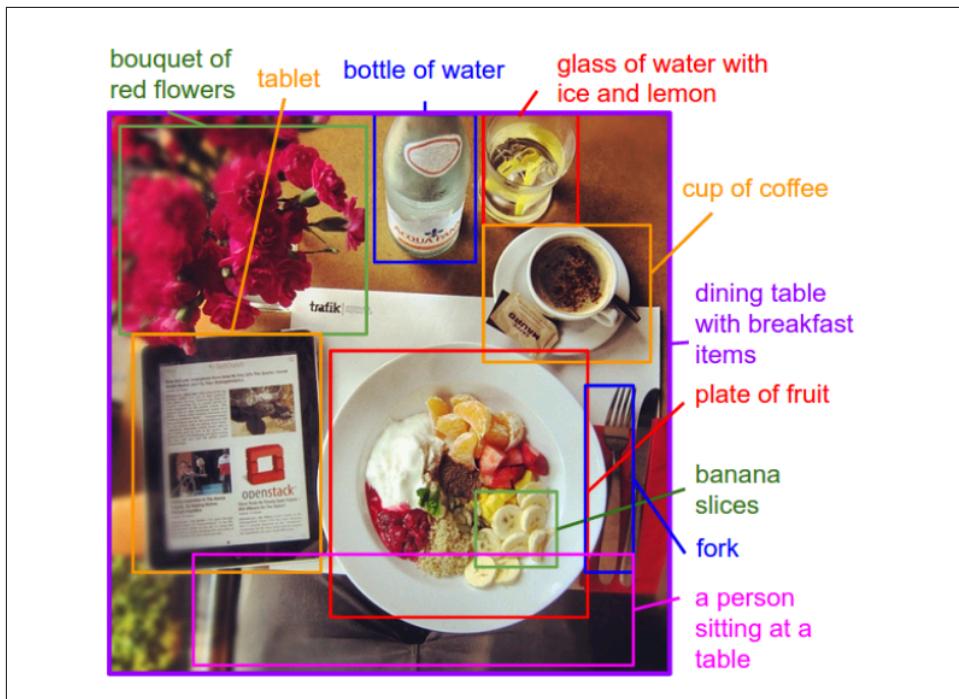


Figure 4-28. Labeling images with a blended CNN/Recurrent Neural Network³⁸

This type of network is a combination of a CNN and a BRNN.

Recursive Neural Networks

Recursive Neural Networks, like Recurrent Neural Networks, can deal with variable length input. The primary difference is that Recurrent Neural Networks have the ability to model the hierarchical structures in the training dataset. Images commonly have a scene composed of many objects. Deconstructing scenes is often a problem domain of interest yet is nontrivial. The recursive nature of this deconstruction challenges us to not only identify the objects in the scene, but also how the objects relate to form the scene.

Network Architecture

A Recursive Neural Network architecture is composed of a shared-weight matrix and a binary tree structure that allows the recursive network to learn varying sequences of

³⁸ The image is from Karpathy and Fei-Fei. 2014. “Deep Visual-Semantic Alignments for Generating Image Descriptions.”

words or parts of an image. It is useful as a sentence and scene parser. Recursive Neural Networks use a variation of backpropagation called *backpropagation through structure* (BPTS). The feed-forward pass happens bottom-up, and backpropagation is top-down. Think of the objective as the top of the tree, whereas the inputs are the bottom.

Varieties of Recursive Neural Networks

Recursive Neural Networks come in a few varieties. One is the recursive autoencoder. Just like its feed-forward cousin, recursive autoencoders learn how to reconstruct the input. In the case of NLP, it learns how to reconstruct contexts. A semisupervised recursive autoencoder learns the likelihood of certain labels in each context.

Another variation of this is a supervised neural network, called a Recursive Neural Tensor Network, which computes a supervised objective at each node of the tree. The tensor part of this means that it calculates the gradient a little differently, factoring in more information at each node by taking advantage of another dimension of information using a tensor (a matrix of three or more dimensions).

Applications of Recursive Neural Networks

Both Recursive and Recurrent Neural Networks share many of the same use cases. Recurrent Neural Networks are traditionally used in NLP because of their ties to binary trees, contexts, and natural-language-based parsers. For example, constituency parsers are able to break up a sentence into a binary tree, segmenting it by the linguistic properties of the sentence. In the case of Recursive Neural Networks, it is a constraint that we use a parser that builds the tree structure (typically constituency parsing).

Recursive Neural Networks can recover both granular structure and higher-level hierarchical structure in datasets such as images or sentences. Applications for recursive neural networks include the following:

- Image scene decomposition
- NLP
- Audio-to-text transcription

Two specific network configurations we see in practice are recursive autoencoders and recursive neural tensors. We use recursive autoencoders to break up sentences into segments for NLP. We use recursive neural tensors to break up an image into its composing objects and semantically label the objects in the scene.

Recurrent Neural Networks tend to be faster to train, thus we typically use them in more temporal applications, but they have been shown to work well in NLP-based domains such as sentiment analysis, as well.

Summary and Discussion

In this chapter, we reviewed the more recent major architectures in deep learning. In summary, to group and summarize these architectures, we note the following:

- To generate data (e.g., images, audio, or text), we'd use:
 - GANs
 - VAEs
 - Recurrent Neural Networks
- To model images, we'd likely use:
 - CNNs
 - DBNs
- To model sequence data, we'd likely use:
 - Recurrent Neural Networks/LSTMs

In the chapters that follow, we present real-world code examples of most of these networks and cover considerations of training and tuning for different kinds of neural networks. In [Chapter 5](#) we see how these concepts come together in API examples in which we see the DL4J deep learning library in action. Before we move on to some more examples, let's discuss a few topics that come up frequently in the context of deep learning.

Will Deep Learning Make Other Algorithms Obsolete?

The debate around deep learning making other modeling algorithms obsolete comes up many times on internet message boards. The answer today is “no” because for many simpler machine learning applications, we see far simpler algorithms work just fine for the required model accuracy. Models like logistic regression are also easier to work with, so we need to gauge the level of effort against the required accuracy in the domain when making this decision. However, deep learning algorithms tend to perform well when we understand little about the applied domain and struggle to do advanced handcrafted feature construction.

Different Problems Have Different Best Methods

Machine learning on the whole is about applying the correct approach in the appropriate situation. We're not to the point yet where a single technique dominates the landscape so we need to evaluate the problem space and data each time we're looking for the best model to apply. This is based on the “no free lunch theorem.”



No Free Lunch Theorem

The “no free lunch theorem” states that there is no one model that works for every problem. The assumptions for a great model for one problem might not hold for another problem. It is common in machine learning to try multiple models and find the one that works best for a particular problem.

Every machine learning method has some bias and variance. The closer our model is to the true underlying model the better we can do on average with our learning algorithm.

Another way to understand this is to consider it from the viewpoint of a practical example. If the data is clearly linear, as seen in visualizations, would you try and fit the data with a nonlinear model (e.g., a multilayer perceptron)? No, you’d probably approach the problem with something simpler, such as logistic regression. In Kaggle competitions, the method that performs best varies from competition to competition. However, random forests and ensemble methods tend to be the winners when deep learning does not win.

The input dataset size can be another factor in how appropriate deep learning can be for a given problem. Empirical results over the past few years have shown that deep learning provides the best predictive power when the dataset is large enough. This means deep learning results become better as dataset size increases. Neural networks have larger representational capacity than linear models and are better able to exploit the data. A nice rule of thumb is that a practitioner should be able to train a neural network with at least 5,000 training input labeled examples.

When Do I Need Deep Learning?

To close out this chapter, we want to leave you with a simple set of rules to help answer the question: does this project need to use deep learning?

When to use deep learning

You should use deep learning when...

- Simpler models (logistic regression) don’t achieve the accuracy level your use case needs
- You have complex pattern matching in images, NLP, or audio to deal with
- You have high dimensionality data
- You have the dimension of time in your vectors (sequences)

When to stick with traditional machine learning

You should use a traditional machine learning model when...

- You have high-quality, low-dimensional data; for example, columnar data from a database export
- You're not trying to find complex patterns in image data

You'll achieve poor results from both methods when the data is incomplete and/or of poor quality.

CHAPTER 5

Building Deep Networks

*Now is no time
to think of what you do not have.
Think of what you can do
with that there is*

—Ernest Hemingway, *The Old Man and the Sea*

In this chapter, we take a look at the suite of tools that are available in the DL4J and some real-world examples that you can use in your own projects. We begin by reviewing how we map specific deep networks to the appropriate problem. We end the chapter with a deep dive into many of the core examples that come with the library.



For information about DL4J installation and support, refer to [Appendix G](#).

Matching Deep Networks to the Right Problem

A theme we introduced in [Chapter 4](#) was how deep learning is about designing the network architecture to match the problem as opposed to hand-engineering features in the input data. In this chapter, we see examples of deep networks matched to specific types of problems. For the purposes of this chapter, we'll call out applications specifically for the following:

- Modeling columnar data
- Modeling image data

- Modeling sequence/time-series data
- Natural Language Processing applications

The applications in this chapter illustrate the concepts of deep networks we've been building since [Chapter 1](#). Although we don't have an example of every architecture referenced in [Chapter 4](#), we have compiled a set of examples to illuminate core concepts in deep learning such that you could easily extend most any example toward a new purpose. Let's begin by reviewing the data types in the context of the correct network architecture pairings, as introduced previously in the book.



Finding the Examples Online

We've created a fork of the examples in the [GitHub repository](#). Even though a project such as DL4J will evolve over time, we want to provide the readers with a snapshot of the code that will always work with this version of the book.

Columnar Data and Multilayer Perceptrons

Common columnar data has a static structure to it, and is best modeled in DL4J by a classic multilayer perceptron neural network. These problems may benefit from minor feature engineering, but we can often let the network find the best weights on the dataset on its own. Hyperparameter tuning is one of the main challenges when modeling with multilayer perceptrons. We'll cover a variety of techniques to help guide your hyperparameter selection in [Chapter 6](#).

Images and Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have proven to be adept at finding structure in raw image data. Historically, the image modeling field has been dominated by excessive amounts of preprocessing techniques to get the input images aligned and transformed into a form that modeling techniques could better handle. Slight variations in rotation or scale made image processing a hard task. CNNs have made it possible to let the network handle the raw image data and let the practitioner focus on adjusting the network architecture. We'll see an example of classifying handwritten digits by using CNNs later in the chapter.



Convolutional Neural Networks and Raw Image Data

Columnar data, through painstaking effort, shows up with all of the features aligned to a specific column. Historically, machine learning modeling has been relatively brittle when features are not in the correct “spots” the model expects them to be in.

Objects in images rarely appear in the spot we want or expect them to be and provide nontrivial challenges in the feature extraction phase for classic machine learning techniques. A major strength of CNNs is their ability to take raw image data in which the objects in the scene are in arbitrary poses and adeptly recognize the features.

Evolution of CNN Applications

CNN applications continue to evolve at breakneck speed. While we recommend the practitioner start with CNNs for image modeling problems, applications of these networks have begun to appear for text modeling problems, as well. Here are some of these applications:

- Machine translation¹
- Sentence classification²
- Sentiment analysis³

In the same way that we slide windows over regions of an image, CNNs can slide windows over sequences of characters. This feature location-invariance aspect makes CNNs a powerful tool in the deep network toolbox. In this chapter, we focus on an example of applying a CNN to image data. To be clear, though, you should take this not as a limitation of the network architecture, but a fundamental starting point to be investigated further.

Time-series Sequences and Recurrent Neural Networks

When working with sequential data such as web log data, Recurrent Neural Networks are a good fit. When we have sequential data with a timestamp for every sample, we consider this data to be time-series data.

Time-series and sequential data represent a waveform when graphed on a two-dimensional plot. Often, we are looking for areas of the graph that make particular

¹ Kalchbrenner et al. 2016. [“Neural Machine Translation in Linear Time.”](#)

² Kim. 2014. [“Convolutional Neural Networks for Sentence Classification.”](#)

³ Nogueira dos Santos and Gatti. 2014. [“Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts.”](#)

patterns over time. This is in contrast to how image data does not have a time domain but is arranged in a two-dimensional grid to form a picture. However, in both time-series and images, we are looking for specific objects that might occur in the data. These objects might be scaled in size and usually do not appear in the same place every time in the data, providing us with a challenge.

Recurrent Neural Networks have evolved from multilayer perceptrons to better model the time domain for time-series data. As we saw in [Chapter 4](#), Recurrent Neural Networks are better able to model the time domain by allowing a sequence of input vectors to be treated as a single logical input for a Recurrent Neural Network model.



Sensors, Logs, and Other Measurements

When dealing with any data that has a source that takes measurements over time, use a Recurrent Neural Network to better model the change of the data within a span of time.

You can use Recurrent Neural Networks for classification, regression, and generating novel output. We'll see examples of both classification and novel generation later in the chapter.

Playing the Devil's Advocate for Model Choice

Sometimes, we hear the argument that a practitioner has found great success with other techniques such as random forests. However, in the world of sequential data, this is generally not a good path except in specific situations.

Some practitioners argue that they can't apply random forests directly to time-series/sequential data in the same way that they can with Recurrent Neural Networks because the time-series would need to be converted into some kind of flat vector representation⁴ (e.g., extract features) before any modeling was performed.

Typically we've found the debate in which a deep learning model is compared against a powerful model + hand-engineered features to be a trap; we say this because the practitioner who's committed to engineered features can keep trying to invent better features until he finds all of the tricks in a particular dataset (given a long enough timeline).

The real advantages of deep learning for time-series (and really any problem) basically amount to the following:

⁴ You can find an example in Shieh and Keogh. 2008. “[iSAX: Indexing and Mining Terabyte Sized Time Series](#).”

If we go with the tailored-features/algorithms approach, we find ourselves in a situation where we need to write new code for a dozen completely different algorithms/models and then apply all of them to each new problem/dataset. If we were in the pure software engineering world, we'd consider this situation to be *technical debt*. Historically, we've allowed this to be status quo for machine learning.

If we're starting from a place where we don't already know the appropriate features, with current state of the art in deep learning, it's easier to semiautomate hyperparameter tuning of neural networks than it is to semiautomate feature engineering. A well-engineered deep learning system (including hyperparameter tuning) can be efficiently reused and expected to perform pretty well across a variety of problems.

When we're dealing with variable-length time-series with unknown but probably long dependencies, this is the case for which Recurrent Neural Networks are especially warranted. If the input is a fixed-length sequence (i.e., all time-series are the same lengths), you might as well use an multilayer perceptron or CNN.

However, if you want to allow arbitrarily long sequence inputs that can (hypothetically) take the full input data history into account, a Recurrent Neural Network (or a carefully crafted Markov model with a large hidden-state space) is basically your only option.

Using Hybrid Networks

When we see data that is the combination of both time and image (e.g., video), we use a special hybrid network of Long Short-Term Memory (LSTM) and convolutional layers. We recommend this hybrid because LSTM networks are able to capture the temporal aspect of changing in the images over time in video, and convolutional layers are able to capture the structure in the frame data themselves.

The DL4J Suite of Tools

DL4J is a group of deep learning tools that is packaged together as a suite to perform functions such as the following:

- Integration
- Vectorization
- Modeling
- Evaluation

These tools are designed to work on multiple platforms and execute in serial or in parallel. DL4J was designed from the beginning with modern execution platforms in mind and does not suffer from parallelization issues that other machine learning libraries have experienced in the past decade. The DL4J community also has paid attention to needs beyond basic modeling to provide better integration with plat-

forms such as Spark and Hadoop Distributed File System (HDFS). The DL4J project also has robust vectorization capabilities with the introduction of DataVec as a first-class citizen in the suite.

DL4J is focused on enterprise-grade functionality and is targeted at practitioners who need a Java Virtual Machine (JVM) option in deep learning but also want the speed of C++ and the power of Spark for parallel computation. Let's take a quick look at each of these facets of the DL4J suite before we dive into how to set up the tools on your laptop.

Vectorization and DataVec

Because neural networks can train only on vectors, vectorizing data is a necessary preprocessing step. The DL4J suite includes a library called DataVec that allows us to quickly create vectorized data appropriate for modeling with the core DL4J library tools. DataVec, as we'll see later in the book, works in local mode and also in parallel execution mode on Apache Spark.

Runtimes and ND4J

ND4J is a library for scientific computing on the JVM. Its syntax emulates that of Numpy and MATLAB. It exposes n -dimensional arrays for Java that can be used in linear algebra and large-scale matrix manipulation. ND4J presents a clean numerical façade that lets users swap backends from native to graphics processing units (GPU) without having to change the implementation code. We can write an implementation of the linear algebra once and then specify a backend for ND4J in Maven. The most common backends for ND4J include x86 and jcublas. Development in the DL4J project will eventually include OpenCL and Power8. ND4J includes both Java and Scala APIs, providing programmers of JVM-based languages with a familiar environment. The ND4J APIs are intended to offer the functionality and, where possible, the concision and ease of Numpy.

The DL4J Java API allows programmers to configure their own extensible neural networks within a composable framework, adjusting hyperparameters as needed. It provides a space for building, tuning, evaluating, and loading data into neural networks. ND4J's APIs provide essential linear algebra, calculus, and signal processing functions in a Numpy-like environment (which developers can deploy on distributed runtimes over multiple CPUs or GPUs). Together, the Java API and the ND4J backend provide the programmatic accessibility and top-end speed the practitioner needs for today's enterprise deep learning applications.

ND4J and the need for speed

There is always a need for faster computation in the world of deep learning. ND4J provides faster linear algebra execution in three key ways:

- Use of JavaCPP
- CPU backends
- GPU backends

We provide a few notes here on how improvements in these key areas help ND4J accelerate training deep models.

JavaCPP.

- Autogenerates JNI bindings for C++
- Allows for easy maintenance and deployment of C++ binaries in Java

CPU backends.

- OpenMP (multithreading within native operations)
- OpenBLAS or MKL (BLAS operations)
- SIMD extensions

GPU backends.

- DL4J supports Cuda 7.5 (+cuBLAS) at the moment, and will support 8.0 as soon as it comes out.
- Uses cuDNN, as well



Setting Up GPU Support

For more detailed information on setting up GPUs with DL4J, check out the guide in [Appendix I](#).

Benchmarking ND4J and DL4J

To see how DL4J compares to other major deep learning libraries, visit the [GitHub project](#).

Today, DL4J is competitive with the major libraries, as illustrated in [Table 5-1](#).

Table 5-1. LeNet Example w/cuDNN

Package	CPU	GPU	Multi	Accuracy
Dl4j	20m08s	3m13s	1m18s	~99.0%
Caffe	19m52s	53s	1m14s	~99.0%
Tensorflow	5m15s	1m44s	2m44s	~98.6%
Torch	18m03s	6m25s	3m50s	~98.3%

The DL4J community continues to push the performance boundaries of ND4J and DL4J. Feel free to check out the open benchmarks at any time to replicate the results.

Basic Concepts of the DL4J API

In this section, we quickly list core techniques that most DL4J examples use. For expanded coverage on using the API check out [Appendix E](#) or the DL4J online documentation.

Loading and Saving Models

In this section, we highlight a few common things you might want to do with the DL4J API.

Writing a trained model to disk

We use the `ModelSerializer` class to write a model's architecture and parameters to disk. Here's a sample code snippet that shows how:

```
BufferedOutputStream stream = new BufferedOutputStream(...);
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

This works for both writing to local disk and to another filesystem such as HDFS.

Writing to HDFS. To write to HDFS we just need to use the correct `Path` class (`org.apache.hadoop.fs.Path`) and a correctly formatted HDFS path string, as shown here:

```
Path modelPath = new Path( hdfsPathToSaveModel );
BufferedOutputStream stream = new BufferedOutputStream( os );
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

HDFS filepaths commonly look like:

```
hdfs:///path/to/my/file.txt
```

Reading a saved model from disk

To load a previously saved DL4J model from disk, we again would use the `ModelSerializer` class, as demonstrated in the following code snippet:

```
InputStream stream = ...
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

Reading from HDFS. If you want to load the model directly from HDFS, use the same API call, but create the `InputStream` by using the Hadoop `FileSystem` class, as shown here:

```
org.apache.hadoop.conf.Configuration hadoopConfig = new org.apache.hadoop.conf.Configuration();
FileSystem hdfs = FileSystem.get(hadoopConfig);
InputStream stream = hdfs.open( new Path( hdfsPathToSavedModelFile ) );
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

Getting Input for the Model

DL4J works specifically with NDArrays as data structures for training and scoring models. We often see NDArrays for input and output for a model paired in a `DataSet` object. For more information on using the `DataSet` class check out the Appendix on ND4J.



Vectorizing Data

To get ideas for how to convert raw data into NDArray vectors for use with DL4J models check out [Appendix F](#) and the [Appendix E](#).

Loading data during training

Getting data into the training and test workflows of DL4J modeling projects is based around parsing the data from file formats with `RecordReaders` and then batching this data into mini-batches.

RecordReaders

This class is tasked with parsing a specific vector from a file format and converting the data values into a standardized NDArray.

DataSetIterator

This class works with `RecordReaders` to take the produced NDArray for each record and create mini-batches of NDArrays for training.



Loading Data on Spark

Spark data-loading patterns are similar but use special classes to load data on Spark and HDFS. We cover those specifically in [Chapter 9](#).

Setting Up Model Architecture

Each neural network we model with DL4J requires you to configure a specific architecture. We've discussed the major architectures previously in this book, and these concepts match up directly to how you set up network architectures in DL4J.

Building layer-oriented architectures

A `NeuralNetConfiguration` object is the fundamental object used to construct layers in DL4J neural networks. Many single layers combined constitute a deep neural network, as shown in the following example:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
```

```

    .iterations(1)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(learningRate)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.RELU)
        .build())
    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();

```

We design the intended network architecture with this configuration class by adding layers and configuring each layer specifically.

Hyperparameters

The `NeuralNetConfiguration` object also allows us to set many different hyperparameters relevant to training, such as the following:

```

    .learningRate(learningRate)
    .updater(Updater.NESTEROVS).momentum(0.9)

```

In the preceding code snippet, we can see hyperparameters such as the learning rate, an updater, and a momentum value being set for a given `MultiLayerConfiguration` object. This allows you to control how a network is trained. We'll look at strategies for setting hyperparameters for different network architectures later in the book.

Training and Evaluation

After we have our network architecture configured and our data loaded through `RecordReaders` and `Iterators`, we need to train our model. We want to set up multiple epochs and call the `.fit()` method on the network passing our training data in as the parameter, as demonstrated here:

```
model.fit( trainingDataIter );
```

This method loops through all of the training records in the dataset referenced by the iterator and returns after the network has completed the loop.

Each training epoch is considered a full pass over the input dataset. In the following code snippet, we can see a model calling the `.fit()` method on an input dataset iterator:

```

for ( int n = 0; n < nEpochs; n++ ) {
    model.fit( trainingDataIter );
}

```

We'll commonly see this pattern for managing epoch loops in the examples later in the chapter.

Making a prediction

To better understand making predictions, you should check out [Appendix E](#).

Training, validation, and test data

It's considered a best practice to not only split out the training data into training and test splits, but validation splits, as well. We use validation splits to guide our efforts in possibly stopping training early.

Modeling CSV Data with Multilayer Perceptron Networks

Getting started with DL4J can seem complex for the new deep learning practitioner, so we'll begin with building a multilayer perceptron model. This will allow you to see basic usage of the DL4J API in the context of an older neural network architecture that you might recognize. In the accompanying GitHub repository, we have a Java example modeling a synthetic nonlinear dataset⁵ called Saturn. [Example 5-1](#) presents the code from the example:

Example 5-1. Multilayer perceptron example

```
public class MLPClassifierSaturn {  
  
    public static void main(String[] args) throws Exception {  
        Nd4j.ENFORCE_NUMERICAL_STABILITY = true;  
        int batchSize = 50;  
        int seed = 123;  
        double learningRate = 0.005;  
        //Number of epochs (full passes of the data)  
        int nEpochs = 30;  
  
        int numInputs = 2;  
        int numOutputs = 2;  
        int numHiddenNodes = 20;  
  
        final String filenameTrain =  
            new ClassPathResource("/classification/saturn_data_train.csv")  
                .getFile().getPath();  
        final String filenameTest =  
            new ClassPathResource("/classification/saturn_data_eval.csv")  
                .getFile().getPath();  
    }  
}
```

⁵ This is a dataset generated by [Dr. Jason Baldridge](#) to test neural network frameworks for basic functionality.

```

//Load the training data:
RecordReader rr = new CSVRecordReader();
rr.initialize(new FileSplit(new File(filenameTrain)));
DataSetIterator trainIter =
    new RecordReaderDataSetIterator(rr,batchSize,0,2);

//Load the test/evaluation data:
RecordReader rrTest = new CSVRecordReader();
rrTest.initialize(new FileSplit(new File(filenameTest)));
DataSetIterator testIter =
    new RecordReaderDataSetIterator(rrTest,batchSize,0,2);

//log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(1)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(learningRate)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(numInputs)
        .nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.RELU)
        .build())
    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(10)); //Print score every 10
                                                    //parameter updates

for ( int n = 0; n < nEpochs; n++) {
    model.fit( trainIter );
}

System.out.println("Evaluate model....");
Evaluation eval = new Evaluation(numOutputs);
while(testIter.hasNext()){
    DataSet t = testIter.next();
    INDArray features = t.getFeatureMatrix();
    INDArray labels = t.getLabels();
    INDArray predicted = model.output(features,false);

    eval.eval(labels, predicted);
}

```

```

}

System.out.println(eval.stats());
//-----
//Training is complete. Code that follows is for plotting the data &
//predictions only

double xMin = -15;
double xMax = 15;
double yMin = -15;
double yMax = 15;

//Let's evaluate the predictions at every point in the x/y input space, and
//plot this in the background
int nPointsPerAxis = 100;
double[][] evalPoints = new double[nPointsPerAxis*nPointsPerAxis][2];
int count = 0;
for( int i=0; i<nPointsPerAxis; i++ ){
    for( int j=0; j<nPointsPerAxis; j++ ){
        double x = i * (xMax-xMin)/(nPointsPerAxis-1) + xMin;
        double y = j * (yMax-yMin)/(nPointsPerAxis-1) + yMin;

        evalPoints[count][0] = x;
        evalPoints[count][1] = y;

        count++;
    }
}

INDArray allXYPoints = Nd4j.create(evalPoints);
INDArray predictionsAtXYPoints = model.output(allXYPoints);

//Get all of the training data in a single array, and plot it:
rr.initialize(new FileSplit(new File(filenameTrain)));
rr.reset();
int nTrainPoints = 500;
trainIter = new RecordReaderDataSetIterator(rr,nTrainPoints,0,2);
DataSet ds = trainIter.next();
PlotUtil.plotTrainingData(ds.getFeatures(), ds.getLabels(), allXYPoints,
    predictionsAtXYPoints, nPointsPerAxis);

//Get test data, run the test data through the network to generate
//predictions, and plot those predictions:
rrTest.initialize(new FileSplit(new File(filenameTest)));
rrTest.reset();
int nTestPoints = 100;
testIter = new RecordReaderDataSetIterator(rrTest,nTestPoints,0,2);
ds = testIter.next();
INDArray testPredicted = model.output(ds.getFeatures());

```

```

        PlotUtil.plotTestData(ds.getFeatures(), ds.getLabels(), testPredicted,
            allXYPoints, predictionsAtXYPoints, nPointsPerAxis);

        System.out.println("*****Example finished*****");
    }

}

```

In the following sections, we explain how the different parts of the code work together to model the Saturn dataset.

Setting Up Input Data

The input data is a nonlinear dataset in the following form:

```

1,-7.1239700674365,-5.05175898010314
0,1.80771566423302,0.770505522143023
1,8.43184823707231,-4.2287794074931
0,0.451276074541732,0.669574142606103
0,1.52519959303934,-0.953055551414968

```

The first column is the label of the row, and columns two and three represent the two independent variable columns. We need to get a reference to the dataset with a DL4J iterator, so load the data by using CSVRecordReader:

```

//Load the training data:
RecordReader rr = new CSVRecordReader();
rr.initialize(new FileSplit(new File(filenameTrain)));
DataSetIterator trainIter = new RecordReaderDataSetIterator(rr,batchSize,0,2);

```

In the iterator, we tell the record reader how many columns our data has and by which column the label is represented.

Determining Network Architecture

We know that we want a basic multilayer perceptron, and we want to set this up (and any DL4J network architecture) by using the MultiLayerConfiguration object, as illustrated here:

```

//log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(1)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(learningRate)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.RELU)
        .build())

```

```

    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(10)); //Print score every 10
//parameter updates

```

In this example, we have two layers:

- DenseLayer
- OutputLayer

Let's look at the specific sections of the network architecture.

General hyperparameters

We set our optimization algorithm by using the following parameter:

```
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

This instructs DL4J that we intend to use the Stochastic Gradient Descent (SGD) optimization algorithm. We set the learning rate like this:

```
.learningRate(learningRate)
```

We're also instructing DL4J that we intend to use Nesterov's parameter-updating strategy with the momentum factor set as 0.9:

```
.updater(Updater.NESTEROVS).momentum(0.9)
```

First hidden layer

The first hidden layer takes on the raw values from the input that we have produced from our vectorization pipeline. These values are typically in the range of [-1.0,1.0] or [0.0,1.0] after various forms of normalization.

```
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
    .weightInit(WeightInit.XAVIER)
    .activation(Activation.RELU)
    .build())
```

Here our layer needs to have the same number of input neurons as the number of independent variable columns in our input vectors:

```
.nIn(numInputs)
```

The number of outputs for the layer is simply the number of neurons in the next layer of the neural network. In this case, this number is represented by the variable numHid

`denNodes`. For this layer, we're also initializing the weights specifically by using the `WeightInit.XAVIER` strategy and our activation function is the rectified linear function:

```
.weightInit(WeightInit.XAVIER)  
.activation(Activation.RELU)
```

Let's now take a look at the next layer for this network, which is the output layer.

Output layer for classification

The output layer we'll use here is a softmax output activation function. Recall from previous chapters that we use the softmax output layer for multiple labels. In this problem, we're building a binary classifier; thus, we also could use a sigmoid output activation function.

```
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)  
.weightInit(WeightInit.XAVIER)  
.activation(Activation.SOFTMAX)  
.nIn(numHiddenNodes).nOut(numOutputs).build())
```

The number of inputs expected is the same number of outputs from the previous (input) layer. The number of output units is two because this is a binary classifier and our softmax will give us scores for each class.



Sigmoid or Softmax Output Layer?

Mathematically using a sigmoid single output is the same as using a softmax output layer with two output units (with MCXENT/NegativeLogLikelihood and one-hot representation, [1,0] or [0,1] instead of [0] or 1). The one-hot vector representation is described in the following sidebar.

We're also specifying our output layer to use the negative log likelihood loss function here because it is commonly paired with the softmax output layer.

One-Hot Vector Representations

In the one-hot vector representation, we have a group of bits for which only a single column's value in the vector can have the value 1.0. All of the other column values will be 0.0. This method of vector representation is often used to represent categorical integer features using a “1 of K” scheme. [Table 5-2](#) presents an example of the regular binary representation of values (0–4) and then the one-hot representation.

Table 5-2. One-hot vectors visually represented

Value	Binary encoding	One-hot representation
0	000	0000001
1	001	0000010
2	010	0000100
3	011	0001000
4	100	0010000

Training the Model

To train the model, we set up a `for` loop that will train the neural network on the input dataset for a given number of epochs (e.g., passes over the entire dataset).

```
for ( int n = 0; n < nEpochs; n++ ) {  
    model.fit( trainIter );  
}
```

To train on the entire dataset, we call the `.fit()` method on the `MultiLayerNetwork` class instance. This class handles the underlying mechanics of applying the proper hyperparameters specified, such as the mini-batch size we configured in the input dataset iterator previously:

```
DataSetIterator testIter = new RecordReaderDataSetIterator(rrTest, batchSize, 0, 2);
```

The `batchSize` variable in this case controls how many examples are gathered from disk and passed on to the model for training, in a batch. As the model is training we should see output on the console similar to this:

```
o.d.o.l.ScoreIterationListener - Score at iteration 0 is 0.6313823699951172  
o.d.o.l.ScoreIterationListener - Score at iteration 10 is 0.4763660430908203  
o.d.o.l.ScoreIterationListener - Score at iteration 20 is 0.42963680267333987  
o.d.o.l.ScoreIterationListener - Score at iteration 30 is 0.39850467681884766  
o.d.o.l.ScoreIterationListener - Score at iteration 40 is 0.3672478103637695
```

This error number should fall over time, and as it approaches 0.0, we know that our model has nearly finished learning the training data.

Evaluating the Model

The code example that follows deals with evaluating our new multilayer perceptron model. Here, we are using the test dataset through the `testIter` object instance to load the actual labels and the predicted labels into the instance of our `Evaluation` class:

```
System.out.println("Evaluate model....");  
Evaluation eval = new Evaluation(numOutputs);
```

```

while(testIter.hasNext()){
    DataSet t = testIter.next();
    INDArray features = t.getFeatureMatrix();
    INDArray lables = t.getLabels();
    INDArray predicted = model.output(features, false);

    eval.eval(lables, predicted);

}

System.out.println(eval.stats());

```

Recall from [Chapter 1](#) the concept of the F1 score and other evaluation-related metrics. The `eval.stats()` method call will result in the following console output:

```

Evaluate model.....

Examples labeled as 0 classified by model as 0: 48 times
Examples labeled as 1 classified by model as 1: 52 times

```

```

=====
Scores=====
Accuracy: 1
Precision: 1
Recall: 1
F1 Score: 1
=====
```

This is a relatively simple dataset, and after 30 epochs, DL4J was able to get a perfect rating (1.0) on all evaluation facets. Let's now move on to more complex examples.

Modeling Handwritten Images Using CNNs

As we discussed at the beginning of this chapter, CNNs are good at image classification. In our next example, we show you how to load and model images of handwritten digits. The resultant CNN model will be able to classify new handwritten digit images it has not previously seen.

In this example, we set up a dataset iterator for both the training dataset and the test dataset. We'll train the model on the training dataset and then evaluate the accuracy of the model with the held-out test dataset. The training dataset that we'll use is the [MNIST handwritten image dataset](#).

The model architecture is different than the previous example in that it has different layers and parameters, as we'll see in a moment. This particular CNN is known as the "LeNet."



LeNet

The LeNet convolutional architecture is focused on a series of convolutional layers followed by max-pooling layers. ([Example 5-2](#) shows this CNN architecture implemented in DL4J.)⁶

Java Code Listing for the LeNet CNN

[Example 5-2](#) provides the [code](#) for the MNIST LeNet CNN example.

Example 5-2. Building LeNet for MNIST in DL4J

```
public class LenetMnistExample {
    private static final Logger log = LoggerFactory
        .getLogger(LenetMnistExample.class);

    public static void main(String[] args) throws Exception {
        int nChannels = 1; // Number of input channels
        int outputNum = 10; // The number of possible outcomes
        int batchSize = 64; // Test batch size
        int nEpochs = 1; // Number of training epochs
        int iterations = 1; // Number of training iterations
        int seed = 123; //

        /*
         * Create an iterator using the batch size for one iteration
         */
        log.info("Load data....");
        DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize, true, 12345);
        DataSetIterator mnistTest = new MnistDataSetIterator(batchSize, false, 12345);

        /*
         * Construct the neural network
         */
        log.info("Build model....");
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(seed)
            .iterations(iterations) // Training iterations as above
            .regularization(true).l2(0.0005)
        /*
         * Uncomment the following for learning decay and bias
         */
            .learningRate(.01)//.biasLearningRate(0.02)
            //.learningRateDecayPolicy(LearningRatePolicy.Inverse)
            //.lrPolicyDecayRate(0.001).lrPolicyPower(0.75)
            .weightInit(WeightInit.XAVIER)
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

⁶ LeCun et al. 1998. “Gradient-based learning applied to document recognition.”

```

.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new ConvolutionLayer.Builder(5, 5)
    //nIn and nOut specify depth. nIn here is the nChannels and
    //nOut is the number of filters to be applied
    .nIn(nChannels)
    .stride(1, 1)
    .nOut(20)
    .activation(Activation.IDENTITY)
    .build())
.layer(1, new SubsamplingLayer
    .Builder(SubsamplingLayer.PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
    //Note that nIn need not be specified in later layers
    .stride(1, 1)
    .nOut(50)
    .activation(Activation.IDENTITY)
    .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer
    .PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
.layer(4, new DenseLayer.Builder().activation(Activation.RELU)
    .nOut(500).build())
.layer(5, new OutputLayer
    .Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
    .nOut(outputNum)
    .activation(Activation.SOFTMAX)
    .build())
.setInputType(InputType.convolutionalFlat(28,28,1)) //See note below
.backprop(true).pretrain(false).build();

/*
Regarding the .setInputType(InputType.convolutionalFlat(28,28,1)) line: This
does a few things.
(a) It adds preprocessors, which handle things like the transition between
the convolutional/subsampling layers and the dense layer
(b) Does some additional configuration validation
(c) Where necessary, sets the nIn (number of input neurons, or input depth in
the case of CNNs) values for each layer based on the size of the previous
layer (but it won't override values manually set by the user)
InputTypes can be used with other layer types too (RNNs, MLPs etc) not
just CNNs.
For normal images (when using ImageRecordReader) use
InputType.convolutional(height,width,depth).
MNIST record reader is a special case, that outputs 28x28 pixel grayscale
(nChannels=1) images, in a "flattened" row vector format (i.e., 1x784 vectors),
hence the "convolutionalFlat" input type used here.

```

```

*/
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();

log.info("Train model....");
model.setListeners(new ScoreIterationListener(1));
for( int i=0; i<nEpochs; i++ ) {
    model.fit(mnistTrain);
    log.info(" *** Completed epoch {} ***", i);

    log.info("Evaluate model....");
    Evaluation eval = new Evaluation(outputNum);
    while(mnistTest.hasNext()){
        DataSet ds = mnistTest.next();
        INDArray output = model.output(ds.getFeatureMatrix(), false);
        eval.eval(ds.getLabels(), output);

    }
    log.info(eval.stats());
    mnistTest.reset();
}
log.info("*****Example finished*****");
}
}

```

In the sections that follow, we look at the specific parts of this program and how they all work together to model the MNIST image dataset.

Loading and Vectorizing the Input Images

In this example we're using a custom dataset iterator called `MnistDataSetIterator`. This is because the MNIST dataset is in a custom binary format—the files are not a set of individual JPG or PNG files in a directory as you might normally expect. To make this example simpler, you can just trust that the necessary functions are being performed behind the scenes and the raw image data is being extracted into an `NDArray` for training with DL4J. The following code snippet shows `MnistDataSetIterator` in action:

```

DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);

```

In this snippet, the code is loading both the training and test datasets in separate iterators. The program will also automatically download the MNIST dataset from the internet and unzip it locally for use.

Network Architecture for LeNet in DL4J

Again, we see the same MultiLayerConfiguration object being used to describe the network architecture we saw in the previous multilayer perceptron example. However, notice that the network shown in the following code example has many more layers and the types of layers are different than in the multilayer perceptron example:

```
/*
    Construct the neural network
*/
log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations) // Training iterations as above
    .regularization(true).l2(0.0005)
    /*
        Uncomment the following for learning decay and bias
    */
    .learningRate(.01)//.biasLearningRate(0.02)
//.learningRateDecayPolicy(LearningRatePolicy.Inverse)
    .lrPolicyDecayRate(0.001).lrPolicyPower(0.75)
    .weightInit(WeightInit.XAVIER)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn and nOut specify depth. nIn here is the nChannels and nOut
        //is the number of filters to be applied
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
    .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
    .layer(2, new ConvolutionLayer.Builder(5, 5)
        //Note that nIn need not be specified in later layers
        .stride(1, 1)
        .nOut(50)
        .activation(Activation.IDENTITY)
        .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
    .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
        .nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
```

```

        .activation(Activation.SOFTMAX)
        .build())
.setInputType(InputType.convolutionalFlat(28,28,1)) //See note below
.backprop(true).pretrain(false).build();

/*
Regarding the .setInputType(InputType.convolutionalFlat(28,28,1)) line: This does
a few things.
(a) It adds preprocessors, which handle things like the transition between the
    convolutional/subsampling layers and the dense layer
(b) Does some additional configuration validation
(c) Where necessary, sets the nIn (number of input neurons, or input depth in the
    case of CNNs) values for each layer based on the size of the previous layer
    (but it won't override values manually set by the user)
InputTypes can be used with other layer types too (RNNs, MLPs etc) not just CNNs.
For normal images (when using ImageRecordReader) use
InputType.convolutional(height,width,depth).
MNIST record reader is a special case, that outputs 28x28 pixel grayscale
(nChannels=1) images, in a "flattened"
row vector format (i.e., 1x784 vectors), hence the "convolutionalFlat" input type
used here.
*/
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();

```

General hyperparameters

We can see some of the main hyperparameters in the following:

```

.seed(seed)
.iterations(iterations) // Training iterations as above
.regularization(true).l2(0.0005)
/*
    Uncomment the following for learning decay and bias
*/
.learningRate(.01)//.biasLearningRate(0.02)
//.learningRateDecayPolicy(LearningRatePolicy.Inverse).lrPolicyDecayRate(0.001)
//.lrPolicyPower(0.75)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)

```

The following list explains how these hyperparameters are set and the values we set for them.

Regularization

Here we see regularization turned on (set to `true`) with an L2 regularizer set with a parameter of 0.0005.

Weight initialization

For this LeNet architecture example, we've found that the Xavier weight initialization strategy works well.

Optimization algorithm

We also see the optimization algorithm set as SGD. You will commonly see SGD in many deep learning examples because it tends to work well in many cases. We discuss variations in optimization algorithms in Chapters 6 and 7.

Updater

We've chosen Nesterov's as the updater for this example. You will see Nesterov's used a lot in our examples because it works well in practice. Nesterov's essentially increases the scale of updates when those updates are consistently in one direction. We can imagine this effect as if we were going down a smooth but very shallow hill. The direction is clear, but we might want to take larger steps.

Convolution layers

In the code example at the beginning of this section, we see a general pattern of convolutional layers and then max-pooling layers as described in [Chapter 4](#). In the example that follows, we see the definition for the first convolution layer in the LeNet network:

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
    //nIn and nOut specify depth. nIn here is the nChannels and nOut is the
    //number of filters to be applied
    .nIn(nChannels)
    .stride(1, 1)
    .nOut(20)
    .activation(Activation.IDENTITY)
    .build())
```

In this example, we can see the Java `Builder` pattern setting up the properties for a convolution layer. Let's take a look at each of those properties.

Filter size

Here, we are creating a filter size for this layer with a 5×5 dimension.

Input data channels

For this example, the number of channels is set to 1 because the custom dataset iterator converts the raw image automatically to a black-and-white image. Other examples might often define three channels of input representing the RGB channels in the image data.

Stride

The stride for this layer is set to $(1, 1)$, meaning that it takes only a single step over and then down as it slides the filter across the input volume.

Activation function

We use the identity function as the output of this convolution layer.



Notice the Identity Activation Function?

The LeNet network architecture (1998)⁷ predates ReLUs (2012)⁸ by several years. Changing convolutional layers over to use the ReLUs accelerated the convergence of SGD. For historical purposes, we are keeping the original LeNet architecture intact in this example.



Convolution Layers and Activation Functions

In more recent CNNs we often see a convolution layer set up with a ReLU activation function.

Max-pooling layers

In the following example, you can see the part of the code representing the pooling layer directly after the first convolution layer:

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
```

Here are the layer properties:

Max pooling

This layer is set up as a max-pooling layer. This pooling layer has a filter size of $(2,2)$, meaning that we're downsampling from a 5×5 filter in the previous layer to a 2×2 grid in the current layer.

Setting the pooling layer stride

The filter stride is set as $(2,2)$, which means that it will step two values as it slides horizontally and two values when it steps down to the next row.

Output layer

Because we're building a classification model with more than two labels (e.g., the digits 0–9), we want an output layer that is using a softmax activation function. The following example shows how to do this:

⁷ LeCun et al. 1998. “Gradient-based learning applied to document recognition.”

⁸ Krizhevsky, Sutskever, and Hinton. 2012. “ImageNet Classification with Deep Convolutional Neural Networks.”

```

.layer(5, new OutputLayer
    .Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
    .nOut(outputNum)
    .activation(Activation.SOFTMAX)
    .build())

```

This output layer has a negative log likelihood loss function (as is customarily seen with a softmax output layer) and the number of output units is equal to the number of classes or labels we have for this dataset.

Training the CNN

Now that we have a LeNet convolutional model architecture set up, we can initialize the `MultiLayerNetwork` object to train on our input datasets. The code snippet that follows shows how to take the configuration from the previous section and set up the network:

```

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();

```

With the model configured, we now can train on our input MNIST dataset for the desired number of epochs, as shown in the following:

```

log.info("Train model....");
model.setListeners(new ScoreIterationListener(1));
for( int i=0; i<nEpochs; i++ ) {
    model.fit(mnistTrain);
    log.info("*** Completed epoch {} ***", i);

    log.info("Evaluate model....");
    Evaluation eval = new Evaluation(outputNum);
    while(mnistTest.hasNext()){
        DataSet ds = mnistTest.next();
        INDArray output = model.output(ds.getFeatureMatrix(), false);
        eval.eval(ds.getLabels(), output);

    }
    log.info(eval.stats());
    mnistTest.reset();
}
log.info("*****Example finished*****");

```

In the training loop, we see the same pattern with the MNIST dataset iterator handling the mini-batching of the images to the model `.fit()` method. After the `.fit()` method completes the current epoch, we then check how well the model is trained by testing the model against the test MNIST dataset. This allows us to see F1 scores progressively over the epochs of training. Over time, we should see the evaluation scores rise and the loss function error fall.

Modeling Sequence Data by Using Recurrent Neural Networks

In this section, we show off the generative and discriminative capabilities of Recurrent Neural Networks via two examples: generating Shakespeare and classifying time-series.

Generating Shakespeare via LSTMs

We begin with a fun example of modeling the works of Shakespeare. In this example, we train an LSTM Recurrent Neural Network to generate new lines of Shakespeare (well, “Shakespeare sounding”). Note that we’re treating text as a sequence of characters, and our model is predicting the next most likely character based off the past characters it encountered. You also can adapt this model to work on other common sequence data, such as log data or sensor data as we’ll see in the next example.



The Unreasonable Effectiveness of Recurrent Neural Networks

This example is somewhat inspired by Andrej Karpathy’s blog post, “[The Unreasonable Effectiveness of Recurrent Neural Networks](#)”.



Source Text for Training LSTMs

This example is set up to train on the *Complete Works of William Shakespeare*, downloaded from Project Gutenberg.⁹ Training on other text sources should be relatively easy to implement.

High-level modeling workflow

In this example, we introduce the concept of setting up a Recurrent Neural Network architecture in DL4J. This example builds on some of the concepts we’ve learned in previous examples, such as the following:

- Loading input datasets for training
- Setting up the network architecture configuration

We’ll also introduce some new parts of the DL4J API that supports Recurrent Neural Networks specifically. This example learns lines of Shakespeare character by character. We then ask the model to generate new original lines based on the model it has learned.

⁹ [Complete Works of William Shakespeare](#) 5.3 MB file in UTF-8 Encoding, ~5.4 million characters.

As the example runs, we can see the output change from something like:

```
----- Sample 0 -----
lnee!
Lhir tape shepyang? Nocw; mame. Budt hlant'nthely ler ild
Py theu sfochill'ad my and ocs im nereepapd werer;
Motadid. Mert hatterhirl. Iit nesdoesd'nlowhednanieivetranns deugheuind
Bred yetide rathane fojlonnd thivh uweet.
Thy lametom theuegfast lart souclalitoloe ilntangylrt or

----- Sample 1 -----
l,, ne agly
Lot Bolncanbom bavantenfircasle womlidibl.
NTERIOO. IrdmisfuIoItolleeeddortiss hot buye.
The hetenle of ile,
'merilliyydingiponi, bomgule? Shurtstarer of ate,
Onbibly ot ire pomxatgillant, dakl.
Oxt Mtanlonfye wiudsimotime raugadent deu'y ondtstes.
If vonee.
Whol touEde
```

to output that looks more like:

```
----- Sample 0 -----
ous reward me, Master Warce! I-will stay
shall; for I one as mine lord.
CLOTEN. Come, I will thigh, i'; and what wam! Hath dravelly
The albowed out, Aside dismernicges could be a
druck than there's thoughts, here is we with me and rag.
Thou shalt love it doth my child.
PERDITA. Ti

----- Sample 1 -----
on,
Incie Paties, go, thurst with thy flounds by the bands. Exit COURSTIO
FLORIZEL. Uncle, an if you,
Abassom the man,
Stars, you spite-hath loved.
QUEEN MANGER On stay is! Who is mer?
CLOTEN. Hang't, what I'll remain,
Cap nothes same so here;
My tens
```

Let's take a look at the Java code for this example.

Java code for modeling Shakespeare

Example 5-3 lists the full code for the [example](#).

Example 5-3. Modeling and generating Shakespeare by using DL4J's LSTM

```
public class GravesLSTMCharModellingExample {
    public static void main( String[] args ) throws Exception {
        int lstmLayerSize = 200;                                //Number of units in each
                                                               //GravesLSTM layer
        int miniBatchSize = 32;                                 //Size of mini batch to use when
                                                               //training
        int exampleLength = 1000;                               //Length of each training example
                                                               //sequence to use. This could
                                                               //certainly be increased
        int tbpttLength = 50;                                //Length for truncated
                                                               //backpropagation through time.
                                                               //i.e., do parameter updates ever
                                                               //50 characters
        int numEpochs = 1;                                    //Total number of training epochs
        int generateSamplesEveryNMinibatches = 10;           //How frequently to generate
                                                               //samples from the network?
                                                               //1000 characters / 50 tbptt
                                                               //length: 20 parameter updates
                                                               //per minibatch
        int nSamplesToGenerate = 4;                           //Number of samples to generate
                                                               //after each training epoch
        int nCharactersToSample = 300;                        //Length of each sample to
                                                               //generate
        String generationInitialization = null;             //Optional character
                                                               //initialization; a random
                                                               //character is used if null
        // Above is Used to 'prime' the LSTM with a character sequence to
        // continue/complete.
        // Initialization characters must all be in CharacterIterator
        // .getMinimalCharacterSet() by default
        Random rng = new Random(12345);

        //Get a DataSetIterator that handles vectorization of text into something we
        // can use to train our GravesLSTM network.
        CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
        int nOut = iter.totalOutcomes();

        //Set up network configuration:
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
            .learningRate(0.1)
            .rmsDecay(0.95)
            .seed(12345)
            .regularization(true)
            .l2(0.001)
            .weightInit(WeightInit.XAVIER)
            .updater(Updater.RMSPROP)
            .list()
            .layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns()))
```

```

    .nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX) //MCXENT + softmax for classification
        .nIn(lstmLayerSize).nOut(nOut).build()))
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();
net.setListeners(new ScoreIterationListener(1));

//Print the number of parameters in the network (and for each layer)
Layer[] layers = net.getLayers();
int totalNumParams = 0;
for( int i=0; i<layers.length; i++ ){
    int nParams = layers[i].numParams();
    System.out
        .println("Number of parameters in layer " + i + ":" + nParams);
    totalNumParams += nParams;
}
System.out.println("Total number of network parameters: " + totalNumParams);

//Do training, and then generate and print samples from network
int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
        net.fit(ds);
        if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
            System.out.println("-----");
            System.out.println("Completed " + miniBatchNumber +
                " minibatches of size " + miniBatchSize + "x" + exampleLength
                + " characters" );
            System.out.println("Sampling characters from network
                given initialization \""
                + (generationInitialization == null ? "" :
                generationInitialization) + "\"");
            String[] samples = sampleCharactersFromNetwork(
                generationInitialization,net,iter,rng,nCharactersToSample,
                nSamplesToGenerate);
            for( int j=0; j<samples.length; j++ ){
                System.out.println("---- Sample " + j + " ----");
                System.out.println(samples[j]);
                System.out.println();
            }
        }
    }
}

```

```

        iter.reset(); //Reset iterator for another epoch
    }

    System.out.println("\n\nExample complete");
}

/** Downloads Shakespeare training data and stores it locally (temp directory).
 * Then set up and return a simple DataSetIterator that does
 * vectorization based on the text.
 * @param miniBatchSize Number of text segments in each training mini-batch
 * @param sequenceLength Number of characters in each text segment.
 */
public static CharacterIterator getShakespeareIterator(int miniBatchSize,
    int sequenceLength) throws Exception{
    //The Complete Works of William Shakespeare
    //5.3MB file in UTF-8 Encoding, ~5.4 million characters
    //https://www.gutenberg.org/ebooks/100
    String url = "https://s3.amazonaws.com/dl4j-distribution/pg100.txt";
    String tempDir = System.getProperty("java.io.tmpdir");
    String fileLocation = tempDir + "/Shakespeare.txt"; //Storage location from
                                                       //downloaded file

    File f = new File(fileLocation);
    if( !f.exists() ){
        FileUtils.copyURLToFile(new URL(url), f);
        System.out.println("File downloaded to " + f.getAbsolutePath());
    } else {
        System.out.println("Using existing text file at " + f.getAbsolutePath());
    }

    if(!f.exists()) throw new IOException("File does not exist: " +
        fileLocation); //Download problem?

    //Which characters are allowed? Others will be removed
    char[] validCharacters = CharacterIterator.getMinimalCharacterSet();
    return new CharacterIterator(fileLocation, Charset.forName("UTF-8"),
        miniBatchSize, sequenceLength, validCharacters, new Random(12345));
}

/** Generate a sample from the network, given an (optional, possibly null)
 * initialization. Initialization
 * can be used to 'prime' the RNN with a sequence you want to extend/continue.
 * Note that the initialization is used for all samples
 * @param initialization String, may be null. If null, select a random
 * character as initialization for all samples
 * @param charactersToSample Number of characters to sample from network
 * (excluding initialization)
 * @param net MultilayerNetwork with one or more GravesLSTM/RNN layers and a
 * softmax output layer
 * @param iter CharacterIterator. Used for going from indexes back to characters
 */
private static String[] sampleCharactersFromNetwork(String initialization,

```

```

        MultiLayerNetwork net,
        CharacterIterator iter,
        Random rng,
        int charactersToSample,
        int numSamples ){
    //Set up initialization. If no initialization: use a random character
    if( initialization == null ){
        initialization = String.valueOf(iter.getRandomCharacter());
    }

    //Create input for initialization
    INDArray initializationInput = Nd4j.zeros(numSamples, iter.inputColumns(),
        initialization.length());
    char[] init = initialization.toCharArray();
    for( int i=0; i<init.length; i++ ){
        int idx = iter.convertCharacterToIndex(init[i]);
        for( int j=0; j<numSamples; j++ ){
            initializationInput.putScalar(new int[]{j,idx,i}, 1.0f);
        }
    }
}

StringBuilder[] sb = new StringBuilder[numSamples];
for( int i=0; i<numSamples; i++ ) sb[i] = new StringBuilder(initialization);

//Sample from network (and feed samples back into input) one character at a
//time (for all samples)
//Sampling is done in parallel here
net.rnnClearPreviousState();
INDArray output = net.rnnTimeStep(initializationInput);
output = output.tensorAlongDimension(output.size(2)-1,0); //Gets the last
//time step output

for( int i=0; i<charactersToSample; i++ ){
    //Set up next input (single time step) by sampling from previous output
    INDArray nextInput = Nd4j.zeros(numSamples,iter.inputColumns());
    //Output is a probability distribution. Sample from this for each example
    //we want to generate, and add it to the new input
    for( int s=0; s<numSamples; s++ ){
        double[] outputProbDistribution = new double[iter.totalOutcomes()];
        for( int j=0; j<outputProbDistribution.length; j++ )
            outputProbDistribution[j] = output.getDouble(s,j);
        int sampledCharacterIdx =
            sampleFromDistribution(outputProbDistribution,rng);

        //Prepare next time step input
        nextInput.putScalar(new int[]{s,sampledCharacterIdx}, 1.0f);
        //Add sampled character to StringBuilder (human readable output)
        sb[s].append(iter.convertIndexToCharacter(sampledCharacterIdx));
    }
    output = net.rnnTimeStep(nextInput); //Do one time step of forward pass
}

```

```

String[] out = new String[numSamples];
for( int i=0; i<numSamples; i++ ) out[i] = sb[i].toString();
return out;
}

/** Given a probability distribution over discrete classes, sample from the
 * distribution and return the generated class index.
 * @param distribution Probability distribution over classes. Must sum to 1.0
 */
public static int sampleFromDistribution( double[] distribution, Random rng ){
    double d = rng.nextDouble();
    double sum = 0.0;
    for( int i=0; i<distribution.length; i++ ){
        sum += distribution[i];
        if( d <= sum ) return i;
    }
    //Should never happen if distribution is a valid probability distribution
    throw new IllegalArgumentException("Distribution is invalid? d="+d+", "
        +sum);
}
}
}

```

In the following sections we discuss this code listing and explain how the major sections work together to model and generate Shakespeare.

Setting up input data and vectorization

The input data is automatically downloaded and converted into NDArrays by the support classes included in the example, as demonstrated in the following snippet:

```
CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
```

We'll leave it to you to explore how this works under the hood as interest dictates.

LSTM network architecture

Just as in the previous two examples, here we use the same builder pattern for setting up the layers of the LSTM network.

```

//Set up network configuration:
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns()))

```

```

    .nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
.layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)           //MCXENT + softmax for
                                                //classification
        .nIn(lstmLayerSize).nOut(nOut).build())
.backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
    .tBPTTBackwardLength(tbpttLength)
.pretrain(false).backprop(true)
.build();

```

General comments on hyperparameters. In this example, we again see SGD as the optimization algorithm and the learning rate set to `0.1`. Regularization is turned on, with L2 set to `0.001`. The updater is RMSPROP, which is different than the other examples we've seen thus far.

Hidden layers

We use the special GravesLSTM layers with tanh activation functions for the layers other than the output layer.

Output layer

The output layer is different than we've seen so far because we are using a special `RnnOutputLayer` to handle the different types of output that is possible from a Recurrent Neural Network, as illustrated here:

```

.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)           //MCXENT + softmax for
                                                //classification
        .nIn(lstmLayerSize).nOut(nOut).build())

```

You can also see that even though again we're using a softmax activation function for the output layer, the loss function used here is `LossFunction.MCXENT`.



RMSProp in Practice

RMSProp by design has a normalizing effect on the scale of the updates. This means that parameter updates with different scales (different layers, or different parameters within the same layer) end up having about the same size after using this updater.

Training the LSTM network

The following code example shows how to train the network:

```
//Do training, and then generate and print samples from network
int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
        net.fit(ds);
        if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
            System.out.println("-----");
            System.out.println("Completed " + miniBatchNumber +
                " minibatches of size " + miniBatchSize +
                "x" + exampleLength + " characters" );
            System.out.println("Sampling characters from network given
                initialization \"\" + (generationInitialization == null ? "" :
                generationInitialization) + "\"");
            String[] samples =
                sampleCharactersFromNetwork(
                    generationInitialization,net,iter,rng,nCharactersToSample,
                    nSamplesToGenerate);
            for( int j=0; j<samples.length; j++ ){
                System.out.println("---- Sample " + j + " ----");
                System.out.println(samples[j]);
                System.out.println();
            }
        }
    }
}

iter.reset(); //Reset iterator for another epoch
}

System.out.println("\n\nExample complete");
```

We're using the DL4J API a bit differently in this example. As opposed to calling the `.fit()` method on the iterator itself, here we see it being called on mini-batches explicitly. This allows us a finer level of control about what we can do between calls to `.fit()` on a mini-batch. In this case, it allows us to generate samples from the network during a pass over the input Shakespeare dataset.

As the training progresses, we should see output on the console in which the loss function is seeing progressively less error during training:

```
o.d.o.l.ScoreIterationListener - Score at iteration 0 is 217.28348109866505
o.d.o.l.ScoreIterationListener - Score at iteration 1 is 213.24020789706773
o.d.o.l.ScoreIterationListener - Score at iteration 2 is 212.96001041971766
o.d.o.l.ScoreIterationListener - Score at iteration 3 is 175.06079409241767
o.d.o.l.ScoreIterationListener - Score at iteration 4 is 165.25272077487378
```

In this example, we see output similar to the preceding snippet periodically interrupted with the output of generated sentence samples from the in-progress network.

Generating Shakespeare samples

To generate samples from the network, we call the support method listed in the example that generates a synthetic passage from our model, as shown here:

```
String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,
    rng,nCharactersToSample,nSamplesToGenerate);
for( int j=0; j<samples.length; j++ ){
    System.out.println("----- Sample " + j + " -----");
    System.out.println(samples[j]);
    System.out.println();
}
```

This is the part of the code that prints the output to the console periodically during training.

Classifying Sensor Time-series Sequences Using LSTMs

In this example, we take the reader through a sequence classification example using an LSTM Recurrent Neural Network. We use the [Synthetic Control Chart Time Series Data Set](#) from the University of California, Irvine (UCI) machine learning repository. This code example will build a model to classify univariate time-series as belonging to one of six categories:

- Normal (C)
- Cyclic (B)
- Increasing trend (E)
- Decreasing trend (A)
- Upward shift (D)
- Downward shift (F)

Figure 5-1 presents renders¹⁰ of examples from each of these categories.

¹⁰ Image from <https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series>.

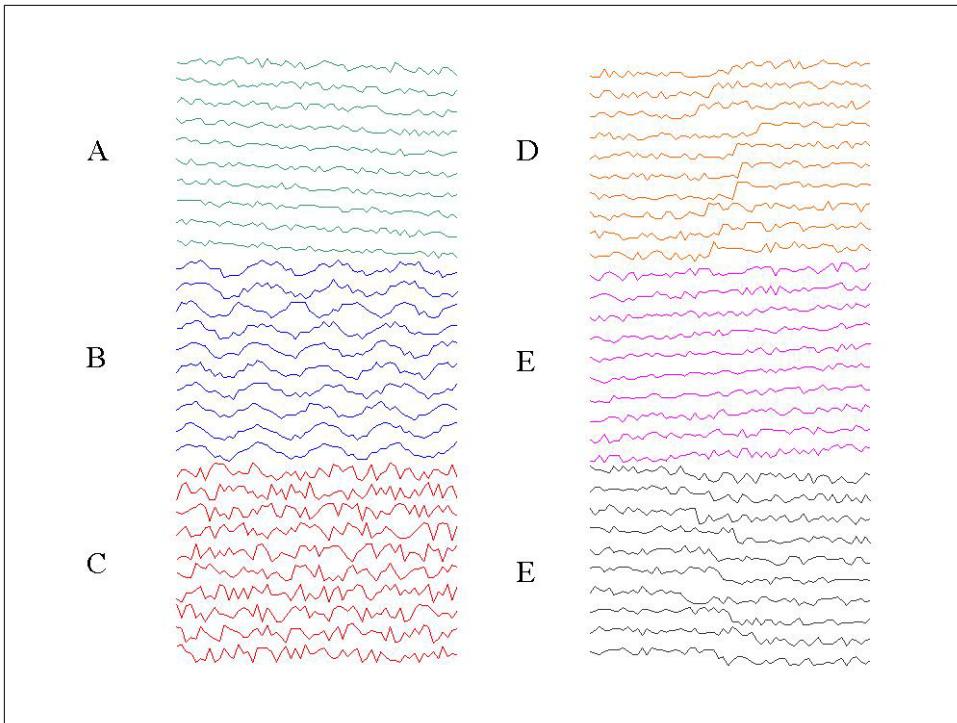


Figure 5-1. Renders of synthetic time-series sequences from the UCI repository

These six classes of sequences are a great example of real data that would be generated in the embedded sensor realm and give us a practical dataset with which to work.

Java code listing for recurrent classification example

Example 5-4 presents the [Java code for our next example](#), the recurrent classification example, in which we build a model to classify the sequences in “UCI’s Synthetic Control Chart Time Series Data Set.”

Example 5-4. Java code for UCI sequence classification example

```
public class UCISequenceClassificationExample {
    private static final Logger log = LoggerFactory
        .getLogger(UCISequenceClassificationExample.class);

    // 'baseDir': Base directory for the data. Change this if you want to save the
    // data somewhere else
    private static File baseDir = new File("src/main/resources/uci/");
    private static File baseTrainDir = new File(baseDir, "train");
    private static File featuresDirTrain = new File(baseTrainDir, "features");
    private static File labelsDirTrain = new File(baseTrainDir, "labels");
```

```

private static File baseTestDir = new File(baseDir, "test");
private static File featuresDirTest = new File(baseTestDir, "features");
private static File labelsDirTest = new File(baseTestDir, "labels");

public static void main(String[] args) throws Exception {
    downloadUCIData();

    // ----- Load the training data -----
    //Note that we have 450 training files for features: train/features/0.csv
    //through train/features/449.csv
    SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
    trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
        .getAbsolutePath() + "%d.csv", 0, 449));
    SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
    trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
        .getAbsolutePath() + "%d.csv", 0, 449));

    int miniBatchSize = 10;
    int numLabelClasses = 6;
    DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(
        trainFeatures, trainLabels, miniBatchSize, numLabelClasses,
        false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

    //Normalize the training data
    DataNormalization normalizer = new NormalizerStandardize();
    normalizer.fit(trainData);           //Collect training data statistics
    trainData.reset();

    //Use previously collected statistics to normalize on-the-fly. Each DataSet
    //returned by 'trainData' iterator will be normalized
    trainData.setPreProcessor(normalizer);

    // ----- Load the test data -----
    //Same process as for the training data.
    SequenceRecordReader testFeatures = new CSVSequenceRecordReader();
    testFeatures.initialize(new NumberedFileInputSplit(featuresDirTest
        .getAbsolutePath() + "%d.csv", 0, 149));
    SequenceRecordReader testLabels = new CSVSequenceRecordReader();
    testLabels.initialize(new NumberedFileInputSplit(
        labelsDirTest.getAbsolutePath() + "%d.csv", 0, 149));

    DataSetIterator testData = new SequenceRecordReaderDataSetIterator(
        testFeatures, testLabels, miniBatchSize, numLabelClasses,
        false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

    testData.setPreProcessor(normalizer); //Note that we are using the exact
                                         //same normalization process as the
                                         //training data

    // ----- Configure the network -----

```

```

MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(123)      //Random number generator seed for
                    //improved repeatability. Optional.
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .learningRate(0.005)
    .gradientNormalization(GradientNormalization
        .ClipElementWiseAbsoluteValue) //Not always required, but helps
                                    //with this data set
    .gradientNormalizationThreshold(0.5)
    .list()
    .layer(0, new GravesLSTM.Builder().activation(Activation.TANH).nIn(1)
        .nOut(10).build())
    .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction
        .MCXENT)
        .activation(Activation.SOFTMAX).nIn(10).nOut(numLabelClasses)
        .build())
    .pretrain(false).backprop(true).build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();

net.setListeners(new ScoreIterationListener(20)); //Print the score (loss
                                                //function value) every 20 iterations

// -- Train the network, evaluating the test set performance at each epoch --
int nEpochs = 40;
String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);

    //Evaluate on the test set:
    Evaluation evaluation = net.evaluate(testData);
    log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

    testData.reset();
    trainData.reset();
}

log.info("----- Example Complete -----");
}

//This method downloads the data, and converts the "one time series per line"
//format into a suitable
//CSV sequence format that DataVec (CsvSequenceRecordReader) and DL4J can read.
private static void downloadUCIData() throws Exception {
    if (baseDir.exists()) return; //Data already exists, don't download it again
}

```

```

String url =
    "https://archive.ics.uci.edu/ml/machine-learning-databases/
     synthetic_control-mld/synthetic_control.data";
String data = IOUtils.toString(new URL(url));

String[] lines = data.split("\n");

//Create directories
baseDir.mkdir();
baseTrainDir.mkdir();
featuresDirTrain.mkdir();
labelsDirTrain.mkdir();
baseTestDir.mkdir();
featuresDirTest.mkdir();
labelsDirTest.mkdir();

int lineCount = 0;
List<Pair<String, Integer>> contentAndLabels = new ArrayList<>();
for (String line : lines) {
    String transposed = line.replaceAll(" +", "\n");

    //Labels: first 100 examples (lines) are label 0, second 100 examples are
    //label 1, and so on
    contentAndLabels.add(new Pair<>(transposed, lineCount++ / 100));
}

//Randomize and do a train/test split:
Collections.shuffle(contentAndLabels, new Random(12345));

int nTrain = 450;    //75% train, 25% test
int trainCount = 0;
int testCount = 0;
for (Pair<String, Integer> p : contentAndLabels) {
    //Write output in a format we can read, in the appropriate locations
    File outPathFeatures;
    File outPathLabels;
    if (trainCount < nTrain) {
        outPathFeatures = new File(featuresDirTrain, trainCount + ".csv");
        outPathLabels = new File(labelsDirTrain, trainCount + ".csv");
        trainCount++;
    } else {
        outPathFeatures = new File(featuresDirTest, testCount + ".csv");
        outPathLabels = new File(labelsDirTest, testCount + ".csv");
        testCount++;
    }

    FileUtils.writeStringToFile(outPathFeatures, p.getFirst());
    FileUtils.writeStringToFile(outPathLabels, p.getSecond().toString());
}
}
}

```

In the sections that follow, we look at the various parts of the code.

Setting up input data and vectorization

In [Example 5-4](#) we don't need to manually retrieve the data because the `downloadUCI` Data() method does that for us. It also splits the 600 time-series examples into a training set of 450 and a test set of 150 examples. It then writes the data in a format suitable for loading by using the `CSVSequenceRecordReader` for sequence classification. With this format, we have one time-series per file, and a separate file for the labels.

For example, `train/features/0.csv` represents the features for record 0 while the file `train/labels/0.csv` represents the label for record 0. Because the data is a univariate time-series, we have only one column in the CSV file. Normally, each column would contain multiple values (e.g., one time-step per row). Furthermore, because we have only one label for each time-series, the labels CSV files contain only a single value.

If we isolate part of the code to load the training data, as shown in the code that follows, we can see how to use the `SequenceRecordReader` in DL4J to read the CSV file,¹¹ create the labels, and then create a `DataSetIterator` to work with this specific type of sequence data:

```
// ----- Load the training data -----
//Note that we have 450 training files for features: train/features/0.csv through
//train/features/449.csv
SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
    .getAbsolutePath() + "%d.csv", 0, 449));
SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
    .getAbsolutePath() + "%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;
DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(
    trainFeatures, trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

//Normalize the training data
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData); //Collect training data statistics
trainData.reset();

//Use previously collected statistics to normalize on-the-fly. Each DataSet
//returned by 'trainData' iterator will be normalized
trainData.setPreProcessor(normalizer);
```

¹¹ For more details on this step, go to <http://deeplearning4j.org/usingrnn#data>.

Toward the end of this code snippet, we can see the DataNormalizer object being created to collect dataset statistics. This allows us to collect global dataset statistics and normalize the training data for better training.

Network architecture and training

Our neural network base architecture is an LSTM Recurrent Neural Network, which you can see being set up with the MultiLayerConfiguration object in the following code snippet:

```
// ----- Configure the network -----
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(123)      //Random number generator seed for improved repeatability.
                    //Optional.
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .learningRate(0.005)
    .gradientNormalization(GradientNormalization
        .ClipElementWiseAbsoluteValue) //Not always required, but helps with
                                    //this data set
    .gradientNormalizationThreshold(0.5)
    .list()
    .layer(0, new GravesLSTM.Builder().activation(Activation.TANH).nIn(1)
        .nOut(10).build())
    .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
        .activation(Activation.SOFTMAX).nIn(10).nOut(numLabelClasses)
        .build())
    .pretrain(false).backprop(true).build();
```

Here, we need only a single LSTM layer connected to a softmax output layer. We're using the XAVIER weight initialization method with NESTEROVS update strategy and a learning rate of 0.005.

As with the other code examples, we train for a number of epochs until our error rate has fallen sufficiently, as demonstrated here:

```
// ----- Train the network, evaluating the test set performance at each epoch ---
int nEpochs = 40;
String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);

    //Evaluate on the test set:
    Evaluation evaluation = net.evaluate(testData);
    log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

    testData.reset();
    trainData.reset();
}
```

```
log.info("----- Example Complete -----");
```

Here, again, we're calling the `.fit()` method to train on our vectorized and normalized input training set. We then use the `Evaluation` object to measure how well our Recurrent Neural Networks model generalizes to our held-out test data.

Using Autoencoders for Anomaly Detection

To demonstrate practical use of autoencoders, we'll show anomaly detection performs on the MNIST dataset using a simple autoencoder without pretraining.

The goal is to identify outlier digits; that is, those digits that are unusual or not like the typical digits. In this example, we do this by using reconstruction error: stereotypical examples should have low reconstruction error, whereas outliers should have high reconstruction error.

Also in this example, we'll set up the model configuration as in the other examples, but our model architecture will be different. For autoencoders, we're using `Dense Layers` (fully connected) for most layers, but we're focused on a series of layers that "funnels" down and then "expands" back out to the output layer.

Java Code Listing for Autoencoder Example

Example 5-5 shows the Java code for the anomaly autoencoder example.

Example 5-5. Java code for autoencoder to find anomalies

```
public class MNISTAnomalyExample {  
  
    public static void main(String[] args) throws Exception {  
  
        //Set up network. 784 in/out (as MNIST images are 28x28).  
        //784 -> 250 -> 10 -> 250 -> 784  
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()  
            .seed(12345)  
            .iterations(1)  
            .weightInit(WeightInit.XAVIER)  
            .updater(Updater.ADAGRAD)  
            .activation(Activation.RELU)  
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)  
            .learningRate(0.05)  
            .regularization(true).l2(0.0001)  
            .list()  
            .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)  
                .build())  
            .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)  
                .build())  
    }  
}
```

```

    .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
          .build())
    .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
          .lossFunction(LossFunctions.LossFunction.MSE)
          .build())
    .pretrain(false).backprop(true)
    .build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.setListeners(Collections.singletonList((IterationListener) new
    ScoreIterationListener(1)));

//Load data and split into training and testing sets. 40000 train, 10000 test
DataSetIterator iter = new MnistDataSetIterator(100,50000, false);

List<INDArray> featuresTrain = new ArrayList<>();
List<INDArray> featuresTest = new ArrayList<>();
List<INDArray> labelsTest = new ArrayList<>();

Random r = new Random(12345);
while(iter.hasNext()){
    DataSet ds = iter.next();
    SplitTestAndTrain split = ds.splitTestAndTrain(80, r); //80/20 split
    //from miniBatch
    //= 100
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argmax(dsTest.getLabels(),1); //Convert from
    //one-hot
    //representation
    //-> index
    labelsTest.add(indexes);
}

//Train model:
int nEpochs = 30;
for( int epoch=0; epoch<nEpochs; epoch++ ){
    for(INDArray data : featuresTrain){
        net.fit(data,data);
    }
    System.out.println("Epoch " + epoch + " complete");
}

//Evaluate the model on test data
//Score each digit/example in test set separately
//Then add triple (score, digit, and INDArray data) to lists and sort by
//score
//This allows us to get best N and worst N digits for each type
Map<Integer, List<Triple<Double, Integer, INDArray>>> listsByDigit =
    new HashMap<>();
for( int i=0; i<10; i++ ) listsByDigit.put(i,new ArrayList<Triple<Double,

```

```

        Integer,INDArray>>()));

int count = 0;
for( int i=0; i<featuresTest.size(); i++ ){
    INDArray testData = featuresTest.get(i);
    INDArray labels = labelsTest.get(i);
    int nRows = testData.rows();
    for( int j=0; j<nRows; j++){
        INDArray example = testData.getRow(j);
        int label = (int)labels.getDouble(j);
        double score = net.score(new DataSet(example,example));
        listsByDigit.get(label).add(new ImmutableTriple<>(score, count++,
            example));
    }
}

//Sort data by score, separately for each digit
Comparator<Triple<Double, Integer, INDArray>> c
= new Comparator<Triple<Double, Integer, INDArray>>() {
@Override
public int compare(Triple<Double, Integer, INDArray> o1, Triple<Double,
    Integer, INDArray> o2) {
    return Double.compare(o1.getLeft(),o2.getLeft());
}
};

for(List<Triple<Double, Integer, INDArray>> list : listsByDigit.values()){
    Collections.sort(list, c);
}

//Select the 5 best and 5 worst numbers (by reconstruction error) for each
//digit
List<INDArray> best = new ArrayList<>(50);
List<INDArray> worst = new ArrayList<>(50);
for( int i=0; i<10; i++ ){
    List<Triple<Double, Integer, INDArray>> list = listsByDigit.get(i);
    for( int j=0; j<5; j++ ){
        best.add(list.get(j).getRight());
        worst.add(list.get(list.size()-j-1).getRight());
    }
}

//Visualize the best and worst digits
MNISTVisualizer bestVisualizer = new MNISTVisualizer(2.0,best,"Best
    (Low Rec. Error)");
bestVisualizer.visualize();

MNISTVisualizer worstVisualizer = new MNISTVisualizer(2.0,worst,"Worst
    (High Rec. Error)");
worstVisualizer.visualize();
}

```

```

public static class MNISTVisualizer {
    private double imageScale;
    private List<INDArray> digits; //Digits (as row vectors), one per INDArray
    private String title;
    private int gridWidth;

    public MNISTVisualizer(double imageScale, List<INDArray> digits,
                           String title ) {
        this(imageScale, digits, title, 5);
    }

    public MNISTVisualizer(double imageScale, List<INDArray> digits,
                           String title, int gridWidth ) {
        this.imageScale = imageScale;
        this.digits = digits;
        this.title = title;
        this.gridWidth = gridWidth;
    }

    public void visualize(){
        JFrame frame = new JFrame();
        frame.setTitle(title);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(0,gridWidth));

        List<JLabel> list = getComponents();
        for(JLabel image : list){
            panel.add(image);
        }

        frame.add(panel);
        frame.setVisible(true);
        frame.pack();
    }

    private List<JLabel> getComponents(){
        List<JLabel> images = new ArrayList<>();
        for( INDArray arr : digits ){
            BufferedImage bi = new BufferedImage(28,28,BufferedImage
                .TYPE_BYTE_GRAY);
            for( int i=0; i<784; i++ ){
                bi.getRaster().setSample(i % 28, i / 28, 0, (int)(255*arr
                    .getDouble(i)));
            }
            ImageIcon orig = new ImageIcon(bi);
            Image imageScaled = orig.getImage().getScaledInstance((int)
                (imageScale*28),(int)(imageScale*28),Image.SCALE_REPLICATE);
            ImageIcon scaled = new ImageIcon(imageScaled);
            images.add(new JLabel(scaled));
        }
    }
}

```

```

        return images;
    }
}
}

```

In the sections that follow, we'll comment on specific parts of the example's code.

Setting Up Input Data

Following is the section of the example code that loads the MNIST dataset with a custom iterator.

```

//Load data and split into training and testing sets. 40000 train, 10000 test
DataSetIterator iter = new MnistDataSetIterator(100,50000, false);

List<INDArray> featuresTrain = new ArrayList<>();
List<INDArray> featuresTest = new ArrayList<>();
List<INDArray> labelsTest = new ArrayList<>();

Random r = new Random(12345);
while(iter.hasNext()){
    DataSet ds = iter.next();
    SplitTestAndTrain split = ds.splitTestAndTrain(80, r); //80/20 split
                                                        // (from miniBatch
                                                        // = 100)
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argmax(dsTest.getLabels(),1); //Convert from one-hot
                                                        // representation ->
                                                        // index
    labelsTest.add(indexes);
}

```

We're handling the training and test data differently in this example compared to previous examples so that we can demonstrate variations on how to use the DL4J and ND4J APIs. In the `while` loop, we manually separate the dataset into test and train datasets.

Autoencoder Network Architecture and Training

As discussed previously, the network architecture for the autoencoder typically is shaped like a funnel and then expands back out to the full input dataset size at the output layer. We seek to learn the most efficient form of the input data by using autoencoders. Autoencoders learn to best represent the data, hence their name. Following is the code to set up our autoencoder:

```

//Set up network. 784 in/out (as MNIST images are 28x28).
//784 -> 250 -> 10 -> 250 -> 784
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(12345)
    .iterations(1)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.ADAGRAD)
    .activation(Activation.RELU)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(0.05)
    .regularization(true).l2(0.0001)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
        .build())
    .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)
        .build())
    .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
        .build())
    .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
        .lossFunction(LossFunctions.LossFunction.MSE)
        .build())
    .pretrain(false).backprop(true)
    .build();

```

This architecture has four layers, with the last layer having the full 784 units that the input layer has for the input data. We've set all of the activation functions in this network as ReLUs because we've found it best for working with this dataset.

Training the autoencoder network uses the same general pattern we've seen with the other DL4J examples in this chapter, as demonstrated here:

```

//Train model:
int nEpochs = 30;
for( int epoch=0; epoch<nEpochs; epoch++ ){
    for(INDArray data : featuresTrain){
        net.fit(data,data);
    }
    System.out.println("Epoch " + epoch + " complete");
}

```

We loop through the epochs of training and train on the data. The interesting variant in how we're using the API here is in this line:

```
net.fit(data,data);
```

Here, we notice that the data also is being used as the output for the network. This is because with autoencoders we're learning to reconstruct the data itself, so the data is both the input and the output, causing us to call the `fit()` method a little differently than in previous examples.

Evaluating the Model

When we run the example code, it generates an image for the best learned images and then another image for the images with the worst reconstruction error. [Figure 5-2](#) depicts the generated image for the best learned (e.g., the one that had the lowest reconstruction error) handwritten digit images.



Figure 5-2. Digits best learned by the autoencoder

The images that had the highest reconstruction error are composed in another render, as shown in [Figure 5-3](#).



Figure 5-3. Hard-to-learn handwritten digits

You can visually see how some of these images would certainly be considered anomalous compared to the other set of images. We don't always know what an anomaly might look like for our data, but if we base it on a model such as an autoencoder, we have a tool that doesn't need to see the anomalies beforehand.

Using Variational Autoencoders to Reconstruct MNIST Digits

In [Chapter 3](#), we introduced the concept of the variational autoencoder (VAE) as a way to reconstruct input data in an unsupervised learning fashion.

VAEs have many uses but the most common use cases include:

- Unsupervised learning as well as semisupervised learning of features, the general idea being a large amount of unlabeled data and a small amount of labeled data (we can get much better results than using labeled data alone, when labeled data is limited).
- Anomaly detection (unsupervised).
- As a generative model (e.g., to generate samples) they can generate images (as in this example), but also things like sentences.¹²

In [this example](#), we show you how to generate variations of the MNIST digit dataset to show the capabilities of VAEs as generative models.

Code Listing to Reconstruct MNIST Digits

Our next example ([Example 5-6](#)) is a simple example of training a VAE on MNIST to demonstrate the generative capabilities of VAEs. This example intentionally has a small hidden state Z (two values) for visualization on a two-dimensional grid.

After training, this example plots two things:

- The MNIST digit reconstructions versus the latent space
- The latent space values for the MNIST test set, as training progresses (every N mini-batches)

Note that for both plots, there is a slider at the top. Change this to see how the reconstructions and latent space changes over time.

Example 5-6. Modeling MNIST digits with VAEs in Java

```
public class VariationalAutoEncoderExample {  
    private static final Logger log =  
        LoggerFactory.getLogger(VariationalAutoEncoderExample.class);  
  
    public static void main(String[] args) throws IOException {  
        int minibatchSize = 128;  
        int rngSeed = 12345;
```

¹² Bowman et al. 2015. “Generating Sentences from a Continuous Space.”

```

int nEpochs = 150;                                //Total number of training epochs

//Plotting configuration
int plotEveryNMinibatches = 100;                  //Frequency with which to collect data
                                                //for later plotting
double plotMin = -4;                             //Minimum values for plotting
                                                //((x and y dimensions)
double plotMax = 4;                            //Maximum values for plotting
                                                //((x and y dimensions)
int plotNumSteps = 16;                           //Number of steps for reconstructions,
                                                //between plotMin and plotMax

//MNIST data for training
DataSetIterator trainIter = new MnistDataSetIterator(minibatchSize, true,
rngSeed);

//Neural net configuration
Nd4j.getRandom().setSeed(rngSeed);
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(rngSeed)
.iterations(1).optimizationAlgo(OptimizationAlgorithm
    .STOCHASTIC_GRADIENT_DESCENT)
.learningRate(1e-3)
//.updater(Updater.RMSPROP).rmsDecay(0.95)
.updater(Updater.ADAM)
.weightInit(WeightInit.RELU)
.regularization(true).l2(1e-5)
.list()
.layer(0, new VariationalAutoencoder.Builder()
.activation(Activation.LEAKYRELU)
.encoderLayerSizes(512, 512)           //2 encoder layers, each of
                                                //size 256
.decoderLayerSizes(512, 512)          //2 decoder layers, each of
                                                //size 256
.pzxActivationFunction("identity")   //p(z|data) activation function
.reconstructionDistribution(new BernoulliReconstructionDistribution(
    //Bernoulli distribution for p(data/z)
    //(binary or 0 to 1 data only)
    Activation.SIGMOID.getActivationFunction()))
.nIn(28 * 28)                         //Input size: 28x28
.nOut(2)                               //Size of the latent variable
                                                //space: p(z|x). 2 dimensions
                                                //here for plotting, use more
                                                //in general
.build())
.pretrain(true).backprop(false).build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();

//Get the variational autoencoder layer
org.deeplearning4j.nn.layers.variational.VariationalAutoencoder vae

```

```

= (org.deeplearning4j.nn.layers.variational.VariationalAutoencoder)
    net.getLayer(0);

//Test data for plotting
DataSet testdata = new MnistDataSetIterator(10000, false, rngSeed).next();
INDArray testFeatures = testdata.getFeatures();
INDArray testLabels = testdata.getLabels();
//X/Y grid values, between plotMin and plotMax
INDArray latentSpaceGrid =
    getLatentSpaceGrid(plotMin, plotMax, plotNumSteps);

//Lists to store data for later plotting
List<INDArray> latentSpaceVsEpoch = new ArrayList<>(nEpochs + 1);
//Collect and record the latent space values before training starts
INDArray latentSpaceValues = vae.activate(testFeatures, false);
latentSpaceVsEpoch.add(latentSpaceValues);
List<INDArray> digitsGrid = new ArrayList<>();

//Perform training
int iterationCount = 0;
for (int i = 0; i < nEpochs; i++) {
    log.info("Starting epoch {} of {}", (i+1), nEpochs);
    while (trainIter.hasNext()) {
        DataSet ds = trainIter.next();
        net.fit(ds);

        //Every N=100 minibatches:
// (a) collect the test set latent space values for later plotting
// (b) collect the reconstructions at each point in the grid
if (iterationCount++ % plotEveryNMinibatches == 0) {
    latentSpaceValues = vae.activate(testFeatures, false);
    latentSpaceVsEpoch.add(latentSpaceValues);

    INDArray out = vae.generateAtMeanGivenZ(latentSpaceGrid);
    digitsGrid.add(out);
}
}

trainIter.reset();
}

//Plot MNIST test set - latent space vs. iteration
//(every 100 minibatches by default)
PlotUtil.plotData(latentSpaceVsEpoch, testLabels, plotMin, plotMax,
    plotEveryNMinibatches);

//Plot reconstructions - latent space vs. grid
double imageScale = 2.0;           //Increase/decrease this to zoom in on the
                                //digits
PlotUtil.MNISTLatentSpaceVisualizer v =
    new PlotUtil.MNISTLatentSpaceVisualizer(imageScale, digitsGrid,

```

```

        plotEveryNMinibatches);
v.visualize();
}

//This simply returns a 2d grid: (x,y) for x=plotMin to plotMax, and y=plotMin
//to plotMax
private static INDArray getLatentSpaceGrid(double plotMin, double plotMax,
    int plotSteps) {
    INDArray data = Nd4j.create(plotSteps * plotSteps, 2);
    INDArray linspaceRow = Nd4j.linspace(plotMin, plotMax, plotSteps);
    for (int i = 0; i < plotSteps; i++) {
        data.get(NDArrayIndex.interval(i * plotSteps, (i + 1) * plotSteps),
            NDArrayIndex.point(0)).assign(linspaceRow);
        int yStart = plotSteps - i - 1;
        data.get(NDArrayIndex.interval(yStart * plotSteps,
            (yStart + 1) * plotSteps), NDArrayIndex.point(1)).assign(linspaceRow
                .getDouble(i));
    }
    return data;
}
}

```

In the sections that follow, we'll explain the generated latent space grid and the generated MNIST images.

Examining the VAE Model

Figure 5-4 presents the generated image produced by the code for this example. These are the reconstructions over the latent space at a specific iteration of training the network.



Latent Space Variables

In statistics, latent space variables are variables that are derived or inferred through a mathematical model. This is in contrast to variables that are observed.

You can generate different variations of the images based on the length the training runs and the training settings.

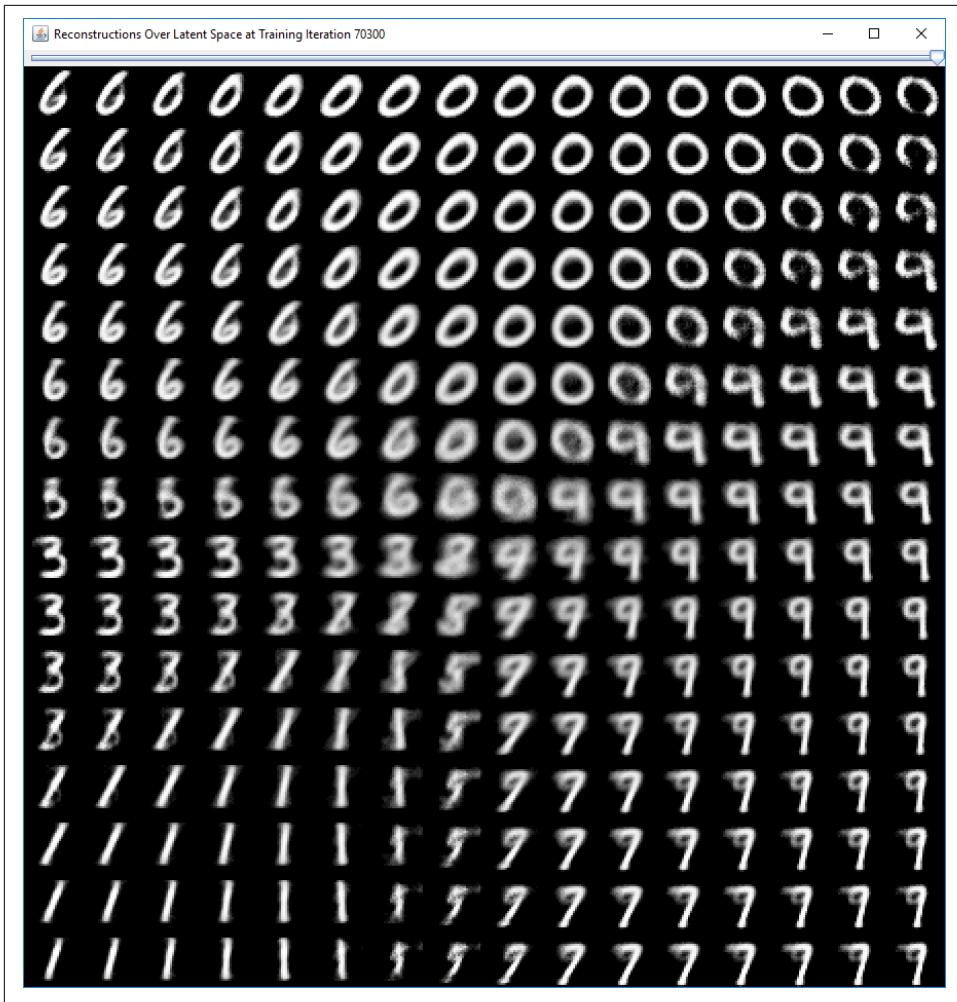


Figure 5-4. MNIST digits generated by the VAE example

To better understand how these images were generated, let's relate this example back to the network architecture for VAEs introduced in Chapter 3. Figure 5-5 shows the network architecture for the VAE.

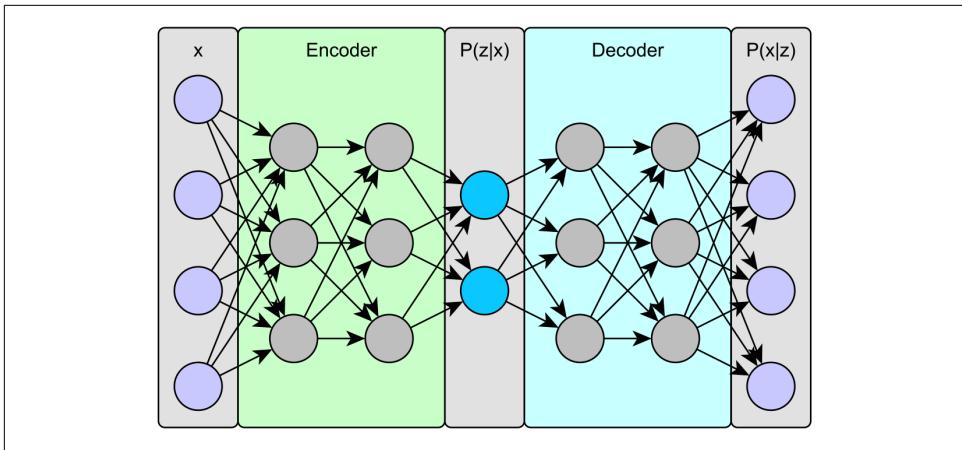


Figure 5-5. VAE network architecture

In Figure 5-6 the scatterplot of the VAE latent space of the MNIST test data is just the encoder part of the network shown in Figure 5-5.

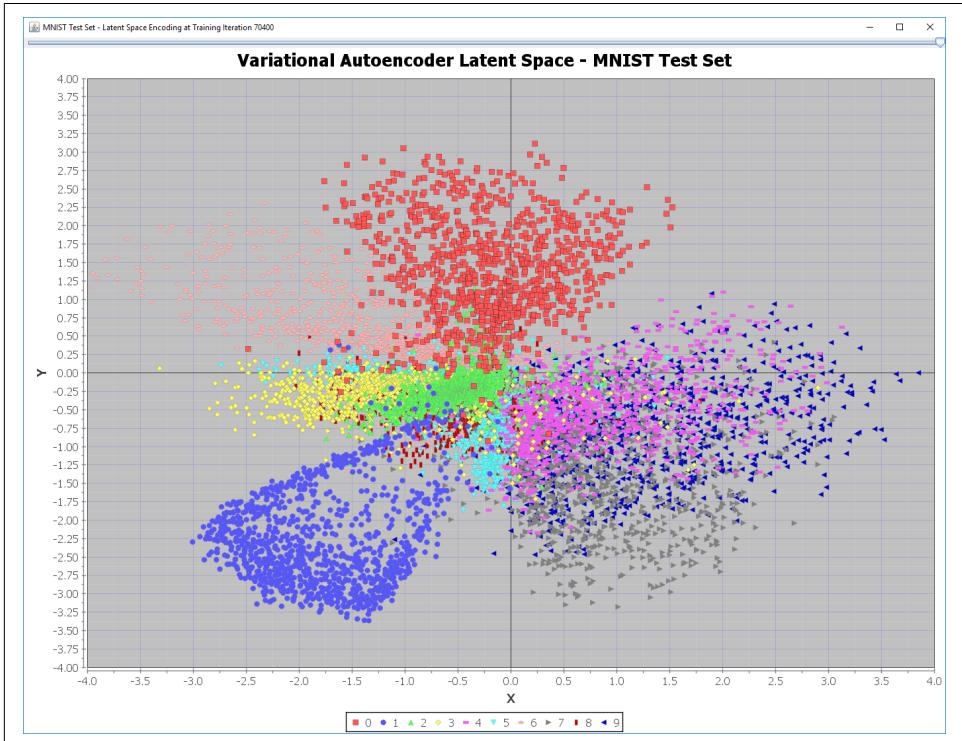


Figure 5-6. Scatterplot of the VAE latent space versus epoch on MNIST test dataset

Understanding the scatterplot

The scatterplot in [Figure 5-6](#) is the projection of the variable latent space versus the epoch. We project the MNIST test set data (e.g., x) into the values $p(z|x)$, where $p(z|x)$ is Gaussian with two values and z is the latent variable.

We can think of z as a probabilistic version of the bottleneck in a compression autoencoder. During training, we essentially randomly sample from z to give us actual numbers. Obviously, we can't forward pass a probability distribution, but we can forward pass numbers drawn from that probability distribution (e.g., a Monte Carlo estimate approach of sorts).

Our modeling assumption is that we begin by assuming that the data is generated by first starting with $p(z)$ (e.g., some probability distribution over the latent space and we sample from it). We then assume that there's some process that generates data x , using a distribution $p(x|z)$.

We take the mean of $p(z|x)$ and plot it on the scatterplot. In general, we'd have more than two values, but we are using two only because we want pretty pictures.

Understanding the generated images

The generated images in [Figure 5-4](#) are just the righthand side of the network in [Figure 5-6](#). Specifically, we generate a 16×16 grid of values in the range -4 to $+4$, and then we feed forward those values as if they were the mean value of $p(z|x)$.

To understand the render another way, we're just setting the z values and doing forward pass through the decoder to give us $p(x|z)$ (the MNIST data reconstructions).

In this case, $p(x|z)$ is a Bernoulli distribution and is bounded in the range 0 to 1. This means that we can just plot those probabilities (which also happens to be the mean value for the Bernoulli distribution), where 0 to 1 becomes 0 to 255 for pixel intensities. In this case, both x and $p(x|z)$ have the same size: 784 in/out (from MNIST's 28×28 size images).



The Relationship Between Scatterplot and Generated Images

In a sense the scatterplot and the generated images are the same thing, just in different directions.

Applications of Deep Learning in Natural Language Processing

Deep learning has shown to be effective in the area of Natural Language Processing (NLP). Techniques such as Part-of-Speech (POS) tagging,¹³ character generation, and learning word embeddings are common applications of deep learning. Here are some NLP applications we'll focus on in this chapter:

- Learning word embeddings by using Word2Vec¹⁴
- Distributed representations of sentences with paragraph vectors¹⁵
- Document classification with paragraph vectors

We'll look at examples of each of these in the sections that follow.

Learning Word Embedding Using Word2Vec

Word2Vec detects similarities between words mathematically by learning the context in surrounding words. Word2Vec creates vectors that are distributed numerical representations of word features, such as the context of individual words. Word2Vec trains on a text corpus as input data and produces a list of word vectors/embeddings as the output model. The word meanings and relationships in the produced word embeddings are encoded spatially and have useful characteristics such as vector arithmetic, as we'll see later.

The Word2Vec model and algorithm

The algorithm begins by building a vocabulary from the input training data and then builds the representations of the individual words. Initially, we're not building a vector per document in the same way we've done with the other vectorization techniques. However, with Word2Vec, the output dataset from training on the input corpus is the unique set of words in the corpus with a vector attached to each word. Each vector in this output contains the word's context (or usage). Word2Vec uses a feed-forward Neural Network Language Model (NNLM)¹⁶ to build these contextual word vectors in an unsupervised learning fashion. In practice, this is a two-layer neural network trained with SGD.

¹³ Nogueira dos Santos and Zadrozny. 2014. “Learning Character-level Representations for Part-of-Speech Tagging.”

¹⁴ Mikolov et al. 2013. “Efficient Estimation of Word Representations in Vector Space.”

¹⁵ Le and Mikolov. 2014. “Distributed Representations of Sentences and Documents.”

¹⁶ Mikolov et al. 2013. “Efficient Estimation of Word Representations in Vector Space.”

Word vectors represent a sequence of words in a way that allows us to model their context. They represent how a word is used with respect to the words around them. This is really handy when there is a lot of contextual information around the word to be classified in scenarios such as Named-Entity Resolution (NER), POS tagging, and Semantic Role Labeling (SRL).



Feature Count of Generated Word2Vec Vectors

Each word vector has about 50 to 300 features as the result of the distributed representation of words learned by the neural network.

Another technique in this space, Latent Semantic Indexing (LSI), detects dimensions that appear to go together and merges them into a single one, which helps speed computations like clustering. The designers of Word2Vec considered many approaches for estimating continuous representations of words, including Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA). Word2Vec has been shown to perform significantly better than LSA for preserving linear regularities among words, and is less computationally expensive than LDA.

Modeling context

Word vectors retain context of the source text around the word in the form of multi-word windows. For the purpose of machine learning, we define a word's meaning as the words that surround it most consistently. With enough data, we can use these semantics and build representations of the words in the corpus that fairly accurately model a word's meaning. Word2Vec creates features from windows of words where some of the features include the context of individual words.

When we build a Word2Vec model, we use a moving window across (typically) five words in the moving window at a time. The concept of using moving windows to train Word2Vec crosses over into many similar techniques, such as POS tagging, SRL, and NER. Word2Vec uses what's called a *Viterbi algorithm* to build the model. The Viterbi algorithm calculates the most likely sequence of events (labels) given the probability of going from one state to another or "a transition matrix."

Learning similar meaning and semantic relationships

We can find similar words by using a vector distance metric such as Euclidean distance (we described this in [Chapter 1](#)) or [cosine distance](#). This gives us words that have short cosine similarity distances or, more generally, "close representations." [Table 5-3](#) lists some examples of words most similar to "France" based on a trained Word2Vec model.

Table 5-3. Word2Vec cosine similarity

Word	Cosine distance
Spain	0.678515
Belgium	0.665923
Netherlands	0.652428
Italy	0.633130
Switzerland	0.622323
Luxembourg	0.610033
Portugal	0.577154
Russia	0.571507
Germany	0.563291
Catalonia	0.534176



Understanding Cosine Distance

A perfect match represented by the cosine distance of 1.0 would be between two of the same words (identity, or “Amsterdam” equals “Amsterdam”). The closer the cosine distance is to the value 1.0, the more similar the two word vectors’ meanings are.

We need to train our Word2Vec model on large datasets with sufficient vector dimensionality to see strong regularities in the word vector space.

Vector arithmetic and word embedding

We also can do some basic operations with the Word2Vec vectors. Here’s an example:

```
vector('Rome') = vector('Paris') - vector('France') + vector('Italy')
```

This operation results in a vector that is very close to `vector('Rome')`. Other interesting similarity semantics include showing the word “big” is similar to the word “bigger” in the same sense that “small” is similar to “smaller.” An interesting exercise is to calculate the word that is similar to small in the same sense as biggest is similar to big. We do this by computing simple algebraic operations with the vector representations of words:

```
vector('smallest') = vector("biggest") - vector("big") + vector("small")
```

We then search vector space for the word closest to our search term measured by our distance metric (cosine distance here) in a **1NN-search** fashion to find our word vector. When our word vectors are well trained, we should find the correct word (smallest) using this technique. We also can find subtle relationships such as a capital cities of countries (e.g., France is to Paris as Germany is to Berlin).

These semantic relationships can be effective in improving existing NLP techniques such as machine translation,¹⁷ information retrieval, and question answering¹⁸ systems. Let's now take a look at a basic example of using Word2Vec.

Java code listing for Word2Vec example

The code in Example 5-7 learns vector representation from raw text sentences. It's a small example to demonstrate the larger potential of learning the semantic meaning of words in a corpus.

Example 5-7. Word2Vec example

```
public class Word2VecRawTextExample {  
  
    private static Logger log =  
        LoggerFactory.getLogger(Word2VecRawTextExample.class);  
  
    public static void main(String[] args) throws Exception {  
  
        // Gets Path to Text file  
        String filePath = new ClassPathResource("raw_sentences.txt").getFile()  
            .getAbsolutePath();  
  
        log.info("Load & Vectorize Sentences....");  
        // Strip white space before and after for each line  
        SentenceIterator iter = new BasicLineIterator(filePath);  
        // Split on white spaces in the line to get words  
        TokenizerFactory t = new DefaultTokenizerFactory();  
  
        /*  
         * CommonPreprocessor will apply the following regex to each token:  
         * [|d|.:,''|()|[]//?!;]+  
         * So, effectively all numbers, punctuation symbols and some special symbols  
         * are stripped off.  
         * Additionally it forces lower case for all tokens.  
         */  
        t.setTokenPreProcessor(new CommonPreprocessor());  
  
        log.info("Building model....");  
        Word2Vec vec = new Word2Vec.Builder()  
            .minWordFrequency(5)  
            .iterations(1)  
            .layerSize(100)  
            .seed(42)  
            .windowSize(5)
```

¹⁷ Mikolov, Le, and Sutskever. 2013. “Exploiting Similarities among Languages for Machine Translation.”

¹⁸ Feng et al. 2015. “Applying Deep Learning to Answer Selection: A Study and An Open Task.”

```

        .iterate(iter)
        .tokenizerFactory(t)
        .build();

    log.info("Fitting Word2Vec model....");
    vec.fit();

    log.info("Writing word vectors to text file....");

    // Write word vectors to file
    WordVectorSerializer.writeWordVectors(vec, "pathToWriteto.txt");

    // Prints out the closest 10 words to "day". An example on what to do with
    //these Word Vectors.
    log.info("Closest Words:");
    Collection<String> lst = vec.wordsNearest("day", 10);
    System.out.println("10 Words closest to 'day': " + lst);

}
}

```

In the section that follow, we'll discuss specific parts of this code to better understand what is happening in the example.

Understanding the Word2Vec example

In the following code snippet, we see how the raw sentences file included with the code example repository is referenced and loaded by using a SentenceIterator object:

```

// Gets Path to Text file
String filePath = new ClassPathResource("raw_sentences.txt").getFile()
    .getAbsolutePath();

log.info("Load & Vectorize Sentences....");
// Strip white space before and after for each line
SentenceIterator iter = new BasicLineIterator(filePath);
// Split on white spaces in the line to get words
TokenizerFactory t = new DefaultTokenizerFactory();

/*
CommonPreprocessor will apply the following regex to each token:
[|d|.:,"'|()|[]|/?!;]+
So, effectively all numbers, punctuation symbols and some special symbols are
stripped off.
Additionally it forces lower case for all tokens.
*/
t.setTokenPreProcessor(new CommonPreprocessor());

```

Here, the preprocessing is performed by using the `CommonPreprocessor` object. Most of the text manipulation for this example is performed under the hood with these classes to make this a simpler example.

The Word2Vec architecture looks different than some of the previous network architectures in this chapter. This is because the `Word2Vec` class is largely self-contained in DL4J:

```
log.info("Building model....");
Word2Vec vec = new Word2Vec.Builder()
    .minWordFrequency(5)
    .iterations(1)
    .layerSize(100)
    .seed(42)
    .windowSize(5)
    .iterate(iter)
    .tokenizerFactory(t)
    .build();
```

You might also notice how the sentence iterator is set directly with the network architecture. This is not a major difference in functionality yet more of a legacy API design that has stayed around for this specific network.

Given that the sentence iterator was already set for this network we simply call the `.fit()` method with no other parameters to train the network, as in the following snippet:

```
log.info("Fitting Word2Vec model....");
vec.fit();
```

After the `.fit()` method completes, the example saves the network and prints out the 10 words closest to the word “day.”

Other practical uses of Word2Vec

The output of the Word2Vec neural network is a vocabulary in which each item has a vector attached to it, which can be fed into a deep learning net or simply queried to detect relationships between words. We also can use Word2Vec vectors for document classification, as we’ll see later in the chapter (using paragraph vectors).

To do canonical document classification, after we’ve generated word vectors from a corpus, we take every word occurring in the document we want to classify and sum their word vectors together. This gives a vector that represents the document while including the contextual information included with each word vector. We then use this document vector in the same way as a TF-IDF generated vector for a document would be used.

Learning Global Vectors for Word Representation by Using GloVe

An alternative method to Word2Vec is called **GloVe**.

GloVe is an unsupervised learning algorithm for building word embeddings. It builds the model based on aggregated global word-word co-occurrence statistics from the input data (e.g., group of text documents). The major difference to note is where Word2Vec is a “predictive” model, GloVe is a “count-based” model.¹⁹ It’s worth noting that GloVe is considered to be more difficult to train and tune as compared to Word2Vec.

DL4J has a [GloVe example](#) similar to the preceding Word2Vec exercise, which you can find in the [examples repository](#).

Distributed Representations of Sentences with Paragraph Vectors

If we continue with the ideas established with Word2Vec, we can expand this technique to model arbitrarily larger sequences (e.g., sentences or documents) to be modeled in a similar fashion. In this example, we’ll create paragraph vectors (e.g., Doc2Vec).

Historically, machine learning algorithms have required fixed-length representations of documents for modeling. One of the most common methods for doing this is the *bag-of-words* vectorization method. The *bag-of-words* method creates a count for every unique word in the document in a vector. This has two major issues:

- We lose the ordering of words
- We are not modeling the semantics of the words

With paragraph vectors, we learn a fixed-length vector representing variable length sequences of text (e.g., sentences, paragraphs, or documents). Each logical record is represented by a dense vector. Research²⁰ has shown the paragraph vector method to outperform the *bag-of-words* method and other text representation approaches. Paragraph vectors also have been shown to improve accuracy on several text classification and sentiment analysis problems.

In this example, we work with paragraph vectors in the Doc2Vec implementation in DL4J. Doc2Vec is an extension of Word2Vec that learns to correlate labels and words, rather than words with other words. Doc2Vec learns the input corpus as an unsuper-

¹⁹ Baroni, Dinu, and Kruszewski. 2014. “Don’t count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors.”

²⁰ Le and Mikolov. 2014. “Distributed Representations of Sentences and Documents.”

vised training method because we don't need to explicitly give it labels. Let's take a look at a code example using DL4J to model paragraph vectors.

Building paragraph vectors

In the [code](#) in [Example 5-8](#) we build a distributed representation (paragraph vectors) of all sentences present in the training corpus. The training data for the example is included in the example repository, so you need only execute the class after the repository has been cloned and is running locally.

Example 5-8. Building paragraph vectors by using DL4J in Java

```
public class ParagraphVectorsTextExample {  
  
    private static final Logger log =  
        LoggerFactory.getLogger(ParagraphVectorsTextExample.class);  
  
    public static void main(String[] args) throws Exception {  
        ClassPathResource resource = new ClassPathResource("/raw_sentences.txt");  
        File file = resource.getFile();  
        SentenceIterator iter = new BasicLineIterator(file);  
  
        AbstractCache<VocabWord> cache = new AbstractCache<>();  
  
        TokenizerFactory t = new DefaultTokenizerFactory();  
        t.setTokenPreProcessor(new CommonPreprocessor());  
  
        /*  
         * if you don't have LabelAwareIterator handy, you can use synchronized  
         * labels generator it will be used to label each document/sequence/line  
         * with its own label.  
         * But if you have LabelAwareIterator ready, you can provide it,  
         * for your in-house labels  
        */  
        LabelsSource source = new LabelsSource("DOC_");  
  
        ParagraphVectors vec = new ParagraphVectors.Builder()  
            .minWordFrequency(1)  
            .iterations(5)  
            .epochs(1)  
            .layerSize(100)  
            .learningRate(0.025)  
            .labelsSource(source)  
            .windowSize(5)  
            .iterate(iter)  
            .trainWordVectors(false)  
            .vocabCache(cache)  
            .tokenizerFactory(t)  
            .sampling(0)  
            .build();
```

```

vec.fit();

/*
In training corpus we have few lines that contain pretty close words
involved.
These sentences should be pretty close to each other in vector space
line 3721: This is my way .
line 6348: This is my case .
line 9836: This is my house .
line 12493: This is my world .
line 16393: This is my work .
this is special sentence, that has nothing common with previous sentences
line 9853: We now have one .
Note that docs are indexed from 0
*/

double similarity1 = vec.similarity("DOC_9835", "DOC_12492");
log.info("9836/12493 ('This is my house .')/'This is my world .') similarity:
" + similarity1);

double similarity2 = vec.similarity("DOC_3720", "DOC_16392");
log.info("3721/16393 ('This is my way .')/'This is my work .') similarity:
" + similarity2);

double similarity3 = vec.similarity("DOC_6347", "DOC_3720");
log.info("6348/3721 ('This is my case .')/'This is my way .') similarity:
" + similarity3);

// likelihood in this case should be significantly lower
double similarityX = vec.similarity("DOC_3720", "DOC_9852");
log.info("3721/9853 ('This is my way .')/'We now have one .') similarity:
" + similarityX + "(should be significantly lower)");
}

}

```

As you can see in the results, the first three similarity queries produce higher similarity scores than the last one, which is significantly lower.

Understanding the paragraph vectors example

The paragraph vectors example is similar structurally to the previous Word2Vec example in how we load the input training data, set up the network, and then train on the data. Again, the file and directory handling is done automatically by the included helper classes:

```

ClassPathResource resource = new ClassPathResource("/raw_sentences.txt");
File file = resource.getFile();
SentenceIterator iter = new BasicLineIterator(file);

AbstractCache<VocabWord> cache = new AbstractCache<>();

TokenizerFactory t = new DefaultTokenizerFactory();
t.setTokenPreProcessor(new CommonPreprocessor());

/*
    if you don't have LabelAwareIterator handy, you can use synchronized labels
    generator it will be used to label each document/sequence/line with
    its own label.
    But if you have LabelAwareIterator ready, you can provide it, for your
    in-house labels
*/
LabelsSource source = new LabelsSource("DOC_");

```

The paragraph vector implementation in DL4J uses a helper class called **Paragraph Vectors**, as shown here:

```

ParagraphVectors vec = new ParagraphVectors.Builder()
    .minWordFrequency(1)
    .iterations(5)
    .epochs(1)
    .layerSize(100)
    .learningRate(0.025)
    .labelsSource(source)
    .windowSize(5)
    .iterate(iter)
    .trainWordVectors(false)
    .vocabCache(cache)
    .tokenizerFactory(t)
    .sampling(0)
    .build();

vec.fit();

```

Key hyperparameters are set directly using methods on this class because **Paragraph Vectors** has a more dedicated API (like Word2Vec) as compared to the broader approach other networks use with the **MultiLayerConfiguration** class.

Again, we call the `.fit()` method, and at the end of the example, we see a few documents compared against one another with the cosine similarity metric (as we saw in the Word2Vec example). This example should give you a great feel for how to build paragraph vectors with DL4J. In the next example, let's take a look at how to incorporate paragraph vectors into a broader application using them for classification.



Generalizing Sequence Vectors

Paragraph vectors are an implementation of *sequence vectors*. The DL4J project has a **general purpose implementation** of skipgram that underlies models such as Word2Vec, paragraph vectors, and DeepWalk.²¹

Hinton on the Potential Implications of Paragraph Vectors

In a 2015 speech to the Royal Society in London, Geoffrey Hinton said the following:

The implications of this for document processing are very important. If we convert a sentence into a vector that captures the meaning of the sentence, then Google can do much better searches; they can search based on what's being said in a document.

Also, if you can convert each sentence in a document into a vector, then you can take that sequence of vectors and [try to model] natural reasoning. And that was something that old fashioned AI could never do.

If we can read every English document on the web, and turn each sentence into a thought vector, you've got plenty of data for training a system that can reason like people do.

Now, you might not want it to reason like people do, but at least we can see what they would think.

What I think is going to happen over the next few years is this ability to turn sentences into thought vectors is going to rapidly change the level at which we can understand documents.

To understand it at a human level, we're probably going to need human level resources and we have trillions of connections [in our brains], but the biggest networks we have built so far only have billions of connections. So we're a few orders of magnitude off, but I'm sure the hardware people will fix that.

Using Paragraph Vectors for Document Classification

As mentioned earlier in the chapter, we can use paragraph vectors in the context of NLP and document classification. In this example, we'll use a paragraph vector model similar to the one in the last example in an application to build a document classifier that gives us one of three labels as output similar to the following:

```
Document 'health' falls into the following categories:  
health: 0.29721372296220205  
science: 0.011684473733853906  
finance: -0.14755302887323793
```

²¹ Perozzi, Al-Rfou, and Skiena. 2014. “DeepWalk: Online Learning of Social Representations.”

As you can see in the preceding snippet, the label with the highest score is our classification. Let's take a look at [Example 5-9](#) for our paragraph vectors classification application.

Example 5-9. Using paragraph vectors to classify documents in Java

```
public class ParagraphVectorsClassifierExample {

    ParagraphVectors paragraphVectors;
    LabelAwareIterator iterator;
    TokenizerFactory tokenizerFactory;

    private static final Logger log =
        LoggerFactory.getLogger(ParagraphVectorsClassifierExample.class);

    public static void main(String[] args) throws Exception {

        ParagraphVectorsClassifierExample app =
            new ParagraphVectorsClassifierExample();
        app.makeParagraphVectors();
        app.checkUnlabeledData();
        /*
           Your output should be like this:
           Document 'health' falls into the following categories:
           health: 0.29721372296220205
           science: 0.011684473733853906
           finance: -0.14755302887323793
           Document 'finance' falls into the following categories:
           health: -0.17290237675941766
           science: -0.09579267574606627
           finance: 0.4460859189453788
           so, now we know categories for yet unseen documents
        */
    }

    void makeParagraphVectors() throws Exception {
        ClassPathResource resource = new ClassPathResource("paravec/labeled");

        // build a iterator for our dataset
        iterator = new FileLabelAwareIterator.Builder()
            .addSourceFolder(resource.getFile())
            .build();

        tokenizerFactory = new DefaultTokenizerFactory();
        tokenizerFactory.setTokenPreProcessor(new CommonPreprocessor());

        // ParagraphVectors training configuration
        paragraphVectors = new ParagraphVectors.Builder()
            .learningRate(0.025)
            .minLearningRate(0.001)
            .batchSize(1000)
```

```

    .epochs(20)
    .iterate(iterator)
    .trainWordVectors(true)
    .tokenizerFactory(tokenizerFactory)
    .build();

    // Start model training
    paragraphVectors.fit();
}

void checkUnlabeledData() throws FileNotFoundException {
    /*
    At this point we assume that we have model built and we can check
    which categories our unlabeled document falls into.
    So we'll start loading our unlabeled documents and checking them
    */
    ClassPathResource unClassifiedResource =
        new ClassPathResource("paravec/unlabeled");
    FileLabelAwareIterator unClassifiedIterator = new FileLabelAwareIterator
        .Builder()
            .addSourceFolder(unClassifiedResource.getFile())
            .build();

    /*
    Now we'll iterate over unlabeled data, and check which label it could be
    assigned to
    Please note: for many domains it's normal to have 1 document fall into few
    labels at once, with different "weight" for each.
    */
    MeansBuilder meansBuilder = new MeansBuilder(
        (InMemoryLookupTable<VocabWord>)paragraphVectors.getLookupTable(),
        tokenizerFactory);
    LabelSeeker seeker = new LabelSeeker(iterator.getLabelsSource().getLabels(),
        (InMemoryLookupTable<VocabWord>) paragraphVectors.getLookupTable());

    while (unClassifiedIterator.hasNextDocument()) {
        LabelledDocument document = unClassifiedIterator.nextDocument();
        INDArray documentAsCentroid = meansBuilder.documentAsVector(document);
        List<Pair<String, Double>> scores = seeker.getScores(documentAsCentroid);

        /*
        please note, document.getLabel() is used just to show which document we're
        looking at now, as a substitute for printing out the whole document name.
        So, labels on these two documents are used like titles,
        just to visualize our classification done properly
        */
        log.info("Document '" + document.getLabel()
            + "' falls into the following categories: ");
        for (Pair<String, Double> score: scores) {
            log.info("           " + score.getFirst() + ":" + score.getSecond());
        }
    }
}

```

```
    }
}
```

The overall idea in [Example 5-9](#) is to use the `ParagraphVectors` class in the same way that we use [LDA](#) (e.g., “topic space modeling”).

In this example, we assume that we have few labeled categories that we can use for training, and then a few unlabeled documents. Our goal with the application is to determine which category these unlabeled documents fall into by taking advantage of information contained in the paragraph vector.

Understanding the paragraph vectors classification example

This example has some similar mechanics as the previous example yet has been extended to classify unlabeled documents. Here's the main driver code:

```
public static void main(String[] args) throws Exception {
    ParagraphVectorsClassifierExample app =
        new ParagraphVectorsClassifierExample();
    app.makeParagraphVectors();
    app.checkUnlabeledData();
```

You can see how the previous example has been wrapped by the `.makeParagraphVectors()` method. The next line proceeds to use this paragraph vector model to classify new unlabeled documents. Within the `.checkUnlabeledData()` method is the list of documents being classified inside the loop with the code:

```
LabelledDocument document = unClassifiedIterator.nextDocument();
INDArray documentAsCentroid = meansBuilder.documentAsVector(document);
List<Pair<String, Double>> scores = seeker.getScores(documentAsCentroid);
```

The next lines take the generated scores and print them to the screen:

```
log.info("Document '" + document.getLabel() + "'"
    + " falls into the following categories: ");
for (Pair<String, Double> score: scores) {
    log.info("      " + score.getFirst() + ":" + score.getSecond());
}
```

To produce the output:

```
Document 'health' falls into the following categories:
  health: 0.29721372296220205
  science: 0.011684473733853906
  finance: -0.14755302887323793
```

The highest score

```
health: 0.29721372296220205
```

would be considered our classification. You can clearly see in the preceding output the document was classified as “health,” which is correct because it matches the associated label in the data.

Further exploration of the Word2Vec approach

As we mentioned earlier in the chapter with regard to CNNs, many of the applications referenced here for different architectures are just a good starting place. In general terms, embeddings are mappings of relationships in data as learned by a neural network. Other interesting variants of the Word2Vec approach include applications in the following areas:

- Extensions into specific domains
- Graph analysis
- Recommendations
- Image identification

Extensions into specific domains: Gov2Vec. One interesting application of Word2Vec is an extension into the domain of legal text analysis in government called Gov2Vec. This is description from the author’s paper:²²

We compare policy differences across institutions by embedding representations of the entire legal corpus of each institution and the vocabulary shared across all corpora into a continuous vector space. We apply our method, Gov2Vec, to Supreme Court opinions, Presidential actions, and official summaries of Congressional bills.

This variant of neural word embeddings is able to understand the relationships to how Congress and the President might or might not veto a bill, and how that evolves over time. They are able to answer such questions as the following:

How does Obama and the 113th House differ in addressing climate change and how does this vary from environmental or economic perspectives?

Vector arithmetic approaches are something we’ll most likely see more of as deep learning evolves.

Graphs and Node2Vec. In the domain of graph analysis, there is Node2Vec,²³ an algorithmic framework for learning continuous feature representations for nodes in networks. An interesting property of Node2Vec is that it is scalable to very large graphs (millions of nodes/edges or more).

²² Nay. 2016. “[Gov2Vec: Learning Distributed Representations of Institutions and Their Legal Text](#).”

²³ Grover and Leskovec. 2016. “[node2vec: Scalable Feature Learning for Networks](#).”

Recommendation engines and Item2Vec. Another interesting application of the concept is Item2Vec, in which the approach is used for recommendation systems.²⁴ In this paper, the authors show the neural word embedding approach used as an item-based collaborative filtering technique for recommendations.

Computer vision and FaceNet. In the arena of image identification, there is FaceNet,²⁵ in which we see an application of neural network embeddings used to generate representation of **facial features (images)**.

Overall, the Word2Vec approach is an interesting method to use embeddings to map the relationships between input data. It allows us to further use data for clustering, classification, and comparison.

²⁴ Barkan and Koenigstein. 2016. “Item2Vec: Neural Item Embedding for Collaborative Filtering.”

²⁵ Schroff, Kalenichenko, and Philbin. 2015. “FaceNet: A Unified Embedding for Face Recognition and Clustering.”

CHAPTER 6

Tuning Deep Networks

All things are poisons, for there is nothing without poisonous qualities. It is only the dose which makes a thing poison.

—**Paracelsus**, fifteenth-century Renaissance physician, botanist, alchemist, astrologer, and occultist

Basic Concepts in Tuning Deep Networks

In this chapter, we take a look at the methods and strategies for training neural networks. We examine the following:

- Matching network architecture to the problem at hand
- The basics of hyperparameter tuning
- Better understanding the process of learning

Obviously, this chapter can't be comprehensive of the entire breadth of published tuning work in the space of deep learning. Our strategy was to sample the most relevant material and create a narrative that exposes you to the core concepts in deep architecture tuning. Chapter 7 then specifically focuses on tuning techniques for the most well-known architectures in deep networks:

- Deep Belief Networks (DBNs)
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks

Let's begin with a general idea, or intuition, on how we want to approach building different neural networks with different goals.



Tuning Restricted Boltzmann Machines

In this chapter, we cover tuning Restricted Boltzmann Machines (RBMs) in the context of tuning DBNs.

An Intuition for Building Deep Networks

As we get underway, you should ask yourself two questions:

- What kind of input data will I be modeling?
- What is the output I want to know after this model is constructed?

Understanding what kind of data you're modeling will largely dictate what class of deep learning architecture and input layer you need to implement. Determining what you want to know from data of this class will inform you as to whether you want to get class scores (classify something with probabilities) or produce a real-valued target output (regression). This also will determine what kind of output layer you need to use. There are variations and caveats in many corners of deep networks, but these two questions will help get you started down the path of constructing a well-designed foundation for your deep network architecture. When you understand these two things, you can move to the next tier of design decisions, which focus on setting up the parameters themselves:

- Number of layers
- Parameter count per layer

You also need to take into consideration the memory requirements needed for a particular network architecture decision based on parameter counts. The number of layers and parameter count per layer will ultimately express how much capacity the network has to represent the structure in the data. For some problems, we can represent a surprisingly complicated model with relatively few parameters (total neuron count).

Now that you have an architecture and layers, here are some other items to consider:

- Weight initialization strategy
- Activation function
- Loss function
- Optimization algorithm
- Mini-batching
- Regularization

The weight initialization strategy typically corresponds to our architecture type along with the input data type. Initializing the weights well or poorly can help or hinder the learning process. You'll need an activation function for each layer to model nonlinear

relationships between the input and output data. Activation functions help you to learn certain types of features along with setting up your output layers to give you answers like regression or classification. You want to match certain layer types (activation function types) with certain loss functions in specific architectural situations.



Affecting Learning with Loss Functions

Loss functions are how you define what you want the network to learn (classification or regression, for example). You need to match them with both an appropriate activation function and the type of data/labels you have.

Depending on your problem space, you'll also want to consider different types of optimization methods.

Regularization methods help prevent the model from getting too focused on noise in the data and keep the weights as small as possible to generalize across the full data population—for example, examples not seen during training. As we'll see in this chapter, many of the aforementioned design decisions are tied together or have an impact on all other architectural decisions of the network. We'll spend the bulk of this chapter explaining how these design interdependencies work. Let's now formalize some of these high-level ideas into a more distinct step-by-step process that you can use as a guide for network architecture construction.

Building the Intuition as a Step-by-Step Process

The previous section discussed in general what we need to do to build a deep network architecture. In the steps that follow, we formalize the process in a guide that you can use for most any neural network modeling problem.

1. Determine what our input data should be:
 - a. The input data type informs us as to architecture
2. Determine what our intended result should be:
 - a. Gives us guidance on configuring architecture
 - b. Determines the output layer type
3. Set up architecture of network to support problem:
 - a. Choice of model, architecture, and cost function are all important
 - b. Depending on the architecture, we'll select a number of hidden layers
 - c. Choose activations per layer based on the overall architecture of the network and the intent of the specific layer
4. Work with training data to handle the following tasks:
 - a. Clean data
 - b. Produce visualizations
 - c. Perform vectorization and normalization

- d. Balance classes (as necessary)
 - e. Create test, train, and validate splits
5. Develop a hyperparameter tuning strategy with a balanced subset of the data:
 - a. Increase the subset data size and tweak hyperparameters as needed
 6. If the final training dataset is large, use Spark to train on more data faster (where appropriate).

These steps are still high level in terms of specific tuning detail, but they give us a general path to follow when building any deep network. By the end of this chapter, we hope that you will be armed with a good set of basic tuning principles to approach the emerging domains in data science today. Let's now dig into the specifics of deep network tuning, beginning with the first step in our process: matching input data and network architectures.

Matching Input Data and Network Architectures

As discussed in the previous section, we want to begin our deep network design process by thinking in terms of what our source dataset represents. Here are some examples:

- Columnar data
- Image data
- Audio data
- Video data
- Time-series data

For columnar—or, comma-separated values (CSV)—data, typically we're dealing with a relational database management system (RDBMS) export of a table or the result-denormalized table of multiple sets of data joined together, as shown in the following example:

```
M,0.455,0.365,0.095,0.514,0.2245,0.101,0.15,15  
M,0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,7  
F,0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,9  
M,0.44,0.365,0.125,0.516,0.2155,0.114,0.155,10  
I,0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,7  
I,0.425,0.3,0.095,0.3515,0.141,0.0775,0.12,8
```

We have neither image pixels to extract features from nor any time dependencies to deal with (time-series data), so our architecture can be relatively vanilla. Here, we recommend that you begin with a simple multilayer perceptron neural network.

For image classification tasks, we want to use a CNN. In recent years, CNNs have demonstrated that they provide best-of-class results for image processing tasks (refer to [Chapter 4](#) for more on this).

Sequential data is any situation in which we have a series of inputs to our model. This is commonly seen in the wild as log data produced by servers or sensors. Time-series data also falls into this bucket. We define time-series data as any data that has a series of values each coupled with a value of time. These values, ordered chronologically, track an entity's activity over time. In most machine learning models, we need to do some sort of feature extraction to produce one single input vector per set of time-series data for our training algorithm. For any sequential or time-series data, you should plan to use Recurrent Neural Networks, which have the ability to model N number of input vectors and are not restricted to a single input vector (refer to [Chapter 4](#) for a refresher). This gives us the ability to model activity over time.

We typically see good results on audio input data with Recurrent Neural Networks. Audio data has a temporal nature to it in which we see a time-series of audio samples to produce a waveform, which makes it a good match for Recurrent Neural Networks.

When working with video data, we need to do more elaborate computational gymnastics to extract answers from the series of images. One way to approach this is to combine convolutional, max-pooling, dense (feed-forward) and recurrent (Long Short-Term Memory [LSTM]) layers to classify each frame of a video. Another approach is to extract the individual frames of the video into image files and then use CNNs to analyze the individual frames.



Dealing with Video Data

Using optical flow preprocessing helps a lot in practice. It helps to capture some short-term temporal relationships in the data (e.g., movement between consecutive frames) that are not present in a single frame.

Summary

We've provided you with an idea of where to begin with regard to different network architectures, given a specific data input type. [Table 6-1](#) sums it all up neatly.

Table 6-1. Matching input type to network architecture

Input data type	Suggested architecture
Columnar aka CSV data	Multilayer perceptron
Image	CNN
Sequential	Recurrent Neural Network—LSTM, specifically
Audio	Recurrent Neural Network—LSTM, specifically
Video	Hybrid CNN + Recurrent Neural Network



The Revolving Door of Network Architectures

Although the suggested architecture for each data type might work well, be aware that there is a large amount of ongoing research. New variations of architectures are published at a rapid pace these days, so you should consider the suggestions presented here as nice starting points but also be on the lookout for research extending these architectures.

In the next section, we further flesh-out our architecture heuristics by looking at how to set up layer and neuron counts.

Relating Model Goal and Output Layers

The previous section discussed matching our input data type to a specific neural network architecture. Now, it's time to examine output layers, which involve more thought because they are based on what kind of answer we expect from the model we produce.

Every layer has an associated activation function for passing information to the next layer. If we were trying to get output from the network, we'd have a final layer called an "output layer" and set its activation function type according to our intended output or answer (e.g., classification or regression, as we'll see momentarily).

Regression Model Output Layer

In regression models, we're generating a real-valued output number such as the price of a house based on its square footage. The two main considerations for a regression model's output layer are the loss function and the activation function.

Loss function

For the regression output layer loss function, we have multiple valid choices. The most common are mean squared error (MSE) or sum of squared errors (L2).

Activation function

In this case, we use the identity (linear) output function.

Regression Output Layer and Other Edge Cases

Sometimes, we see the tanh activation function used for regression output layers (only if all data is guaranteed to be in the range $[-1, 1]$) and then softplus or a rectified linear variant (leaky ReLU, randomized leaky ReLU) if data is guaranteed to be in the range $[0, \infty)$.

What about Rectified Linear Unit (ReLU) for regression, for which the labels data is in the range $[0, \infty)$? Though the activation function itself can produce the correct

range of output values—that is, also $[0, \infty)$ —there is another problem to be aware of: the so-called “dying ReLU” problem.

Essentially, there’s a chance that a ReLU can become stuck in the zero output section of the activation function. If this occurs, the output for that ReLU (our network prediction) will always be 0, regardless of the network inputs. Other ReLU variants (leaky and randomized leaky) and softplus don’t have this issue.

Classification Model Output Layer

In classification models, we have N number of output units in the output layer and we produce a class score for each output unit. If $N = 1$, then we have a model with a single label. In this case, we’d be classifying the presence of a condition or no condition (e.g., spam versus not spam). If $N > 1$ then we’re scoring the input with respect to each of the classes, and we use a different output layer configuration. In this variation, we might be classifying a document as to what category—sports, business, politics—it belongs to. There is also the case for which the document could belong to multiple categories—for example, sports and business.

Single-label classification models

In the basic single-label example, or binary classifier, we want to use a sigmoid activation function for our output layer. This will give us a single output with a value somewhere between 0.0 and 1.0, indicating whether the document is indeed spam.

When working with a single output label (for which we have a single value in the range of 0.0 to 1.0 for two classes), use the cross-entropy loss function for the output layer.

Single Output or Two Outputs for a Binary Classifier?

There is a variation for which we might use a softmax activation function with the output layer with two outputs for a binary classifier. In this situation, the output is going to be two double values that sum to 1.0, and the larger value is the index of the label. In this variation, with softmax we want to use the MCXENT loss function.

There is debate around which output layer is the best way to model binary classification models. Mathematically, using a sigmoid single output is the same as using a softmax output layer with two output units (with MCXENT/NegativeLogLikelihood and one-hot representation, [1,0] or [0,1] instead of 0 or 1).

Models with more than two labels

There are two situations we need to consider when we have more than two labels:

- Sometimes we have multiple labels and we want the model to give us the most likely single label. This is called **multiclass classification**.
- Other times we have multiple labels and we want multiple labels per output (e.g., person + car). This is called **multilabel classification**.

We look at each scenario below.

Multiclass classification models. Recalling our discussion from [Chapter 2](#), if we have a multiclass modeling problem yet we care only about the best score across these classes, we'd use a softmax activation function for the output layer. The number of output units is equal to the number of classes we want to classify in this scenario.

The outputs of all the units in the output layer will sum to 1.0. Given our outputs are probabilities per class, we probably won't see exactly 1.0 or exactly 0.0 per output, but some value in between 0.0 and 1.0. We select the class (output unit) with the highest probability as our classification result. We use an `arg-max()` function to get the *index* of the class with the highest score (the predicted class).

We see part of a neural network set up in the following example with a softmax activation function on the output layer:

```
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
      .weightInit(WeightInit.XAVIER)
      .activation(Activation.SOFTMAX)
      .nIn(numHiddenNodes).nOut(numOutputs).build())
```

In the specific output layer section in this example, we see the layer type being specified with the `.activation()` method.

Most of the time, we'll see negative log likelihood used with the softmax activation function, as we saw in the previous code snippet. Sometimes, we might see multiclass cross-entropy used similarly, but it's equivalent to negative log likelihood.

The data representation should be “one-hot” for the output. This would be relevant when building the input/output training vectors because we need to remember to convert the output vector to the one-hot representation. DL4J's `RecordReaderDataSetIterator` can do this class index (0 to [number of classes – 1]) to the one-hot vector conversion for us automatically.



Dealing with a Large Number of Labels

For the case in which the number of intended labels reaches the ten-thousands, we'd want to use a hierarchical softmax output layer. Hierarchical softmax¹ is a fast(er) approximation to softmax suitable for a large number of classes.² We should consider it for computational reasons and not to improve accuracy.

Multilabel classification models. If we want to get multiple classifications per output (e.g., “person + car”), we do not want softmax as an output layer activation function. Instead, we'd use the sigmoid activation function for our output layer with multiple output units, each representing a different class. Each unit then gives us a probability for every class independently.

Again, we'd use the binary cross-entropy (`LossFunction.XENT`) loss function for the output layer in this scenario.

The data representation we want to use for the output vector in the training data would be a 0 or 1 for each class. This would differ from the one-hot representation mentioned earlier because we could have multiple label entries as 1.0, whereas one-hot can have only 1.



Multiple Output Layers

Suppose we wanted to use a neural network to predict the make of a car (Ford, Toyota, GM, etc.) and also its type (SUV, sport, truck, etc.). In this scenario, we'd need to use ComputationGraph + multiple output layers. For more information, go to <https://deeplearning4j.org/compgraph#multitask>.



Quick Reference: Softmax Versus Sigmoid Activation Functions

The softmax activation function bounds the sum of the outputs to be 1.0 (i.e., probabilities) versus the sigmoid activation function, which bounds the output values individually. For example, the probabilities of the output of a softmax layer will all add up to 1.0 yet the output of a sigmoid layer might have every unit with 0.9 as the output. Sigmoid layers do not have this “lateral” constraint.

¹ See Morin and Bengio. 2005. “Hierarchical Probabilistic Neural Network Language Model” and Mikolov et al. 2013. “Distributed Representations of Words and Phrases and their Compositionality.”

² Hierarchical softmax is not the only option. An alternative that solves the same problem is seen in Vishwanathan et al. 2015. “BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies.”

Working with Layer Count, Parameter Count, and Memory

In neural networks, the number of layers and neurons per layer determine how many total parameters the model has. The number of total parameters in the network is dependent on the types of layers and the number of connections (or weights) between the neurons in each layer (plus the bias weights).



Layer Type and Neuron Count

The type of layer involved also can affect the number of parameters in the network. For given layer input/output sizes, LSTM layers have many more neurons than a vanilla Recurrent Neural Network layer, which has more than a `DenseLayer` in DL4J. An example would be how a `DenseLayer` has more parameters than a similarly sized CNN layer.

As the number of parameters grows, the function we can model becomes more complex. After a point, the number of parameters becomes so large that we create a model that overfits too many of the nuances of the dataset and won't generalize as well.

For different network architectures, there are variations in how the layers, neurons, and connection weights are set up. There are some high-level ideas about neural networks that mainly apply to feed-forward multilayer perceptron neural networks. We cover the specific architectures later in the chapter.

Feed-Forward Multilayer Neural Networks

For feed-forward multilayer perceptron neural networks, input layers are required to have the same number of input units as the input vector. The output layer will simply be equal to the number of labels for classification and a single neuron for regression. In the sections that follow, we discuss strategies for determining layer and neuron counts for neural networks.

Determining hidden-layer count

Hidden-layer count and dataset size have a relationship, as well. As our dataset size grows, we want to have more hidden layers. An example of this would be that MNIST needs only around three to four hidden layers (with accuracy decreasing beyond that depth), but Facebook's DeepFace uses nine hidden layers on what we can only guess to be a very large dataset.



Rule of Thumb for Hidden-Layer Count

The rule of thumb for hidden layers is the larger the dataset, the more hidden layers and neurons we can use without overfitting. With more training data, we're less likely to overfit. We can have a larger network and possibly achieve better accuracy. If the network size is too small with a larger input dataset, we run the risk of underfitting and our accuracy will not reach its best case for the dataset.

Another factor that can drive the layer and parameter count up even higher is larger variation in the dataset.

Determining neuron count per layer

If our network has too few neurons in the hidden layers, we'll struggle to model the data well during training. If our network has too many parameters, we will either deal with overfitting issues or take more processing time to find a good model fit.

Here's a good way to think about neuron count: the neuron count of progressive layers should decline. We also want to be careful such that no hidden layer is less than a quarter of the input layer's nodes. When we have too many hidden-layer neurons, we also increase the chance that we'll overfit the dataset.



Rule of Thumb for Layer Neuron Counts

We want the right combination of layer size, regularization, and amount of data. Large layers plus insufficient regularization will hurt generalization. Larger (and more) layers (technically, more parameters) often need more aggressive regularization (and/or more data) to avoid overfitting.

In some cases, using the same number of hidden parameters for all layers as opposed to a decreasing neuron count works generally better; however, this effect might work only on certain types of datasets.

Controlling Layer and Parameter Counts

It is a common (initial) idea to remove parameters (or neurons) from a neural network to solve training issues such as overfitting. In practice, conversely, more often than not we want to select a hidden-neuron count above the optimal value.

It has been empirically shown that the optimal number of hidden neurons is much larger when using unsupervised pretraining (RBMs in DBNs) neural networks. An example of this would be a hidden-neuron count going from hundreds of units to

thousands of units.³ Instead, to combat overfitting it is suggested to use regularization techniques such as the following:

- L1
- L2
- Dropout
- Input noise
- DropConnect

The reasoning is that smaller networks will have few local minima (but possibly bad models) and larger networks will have more local minima.⁴ We don't want to use a smaller network just because we fear overfitting the training dataset. Instead, throw as many parameters at the problem as our computational platform will allow and use the aforementioned techniques to combat overfitting.



Don't Go Overboard with Parameters

Throw more parameters at the problem but be pragmatic about it. Having a single layer with 1,000,000 neurons might not be the best idea, even though the hardware could handle it.

As the variation and size of the training dataset grows, we recommend that you slowly increase the hidden-layer count and the neuron count per layer.



Adding More Data to Control Overfitting

One of the best ways to control overfitting is to use more training data (where possible).

Getting the parameter count for a network

There are two ways in practice to get the parameter count for a network:

- Hand-calculate the number for an architecture
- Use the DL4J API to calculate the parameter count

We can determine the number of parameters in a network by simply adding up the number of parameters in each layer. [Table 6-2](#) gives the number of parameters for some commonly used layers.

³ Bengio. 2012. “Practical Recommendations for Gradient-Based Training of Deep Architectures,” in Muller et al. 2012. *Neural Networks: Tricks of the Trade, Second Edition*.

⁴ Li, Karpathy. “CS231n: Convolutional Neural Networks for Visual Recognition.” (Course Notes)

Table 6-2. Calculating parameter count per network layer type

Layer type	Number of parameters
Fully connected (e.g., Dense, MLP, OutputLayer in DL4J)	$n^{L-1}n^L + n^L$
Convolutional	$d^{L-1}d^H k^W + d^L$
LSTM RNN (e.g., GravesLSTM)	$4n^L(n^{L-1} + 1) + n^L(4n^L + 3)$
Standard (vanilla) RNN	$n^{L-1}n^L + (n^L)^2 + n^L$

Here's how the parameters are calculated:

n^{L-1}

Number of inputs (size of previous layer, $L - 1$)

n^L

Size of the current layer L

k^H and k^W

Convolutional layer kernel sizes in the height and width dimensions

d^{L-1} and d^L

The input and output depths/channels for the convolutional layer

Note that some layer types don't have any parameters: for example, subsampling and activation layers as well as DL4J's `LossLayer`. One complication here is for networks combining both convolutional layers and dense (or output) layers. For the dense (or output) layer, after a convolutional/subsampling layer, we need to establish the size of the activations array of the last convolutional layer. We want to use its length (per example) as the n^{L-1} value for the dense/output layer. For example, if our final convolution layer outputs 5×5 activation with 100 channels, we should use $n^{L-1} = 5 \times 5 \times 100 = 2,500$ as the input to the dense/output layer.⁵

Suppose that we have a DL4J configuration. How do we check the number of parameters? Here's a simple way:

```
MultiLayerConfiguration configuration = ...
MultiLayerNetwork network = new MultiLayerNetwork(configuration);
network.init();

System.out.println("Total number of parameters: " + network.numParams());
for( int i=0; i<network.getnLayers(); i++ ){
    System.out.println("Layer " + i + " number of parameters: " +
```

⁵ For more details on calculating input sizes, see Stanford's CS231n notes and DL4J's `ConvolutionMode` Javadoc. (Note alternatively you can use DL4J's `InputType` functionality to have this `numInputs` value calculated and set for you, and use DL4J to get your network configuration.)

```
    network.getLayer(i).numParams());  
}
```

Estimating Network Memory Requirements

If our network configuration is too large for the hardware we are running on, training will fail when the memory is completely consumed. Though not strictly related to tuning, it's an important topic for large networks—after all, we can't tune our network if we can't train it in the first place.

When training a neural network, memory use comes from the allocation of multi-dimensional arrays (INDArrays in DL4J). There are six different uses for these arrays when training a neural network:

- Network parameters
- Parameter gradients (same size as network parameters)
- Network activations
- Network activation gradients (same size as network activations)
- Updater state (i.e., history for Momentum or RMSProp—an integer multiple of the number of parameters)
- Training data

In addition, we need to consider overhead for working memory/temporary arrays, shape buffer information, asynchronously preloaded data, and everything else the Java Virtual Machine (JVM) needs to run.

Mathematically, the minimum memory requirements for training a neural network (in bytes) can be estimated as

$$N_{bytes, train} = 4[n_{params}(2 + u) + m(2a + d)]$$

whereas for using the network at test time (after training), we require only

$$N_{bytes, test} = 4[n_{params} + m(a + d)]$$

where:

- u is the size of the updater ($u = 0$ for SGD, $u = 1$ for Momentum, RMSProp, and Adagrad; $u = 2$ for Adam and Adadelta updaters)
- m is the mini-batch size
- a is the size of the network activations for a single example (all layers)
- d is the size of a single example



Parallel Processing and Memory Calculations

When scaling out training to multiple machines (using Spark or `ParallelWrapper`), we need to multiply $N_{bytes, train}$ by the number of model replicas.

A Note About DL4J, Memory, and Precision

By default, DL4J (via ND4J) uses 32-bit floating-point (FP32) numbers for all `INDAR` arrays. This means an `INDArray` with N elements will consume $4N$ bytes. 64-bit floating-point (FP64) numbers can also be used for network training, which can sometimes help with numerical stability issues (though numerical stability issues are typically indicative of other tuning problems, anyway). However, FP64 has both a memory (2x) and performance penalty (up to 32x on consumer graphics processing unit [GPUs], or 2x on Tesla GPUs). CPU performance on FP64 will also suffer to a lesser degree, due to memory and cache effects.

Finally, note also that 16-bit floating-point (FP16) format is also available in DL4J for training on GPUs using CUDA. FP16 format reduces the memory requirements by a factor of two compared to the default FP32 data format, allowing for a larger network, larger batch sizes, or both. However, the FP16 memory savings come at the cost of significantly reduced numerical precision, which can make networks more difficult to tune. If you are new to tuning neural networks or are having trouble tuning your network, stick with the FP32 default, unless you really need it. As for performance: FP16 performance is very poor on most consumer GPUs, hence FP16 is used only as a storage format in DL4J. Actual operations are performed after first converting data to FP32.

If we run into memory issues given our hardware, we have a few options:

1. Reduce the mini-batch size m
2. Get better hardware (or use a cloud service such as Azure, Amazon Web Service, or Google Cloud)
3. Use FP16 instead of FP32 on GPUs (but beware of reduced numerical precision)
4. Use a smaller network

Weight Initialization Strategies

Weight initialization can greatly affect the training process. You should take care in matching the appropriate weight initialization strategy with the appropriate problem context. Initializing weights is a key starting place in the learning process for neural networks and deep networks.

In general terms, we use the following strategy as a baseline in weight initialization:

- Biases can generally be initialized to zero
- Hidden weights need to be initialized such that they break symmetry between hidden units

We break the symmetry between hidden units by adding randomization to the weight initialization process. Weight initialization should be a function of the number of inputs (and maybe also outputs).



Use `WeightInit.XAVIER` weight initialization, or use `WeightInit.RELU` when using ReLU or leaky ReLU activation functions.

If weights are too large, it can mean large outputs and large gradients, which can obviously have a detrimental effect during learning. We also need to deal with initializing the biases of the network units, as well. The early stages of learning can be affected by how we initialize the bias of the visible units. There are different ways to initialize the network biases, but we typically default to 0. Initial hidden biases of 0 are usually fine outside of scenarios in which we are targeting a specific sparsity probability.

Two specific situations we'll call out in this section are weights connected to tanh and ReLU units, respectively. Typically, they are set randomly, by sampling from a (typically Gaussian or uniform) distribution with some variance. In a lot of cases, we can just use the following guidelines and move on:

- For ReLU family activation functions (ReLU, Leaky ReLU, Randomized-Leaky ReLU, etc.), use `WeightInit.RELU` weight initialization in DL4J.

This is also known as He initialization in some contexts, based on the 2015 paper by He et al., “Delving Deep into Rectifiers.”

- For most other activation functions (tanh, identity, etc.) use `WeightInit.XAVIER`.

This is also known as Glorot initialization, based on the 2010 paper by Glorot and Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks.”

For weights connected to tanh units, other initialization methods involve concepts such as Sparse Initialization.⁶

⁶ Martens. 2010. “Deep learning via Hessian-free optimization.”

- For biases, these are typically initialized to 0 (this is the default in DL4J).

In a nutshell, the ReLU and Xavier initializations are designed based on the network architecture (layer sizes) and assumptions about the activation function to do two things:

- Ensure the variance of network activations are constant for all layers. For example, activations don't grow too large or too small in later layers in the network.
- Ensure that the variance of activation (and hence parameter) gradients are constant for all layers. For example, gradients don't grow too large or small by the time they are backpropagated to the earlier network layers.



Weight Initialization and Serving Two Masters

ReLU/Xavier initializations can't optimize those two constraints (activation/gradient variances) exactly in networks with different layer sizes (or if activation function assumptions don't hold).⁷

Orthogonal Weight Initialization of RNNs

In [Chapter 7](#) we look at tuning strategies for specific architectures such as LSTMs. It's worth noting here that LSTMs and Gated Recurrent Units (GRUs) have been shown to have better performance in some cases with orthogonal weight initialization.⁸

Using Activation Functions

[Table 6-3](#) presents notes for matching data distribution types to some of the activation functions in feed-forward networks.

Table 6-3. Target distributions versus output layer activation functions in feed-forward networks

Target distribution	Output layer activation function
Binary (0, 1)	Sigmoid Activation Function
Categorical Targets (1 of C coding)	Softmax Activation Function
Continuous-valued (bounded range)	Sigmoid or tanh (with scaled range of outputs to range of targets)

⁷ Glorot and Bengio. 2010. “[Understanding the difficulty of training deep feedforward neural networks](#).”

⁸ Saxe, McClelland, and Ganguli. 2013. “[Exact solutions to the nonlinear dynamics of learning in deep linear neural networks](#).”

Target distribution	Output layer activation function
Positive (no known upper bound)	ReLU family activation function, softplus activation function (or use logarithmic normalization to transform to unbounded continuous)
Continuous-valued (not bounded)	Linear activation function (equates to no activation function)

Sigmoid functions are the most common in neural network literature. The sigmoidal class of functions were popular 20 years ago in neural network practice; today, they have largely fallen out of favor for use in hidden layers.



Sigmoid and Information Loss

Compared to ReLU activation functions, sigmoid functions more often discard information due to saturation in both forward- and backpropagation, giving nonlinear effects to the network due to a single parameter in a small neighborhood.⁹

Sigmoid has issues with zero gradient with large magnitude input. This becomes less of a problem with mini-batch, but we need to be mindful to carefully initialize the weights to avoid saturation.



The Decline of the Sigmoid Activation Function

We prefer the leaky ReLU activation function in place of the sigmoid activation function these days. The leaky ReLU activation function has neither the vanishing gradient problem of sigmoid/tanh, nor the “dying ReLU” problem of the vanilla ReLU.

The best practice is to not use sigmoid for hidden layers.

ReLU activation functions (piecewise linear units) are the most popular type of hidden unit we see in modern deep networks today. We see ReLUs used commonly in CNNs as the activation function of convolution layers. Stochastic Gradient Descent (SGD) has been found to speed up learning with ReLUs when compared to sigmoid and tanh functions. ReLUs are also computationally cheaper than sigmoid and tanh functions.

⁹ <http://www.deeplearningbook.org/>



The Dying ReLU Problem

You must be careful when using ReLUs because some units might never activate (sometimes referred to as the “dying ReLU problem”) across the entire training dataset.

Networks can still learn with many “dead” ReLU units; however, these units won’t contribute to the network output, resulting in wasted computational effort and lower effective network capacity.

The best practice is to use the leaky ReLU variation, which (unlike the vanilla ReLU) has nonzero gradient for all input values.

The hard tanh activation function has a similar zero-gradient issue to the ReLU activation function, in that parts of the input range have zero gradient: specifically, for inputs below -1 and above $+1$, the gradient is zero.

A Note About Maxout

A maxout model is a feed-forward network that uses a maxout unit. Maxout units are considered a generalization of ReLUs but without the dying unit issue. Maxout units also double the number of parameters per neuron, increasing the overall parameter count.

Maxout units have been shown to learn in conditions in which ReLUs do not perform as well. The difference between ReLUs and maxout units is that maxout units are piecewise linear, where each piece of the linear function has its own weight vector. This aspect of maxout units helps them to avoid becoming stuck in learning in places where ReLUs tend to do so.

Summary Table for Activation Functions

Table 6-4 summarizes the activation functions discussed in this section.

Table 6-4. List of common activation functions and where they are used

Function name	Where to use it
Linear	Output layer for regression
Sigmoid	<ul style="list-style-type: none">• Output layer for binary classification• Outputs values on the range $[0,1]$• Fallen out of favor, not to be used in hidden layers
Tanh	<ul style="list-style-type: none">• Continuous data, more than $[-1,1]$• LSTM Layers
Softmax	Multiclass classification model output layer

Function name	Where to use it
ReLU	<ul style="list-style-type: none">• RBMs• CNN layers• Multilayer perceptron network layers

In general, we recommend exploring ReLU activation functions for general use, but be careful with the learning rate and watch for dead neurons in the network. To combat these effects you could try the leaky ReLU activation function or the maxout activation function. Tanh activation functions are another alternative but have been noted to perform worse than ReLU and maxout activation functions in practice. Sigmoid units are generally used only for single-label classification on the output layer and have fallen out of favor for use in hidden layers.

Applying Loss Functions

Loss functions let the optimization function know how well it's doing at the intended task. Previously in the chapter, we referenced the use of loss functions in the context of the model goal (e.g., output layer loss functions). Following are a few notes about other scenarios for loss functions, where to use them most effectively, and any caveats about how they are used. To be direct, loss functions are used only in two places:

- Output layers (or, `LossLayer`)
- Layers that support unsupervised layer-wise pretraining; for example, autoencoders, RBMs, and VAEs

Most layers (convolutional, dense, LSTM, etc.) do not (and cannot) have a loss function because they can't be pretrained in a layer-wise manner.

Beyond the classification loss functions we listed earlier in the chapter, other classification loss functions include hinge loss and logistic loss. Hinge loss is the most commonly used loss function when the network must be optimized for a hard classification; for example, 0 = no fraud and 1 = fraud, which by convention is called a 0-1 classifier. Logistic loss functions are used when probabilities are of greater interest than hard classifications. Great examples of these would be flagging potential fraud, with a human-in-the-loop solution or predicting the probability of someone clicking on an ad, which can then be linked to a dollar amount. Predicting valid probabilities means generating numbers between 0 and 1.



Binary Classification and Hinge Loss

We use the hinge loss mostly for binary classifications. There are extensions for multiclass classification (e.g., one versus all or one versus one) for the hinge loss that are not covered here.

Understanding the Debug Output During Training

During training, we'll see command-line output such as the following:

```
21:36:00.358 [main] INFO o.d.o.l.ScoreIterationListener
- Score at iteration 0
is 0.5154157920151949
```

The value at the end is the average of the loss function for each example in the current mini-batch. This number will not always begin and end at the same levels and is dependent on the loss function used in the network architecture. If we take MSE, for example, we'd get a different progress score as opposed to, say, a negative log likelihood loss function.

Here's the easiest way to understand this number: "low is good, high is bad," and we want to see it generally drop over time.

To turn on this debugging output, add the following line to an example:

```
myNetwork.setListeners(new ScoreIterationListener(1));
```

Table 6-5 presents a summary of where to use each loss function.

Table 6-5. A summary of loss functions and where to use them

Loss function	Where to use it	Properties
Reconstruction entropy	RBM ^s , autoencoders	Used for feature engineering
Squared loss	Output layer	Regression
Cross-entropy	Output layer	Binary classification
Multiclass cross-entropy	Output layer	Classification
Root MSE	Autoencoder, RBMs, output layer	Feature engineering, regression
Hinge loss	Output layer	Classification
Negative log likelihood	Output layer	Classification



Cross-Entropy, Logistic Loss, and Negative Log Likelihood

You might see these loss functions used in a similar manner across literature and examples. As noted in [Chapter 2](#), cross-entropy has its origin in information theory, whereas negative log likelihood for classification has its origin in statistical modeling. These methods are mathematically the same, so although it might not matter which is used, it tends to confuse people.

Understanding Learning Rates

The learning rate hyperparameter in neural networks is one of the most important (if not *the* most important) hyperparameter to set when tuning a neural network. It has a strong impact on both the stability and efficiency of training times.

[Figure 6-1](#) shows how if the rate is too large (left), our network training will likely be unstable (or diverge entirely). If our learning rate is too small (right), training can take orders of magnitude longer than it needs to.

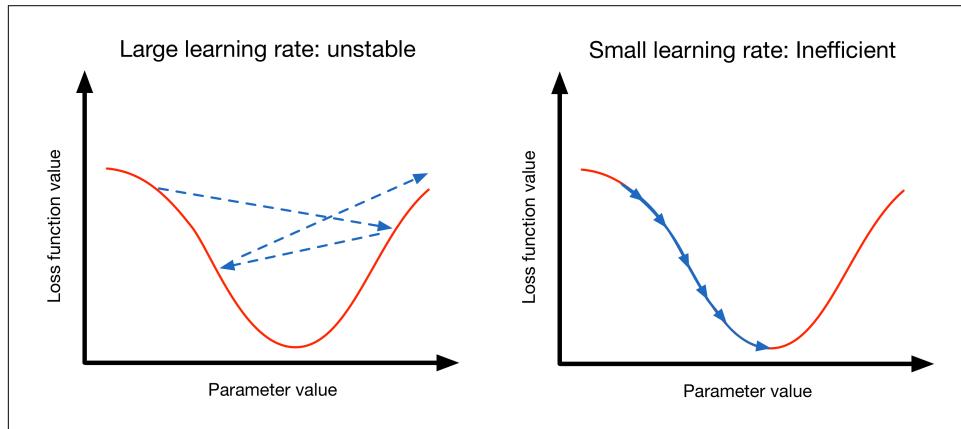


Figure 6-1. SGD and learning rate

We ideally want to see a large learning rate in the beginning of training and then see it fall over time as we approach convergence of the training process. If we are using Momentum, it can be advantageous to slowly increase the Momentum value (described in the next section) as we train as the learning rate falls.



Momentum and Certain Updaters

If you're using an updater such as Adam or RMSProp, the Momentum schedules are not used.



Increasing the Learning Rate Setting

Increasing the learning rate initial setting is not always helpful because although we may initially learn well, our algorithm will learn more slowly in the long term.

Ideally we want to use some type of nonstatic learning rate scheme in which learning rates can be set on multiple tiers:

- Globally
- Per layer
- Per neuron
- Per parameter (where we have more parameters per neuron, bias, etc.)¹⁰

We want the learning rate to begin to drop only near the end of learning. We want a learning rate schedule that decays over time as if we could take the extra computational time we want to train longer with a slower decay rate. We want this decay rate to adapt per parameter and not be too aggressive to hinder learning too quickly.

Using the Ratio of Updates-to-Parameters

A simple and effective approach for setting the learning rate is to use the ratio of updates-to-parameters (or specifically, the mean magnitudes thereof). Recall that the updates are what we have after we apply our updater (RMSProp, Momentum, and so on, and learning rate) to the gradients; that is, our learning equation becomes $\theta \leftarrow \theta - u$ with update u . If we let our updates vector (of length N , same length as our parameters vector) be \mathbf{u} and parameter vector be θ , the ratio of parameters-to-updates is given as follows:

$$\text{update: parameterratio} = \frac{\frac{1}{N} \sum_{n=1}^N |u_n|}{\frac{1}{N} \sum_{n=1}^N |\theta_n|}$$

This ratio measures how much we are changing the parameters relative to the current value of the parameters. We can find this ratio in two places in DL4J's UI: on the Overview page (lower left) and also on the Model page (first graph after selecting a layer vertex). Note that the ratio is presented in DL4J's tuning UI as the logarithm to the base 10, as seen in [Figure 6-2](#).

¹⁰ Bengio. 2012. "Practical Recommendations for Gradient-Based Training of Deep Architectures," in Muller et al. 2012. *Neural Networks: Tricks of the Trade, Second Edition*.

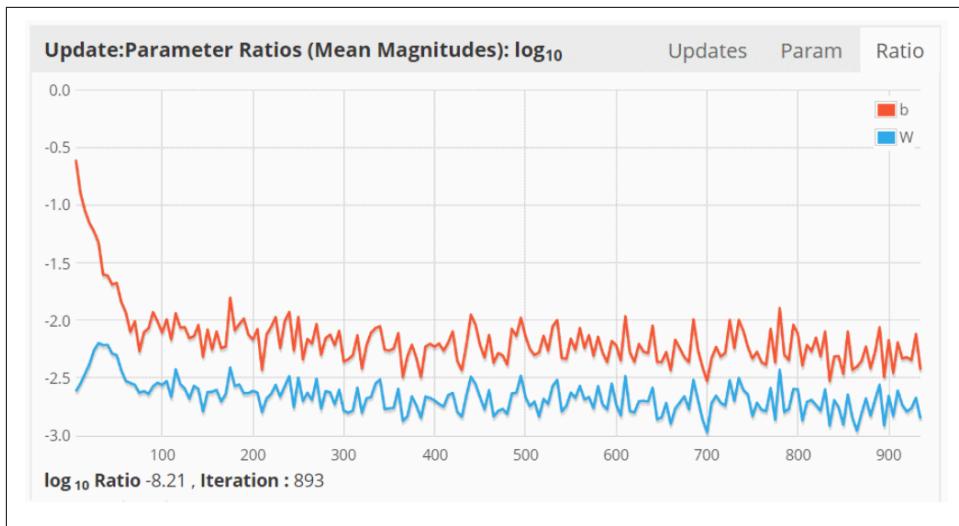


Figure 6-2. DL4J’s tuning UI showing the Update:Parameter Ratios graph

When using DL4J, to view the training progress shown in Figure 6-2, after the network starts training, go to <http://localhost:9000/>.



Using the Ratio to Set the Learning Rate

To set our learning rate, we want an update:parameter ratio of approximately 0.001, which corresponds to a value of \log_{10} ratio = -3 in DL4J’s training UI.

For example, a log ratio of -4 (i.e., 0.0001) would mean we should increase our learning rate; a log ratio of -2 (0.01) means we should decrease our learning rate.¹¹

Specific Recommendations for Learning Rates

Learning rate settings tend to begin with a larger setting and fall over time through a decay mechanism. If the learning rate starts out too large, we generally see the average training loss to increase. The best learning rate is generally close (by a 2x factor) to the largest learning rate that does not cause the learning process to diverge.

¹¹ You also could use other measures, such the ratio of L2 norms, for this purpose; these aren’t available in DL4J’s training UI, but you could calculate them manually.



Good Initial Learning Rates

Ideally, we begin with a large learning rate, and if the learning process diverges, we divide this rate by a factor and try again until the learning process begins to converge.

We suggest starting out with the following learning rates and then see which is the most stable: [0.1, 0.01, 0.001], with 0.001 being one of the most popular places to begin.

We also suggest trying learning rate methods in the following order:

1. Adam

- Currently recommended as the default method to use

2. Nesterov Momentum

3. RMSProp

- Uses moving average of squared gradients
- Doesn't have issue when learning rates become monotonically smaller, as in AdaGrad

4. AdaGrad

Other supported options in DL4J include:

- AdaDelta



We Still Need to Watch Learning Rates

If we set the learning rate too high, training with AdaGrad will still be unstable (as in any other method). AdaGrad can't increase the learning rate. In fact, the effective (per-parameter) learning rate with AdaGrad is **monotonically decreasing**.

AdaGrad does act like a per-weight learning rate decay mechanism. However, a downside of AdaGrad is that the monotonic learning rate usually proves too aggressive and stops learning too early.

Implementations such as DL4J have AdaGrad builtin, so users can worry less about hand-tuning the learning rate.



Working with Momentum/Nesterovs in DL4J

Momentum helps the learning process escape local minima and find better solutions to the optimization process.¹²

We typically see Momentum values around 0.9. Other common values near this setting that are commonly used are: [0.5, 0.9, 0.95, 0.99]. A typical initial setting in Momentum would be 0.5 and then anneal it to 0.9 or so over multiple epochs. It's also critical to have `.updater(Updater.NESTEROVS)` in addition to `.momentum(0.5)`. Without these two settings momentum will not be enabled.

When a hidden unit has a very large fan-in (where many connections are coming in), the updates need to be smaller, given that many small changes in the same direction can have unintended effects on the sign of the gradient. In this context, the learning rate should be smaller. For biases, we see updates that can be bigger yet no adverse effects are seen. These mechanisms can make tuning the learning rate a challenge.

Here are a few other things to keep in mind:

- Learning rates are network specific: what works in one network might not work in another.
- Keep in mind that the “update:parameter” heuristic is a guide only; it doesn’t always work, but it’s frequently a good place to start.
- The required learning rate depends (sometimes a lot) on other hyperparameters. For example, different updaters (Momentum, RMSProp, AdaGrad, etc.) can require different learning rates; changing the weight initialization or number/type/size of layers can also require the learning rate to be retuned. Thus, we should always check our learning rate after making significant changes to our network.
- Give the network some time to settle down before reading the update ratio chart —perhaps a few dozen to a few hundred iterations (parameter updates) after training has begun. The first few updates after starting training are often larger.
- It’s normal—and expected—for the bias update ratio to begin much larger than 0.001; after all, biases are usually initialized to 0.0. Thus, the initial update ratios will be large, and will reduce in magnitude over time.
- In some cases, it can be necessary to set different learning rates for different layers in order to get the correct ratio. However, this can be indicative of other problems, such as vanishing/exploding gradients, poor weight initialization, or problems with data normalization. Adam and RMSProp updaters may help to

¹² Sutskever. 2013. “Training Recurrent Neural Networks” (PhD Thesis).

mitigate this problem, as they can have a normalizing/scaling effect on the gradients.

How Sparsity Affects Learning

The sparsity hyperparameter is commonly used in deep learning. Sparsity encourages hidden units to be sparse (or near zero). Sparse representations have been thought to be good because they encourage representations that separate the underlying factors of representation.¹³

Sparsity helps training to become unstuck when weights get large values and have issues backing out of them. Sparsity sometimes interacts with other hyperparameters that dictate the size of the network. As the number of hidden nodes increases, our sparsity factor should increase. How sparsity interplays with network size also will be affected by the type of activation function we're using for the network architecture.

To understand how well the sparsity setting is working, we take a look at the histogram of the mean activities of the hidden units. From that we can set the sparsity so that the hidden units have mean probabilities near that of the target. If the probabilities are tightly clustered around the target value, we need to lessen the sparsity setting. This way it will interfere less with the main goal of the learning process.



Sparsity and Certain Network Architectures

Sparsity is not available for DenseLayers and LSTM layers in DL4J.



Sparsity Target Values

For quality learning we want to set the sparsity target between 0.01 and 0.1^{-9} .

Applying Methods of Optimization

The most prevalent methodology for optimizing deep learning networks is SGD; it is relatively easy to implement but is challenging to tune and parallelize.

Some researchers have made the argument that adding additional parameters to neural networks fails to reduce underfitting on large datasets and wastes capacity. These

¹³ Le, Jaitly, and Hinton. 2015. “A Simple Way to Initialize Recurrent Networks of Rectified Linear Units.”

researchers have pointed to SGD as the cause and suggested using second-order methods to mitigate these issues, including:

- Broyden-Fletcher-Goldfarb-Shanno (BFG)—Limited-memory BFGS (L-BFGS)
- Conjugate Gradient (CG)
- Hessian-free

Table 6-6 provides a quick reference table on best practices for initial optimization method for each network architecture.

Table 6-6. Optimization algorithms for different architectures

Network	Common training methods and notes
DBNs	SGD
CNNs	SGD (+Dropout)
Recurrent Neural Networks	SGD, Hessian-free

Note of the effect of training dataset size on optimization method choice. When dealing with smaller datasets, we suggest using the second-order optimization methods and setting the batch size to the full dataset. When dealing with larger datasets, we suggest using SGD with a mini-batch setting. You could also try a second-order method here with a mini-batch setting lower than the full dataset size.

Table 6-7 presents a comparison of each method.

Table 6-7. Comparison of optimization methods

Method	First-order info	Second-order info	Pros	Cons
SGD	Gradient	None	Fast to converge, lowest cost per parameter update	Not as robust
L-BFGS	Gradient	Curvature information is estimated from the gradients	Able to find better local minima	Can train better than SGD, higher cost per parameter update, higher memory costs
CG	Gradient	Curvature information is estimated from the gradients	—	Higher cost per parameter update, higher memory costs
SGD Hessian-free	None	Curvature	Automatically derives step size, uses conjugate gradient to find next step	Not generalizable to all architectures, higher cost per parameter update, higher memory costs

Although we offer **Table 6-7** as a quick comparison, for the most part you should focus on starting out with SGD; thus, in the following section, we dive into specific best practices for it.

SGD Best Practices

SGD works well with Momentum and a well-designed random weight initialization scheme. It has been shown to be effective at training both DBNs and Recurrent Neural Networks¹⁴ to levels on par with Hessian-free optimization. Researchers have seen that a carefully tuned SGD method is hard to beat on large classification problems.¹⁵

Following is a quick summary of the best ways to begin training with SGD in deep learning:

- Use a good shuffle method on the input training set¹⁶
- Watch both the error percent over iterations and the validation error rate
 - We want to see training error decrease
 - We can stop training early if we see validation error flatten out
- Use smaller subsets of the training data to try out variations of hyperparameters (specifically for tuning the learning rate)
- Combine with Momentum, AdaGrad, and RMSProp
- Set learning rate appropriately: too large or small is bad
- Normalize the input data (not specific to SGD, but worth repeating because it's often forgotten)

For smaller problems with real-valued outputs (function approximation, control problems, etc.), CG offers great performance. With the second-order methods we typically see larger mini-batch sizes (10,000) than with the SGD methods.



SGD as the Most Frequent Starting Place

In practice people seem to use SGD + (Momentum or AdaGrad or RMSProp, etc.) more frequently than second-order methods.

Using Parallelization and GPUs for Faster Training

As our machine learning application goals grow more ambitious, we need larger models to accommodate them. This means that we need to train more parameters, which takes more training time. We also want to train on larger datasets to get a more complete view of the model, where we see I/O overhead begin to dominate after a point (see also Jeff Dean's 12 Numbers in [Appendix C](#)). In the context of a single

¹⁴ Sutskever et al. 2013. “On the importance of initialization and momentum in deep learning.”

¹⁵ LeCun et al. 1998. “Efficient BackProp” in Muller et al. 2012. *Neural Networks: Tricks of the Trade Second Edition*.

¹⁶ Bottou. 2012. “Stochastic Gradient Descent Tricks” in Muller et al. 2012. *Neural Networks: Tricks of the Trade Second Edition*.

machine, we are at the mercy of the raw clock speeds of modern CPUs. Computer architectures have hit a hardware power limit and new gains are seen by combining the power of multiple CPU cores together in different forms of parallel processing. We can quickly hit the limits of sequential learning in modern machine learning activities.¹⁷ Bandwidth of storage and network per computer has not been able to keep up with the increase in data. We need to begin thinking about data analysis algorithms that are able to perform most steps in a distributed fashion.



Touching Disk at Scale

Accessing a hard drive can be expensive in terms of computational latency. For example, if we have a 2 terabyte hard drive and we can read from the disk at 100 MB/s, it will take around six hours to read the entire contents. For more information on the different latencies of computer architecture take a look at [Appendix C](#) on “Jeff Dean’s 12 Numbers.”

Online Learning and Parallel Iterative Algorithms

The move from batch learning algorithms to online learning algorithms has been able to deal with increasing dataset sizes for more than a decade now. However, whenever we have more than a single drive of data, this task becomes computationally impractical to process all data by SGD due to I/O overhead. In the world of big data, we need to move the data less so we end up looking for methods to parallelize execution of programs.

In recent work traditionally serial learning processes have been redesigned for parallel computations. This is accomplished by having a core perform the local computations and then combining the work from several cores centrally to produce a global result.¹⁸ As our dataset exceeds the capacities of a few drives, we begin to rethink how we design our storage and processing systems. This was the impetus for the design of MapReduce and the Google File System at Google in the early 2000s (and by extension the work done by Doug Cutting and Michael Cafarella on Hadoop in the mid-2000s).

We can parallelize computer programs and more specifically learning algorithms in different ways. Any parallel program is composed of simultaneously executing processes. The different forms of parallelism relate to how we decompose the process into parts that can be run in different places in parallel. Two main ways to do this are *task parallelism* and *data parallelism*.

¹⁷ Zinkevich et al. 2011. “Parallelized Stochastic Gradient Descent.”

¹⁸ Dean and Ghemawat. 2004. “MapReduce: Simplified Data Processing on Large Clusters.”

Task parallelism

With task parallelism (also known as *functional parallelism*), we divide the work to be done into several separate tasks that are then scheduled on different processing units for execution. The processing units could be local threads or a core on a different physical machine.

Data parallelism

With data parallelism we divide the work based on applying the same function to a different parts of a dataset. The tasks to perform the work are then scheduled on different threads (locally or on a remote machine in a distributed cluster), which makes data parallelism a subset of task parallelism. With data parallelism, we spread the computation across multiple processing units in a parallel computation environment. These units can be threads on the same core, threads on different cores, or cores on entirely different physical machines in a cluster.

As we parallelize iterative-class algorithms such as SGD we can use local thread-parallelism effectively with the parameter-averaging technique.¹⁹ As the data size grows, however, we begin to see issues with copying the data to the parallel processing units. The first intuition is to move to a system that is capable of handling big data, such as Hadoop.

Big Data?

It's very common to hear the term "big data" in today's marketing around enterprise software. The term has become so ubiquitous that it has even been mentioned by the President and is frequently referenced in the *Wall Street Journal*. Interestingly enough, most people struggle to define the term even though it's used far and wide. We define big data in the practical sense that the data is processed where it is stored, where we move the computation to the data. MapReduce is architecturally built around this concept. Many companies and tools claim to handle big data, but, simply put, if we move the data before we process it, we aren't working with big data.

¹⁹ Agarwal, Chapelle, Dudik, and Langford. 2011. [A Reliable Effective Terascale Linear Learning System.](http://hunch.net/~vw/)”, <http://hunch.net/~vw/>, and McDonald, Hall, and Mann. 2010. “Distributed training strategies for the structured perceptron.”



How Long Does It Take to Scan a Petabyte?

A great example of the constraint of big data is that moving data becomes a nonstarter after a certain point. A great example of this is calculating how long it takes to run a linear scan over a petabyte of data at the rate of a single commodity drive (around 40 MB/s). This basic scan/copy operation will take around 310 days on a single processing unit at 40 MB/s.

Apache Hadoop is a great place to store and manage a large amount of data such as web logs or sensor readings because it has been shown in enterprises around the world to be petabyte-scale. The traditional parallel framework that was originally included in Hadoop distributions²⁰ was the MapReduce framework. MapReduce parallelization is based around using the location of large amounts of data where it is stored. It pushes computation (or tasks) to the local hosts where the data block is stored and executes locally there to maximize the throughput on the physical spindle.

What Is MapReduce?

MapReduce is a traditional batch-class parallelization inspired by the map and reduce functions commonly used in functional programming. It processes key and value pairs from a map task (locally executing on a host that has the data block to reduce I/O and network cost), shuffling the data into partitions for each reducer to operate on in the reduce phase. Implementations such as Hadoop handle machine failures transparent to the programmer, whereas traditionally this was part of the programming experience in parallel programming. However, MapReduce does not fit well with iterative-class algorithms due to issues with the setup time per pass over the dataset.



Issues with Iterative Algorithms and MapReduce

If you wanted to make 100 passes over the dataset and your MapReduce scheduling cost is 30 seconds per dataset pass, this gives you an overhead of (100×30) 3,000 seconds. This is 50 minutes of just scheduling overhead time, which ends up dominating many training processes. This dynamic precludes MapReduce from being a good parallel framework for parallel iterative algorithms.

Parallel optimization methods have attracted increasing attention in recent years as a way to scale-up machine learning algorithms. From MapReduce class parallelism to

²⁰ See Cloudera and Hortonworks

more iterative methods of parallelism, we're seeing many forms of parallel speedup. SGD has been the focus of many research papers²¹ that have parallelized SGD without using the MapReduce framework. We also see the evolution of open source frameworks ([Vowpal Wabbit](#) and [Apache Spark](#)) for the parallelization of iterative methods for which parameter averaging is a central theme in the implementation.

Parallelizing SGD in DL4J

Another publication²² out of Google describes such a parallel system for iterative class algorithms such as parallel SGD. This system is called *Downpour SGD* and takes advantage of certain features such as AdaGrad, large numbers of model replicas, and Sandblaster (parallel L-BFGS). Downpour (and components) was the inspiration for DL4J's parallelization architecture, which is depicted in [Figure 6-3](#).

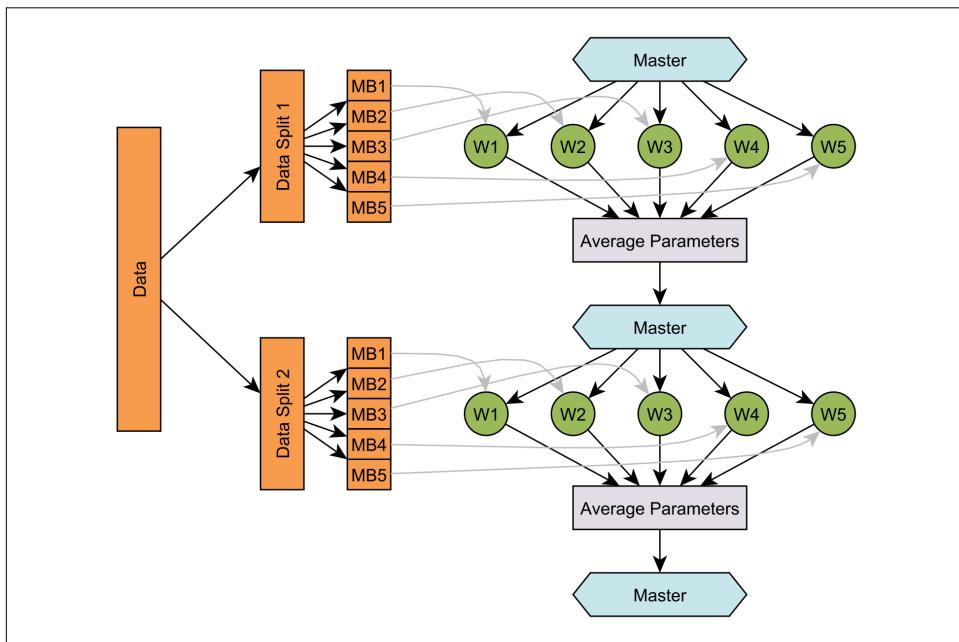


Figure 6-3. DL4J's Parameter averaging parallelization strategy

21 Ngiam et al. 2011. “On optimization methods for deep learning.”

22 Dean et al. 2012. “Large Scale Distributed Deep Networks.”

One of the major research papers that inspired DL4J's parallelization strategy was the work by Jeff Dean and his team at Google on their SandBlaster tool.

Who Is Jeff Dean?

Jeff Dean is a distinguished research engineer at Google where he has worked on some of the core systems. Jeff joined Google in mid-1999 and is a Google senior fellow in the Knowledge Group. As just mentioned, Jeff led the design and implementation of DistBelief and SandBlaster (ICML 2012). DistBelief is Google's large-scale distributed system for training deep neural networks and has been used for image recognition, speech recognition, Natural Language Processing (NLP), and other machine learning tasks. Following are some of the other influential machine learning and distributed systems projects Jeff has led:

- Five generations of crawling, indexing, and query serving systems
- Initial development of Google's AdSense for Content product
- Design and implementation of MapReduce
- Design and implementation of BigTable
- Design and implementation of Spanner

Little known fact:

Jeff Dean writes directly in binary. He then writes the source code as documentation for other developers.

DL4J supports many possible runtimes (e.g., AWS, threaded, Spark-YARN on Hadoop) but all of them use the parameter averaging strategy (as of today). This technique is common in parallel iterative research literature, as referenced earlier, and we designed around a common set of general patterns that was ported to each runtime.



Parameter Server

A parameter server design is currently being added to the project, but for now, parameter averaging is a worthy approach to many parallel deep learning modeling problems.

Parallel SGD execution

A high-level execution flow of this pattern is as follows. Each worker is given a split of the total dataset and runs its own training algorithm over the mini-batches on just the records in the split. Per mini-batch, the worker sends its parameter vector back to the master node to perform parameter averaging at the *super step* stage of the processing flow.



Controlling Parameter Averaging

Parameter averaging frequency is configurable in DL4J. We consider parameter averaging “per mini-batch” to be inefficient. A good setting has shown to be “per 5–20 mini-batches” in practice.

The master node then sends this new global parameter vector back to the workers so that they can perform the next mini-batch within their data split. This continues until we have completed a pass over all records in the dataset or we’ve completed N passes over the dataset (refer to [Chapter 9](#) on Spark for more information).

We can illustrate this with an example of training in parallel. Suppose that we have 10 parallel workers each with even splits of 100 records per worker (to make it easy). Each worker will then further subdivide the split into mini-batches of 10 records. The workers (in parallel) will perform the learning process on each mini-batch until they complete their 100 records, or “split,” of the total dataset. Parallelization gives us the luxury of having relatively slow learning at the worker level but allowing the training time to drastically speed up. Experimental results in literature and open source implementations show that the speed improvements are close to linear in the number of machines.

Parallel training might see different types of training effects in practice than serial training.

The two major hyperparameters we focus on when dealing with parallel training are mini-batch size and regularization. We need to consider these two more closely in parallel training because they can behave differently than in serial training.

Minibatching

One example is that in parallel training when we have a mini-batch size that is too large, we see mini-batches that have outliers from the dataset. This situation creates a longer training convergence period because it’s more difficult for the learning process to gain the information from the outliers in larger mini-batches.

Regularization

Another example is that solving problems with more variance (small regularization constant) derives more model value from parallelization. The major difference in training with parameter averaging as opposed to in a serial fashion is reduced I/O overhead and the ability to take advantage of data locality (in implementations that move compute workers to the data block host location). Another side effect is that parameter averaging acts as a regularizing function on the parameter vector.

GPUs

The parameter average method is not the only way to speed up iterative class algorithms. Let's take a look at another method based on using GPUs that can use vectorized math implementations on its specialized hardware. The common graphics card shipped today with a desktop PC has a peak memory bandwidth several times higher than modern CPUs.



Modern GPUs have more than 1,000 **CUDA cores**.

An interesting property of GPUs is that they can work concurrently with thousands of threads and scheduling linear algebra work on cores is possible with little overhead. This class of fine-grained parallelism makes GPUs an attractive candidate for speeding up our large-scale linear algebra computations in machine learning.

We want to perform our memory accesses in a fashion that allows greater parallelism. GPUs give us a variant of this where memory accesses are coalesced, and the hardware can perform them in parallel where the effective access speed is several times faster than between the normal CPU and RAM scenario. This makes the main bottleneck transferring data between the CPU's RAM and the GPU's RAM. To give a relative sense in this context, we see that in most cases this transfer time dominates even large $\text{matrix} \times \text{matrix}$ computations on the GPU (e.g., matrix operation taking only 0.5 percent of the time in a $1,000 \times 1,000$ -matrix multiplication). We can achieve more efficiency from our hardware and GPUs if we transfer more data at one time into RAM and perform multiple operations in a larger batch.

As we can see in [Appendix C](#), we want to batch memory operations when we touch hardware. After we have an efficient transfer of data into the GPU hardware, we can then take advantage of the thousands of extra threads operating on separate blocks of data within the training batch sent to the GPU at one time. These factors are the major drivers for the linear algebra architecture of DL4J in how we use batching to accelerate computation of many parameters across large neural networks. To further accelerate these computations we created ND4J to make the vectorized computation easily portable between linear algebra runtimes on different systems and different GPUs.



ND4J: An Adaptable Execution Engine

DL4J was built from the beginning with the idea that we'd need to operate on CPUs, GPUs, or on a parallel framework. You will find considerable flexibility working with swappable ND4J backends, giving you options on execution.

To learn more about setting up GPUs as a ND4J backend, see [Appendix E](#).

Controlling Epochs and Mini-Batch Size

In [Chapter 2](#), we introduced the concept of the mini-batch. It's been shown that dividing the training input dataset into mini-batches allows us to more efficiently train the network. A mini-batch tends to be anywhere from 10 input vectors up to the full input dataset.

This method also allows us to compute certain linear algebra operations (specifically matrix–matrix multiplications) in a vectorized fashion. In this scenario, we also have the option of sending the vectorized computations to GPUs if they are present.

Here are a couple of key terms to understand with respect to how we control training:

Epoch

An epoch is a full pass over the entire input dataset. Many times we train on multiple epochs of a dataset before finding training convergence.

Mini-batch size

Mini-batch size is the number of records (or vectors) we pass into our learning algorithm at the same time. This contrasts with where we'd pass in a single input record on which to train.

The relationship between how fast our algorithm can learn the model is typically U-shaped (batch size versus training speed). This means that initially as the batch size becomes larger, the training time will decrease. Eventually, we'll see the training time begin to increase when we exceed a certain batch size that is too large.



As mini-batch size increases, more computation means gradients might be smoother but are more costly to compute.

Ideally, each mini-batch trained on should contain an example of each class to reduce the sampling error when estimating the gradient for the entire training set.

Understanding Mini-Batch Size Trade-Offs

In principle (with appropriate tuning), a neural network can learn with any mini-batch size. In practice, we need to choose a mini-batch size that balances the following:

- Memory requirements
- Computational efficiency
- Optimization efficiency

Memory requirements have been covered in an earlier section, so we won't talk about them here.

Regarding *computational efficiency*, DL4J (and other modern deep learning libraries) parallelize learning at the level of mathematical operations such as matrix multiplications and vector operations (additions, element-wise multiplications, etc.). This means that too small a mini-batch size results in poor hardware utilization (especially on GPUs), and too large a mini-batch size can be inefficient—again, we average gradients over all examples in the mini-batch, which means eventually the computational cost of adding more gradients outweighs the benefit.

For *performance* (this is most important in the case of GPUs), we should use a **multiple of 32 for the batch size**, or multiples of 16, 8, 4, or 2 if multiples of 32 can't be used (or result in too large a change given other requirements). In short, the reason for this is simple: memory access and hardware design is better optimized for operating on arrays with dimensions that are powers of two, compared to other sizes.



We should also consider powers of two when setting our layer sizes. For example, we should use a layer size of 128 over size 125, or size 256 over size 250, and so on.

With regard to *optimization efficiency*, it's sufficient to note that we cannot choose the mini-batch size totally in isolation from the other hyperparameters such as learning rate—a larger mini-batch size means smoother gradients (i.e., more accurate/consistent gradients), which, in conjunction with appropriate tuning, allow for faster learning for a given number of parameter updates. The trade-off of course is that each parameter update will take longer to compute. Using a larger mini-batch size might help our network to learn in some difficult cases, such as for noisy or imbalanced datasets.

So, what mini-batch size should we choose? In practice, 32 to 256 is common for CPU training, and 32 to 1,024 is common for GPU training. Usually, something in this range is good enough for smaller networks, though you should probably test this for larger ones (where training time can be prohibitive).

However, memory requirements can limit the maximum batch size for large networks anyway.



For distributed training, it's not uncommon to use smaller mini-batch sizes per executor on shared hardware (such as when training on Spark with multiple executors per machine).

Finally, don't forget that the number of parameter updates per epoch is reduced when we increase the mini-batch size (where an epoch refers to one full pass of the training data). The number of parameter updates per epoch is just the total number of examples in our training set divided by the mini-batch size.



The Relationship Between Mini-Batch Size and Epochs

If we increase our mini-batch size by a factor of two, we need to increase the number of epochs by a factor of two in order to maintain the same number of parameter updates.

How to Use Regularization

Some types of regularization help drive down weights in the parameter vectors that become too large too fast (e.g., L1 and L2). In some contexts or machine learning textbooks, weight decay is referred to as regularization. Other types of regularization, such as dropout and dropconnect, use other methods to mitigate overfitting besides reducing large weights. In general, we want to use regularization methods in our training because it does more than just prevent overfitting.

Regularization can give us a more efficient representation of the model by using different methods to prevent our parameter values from becoming too large. Finding the right combination of settings for regularization is typically done via manual tuning but also can be accomplished with “random search” or “grid search.” In the section that follows, we'll give a short summary of each regularization method and how it affects the overall training process.

Priors as Regularizers

Applying a prior function is a common machine learning technique for regularizing the parameter vector. Weight decay is typically performed through a regularizing function such as the L1 or L2 prior functions. In some cases, these functions are com-

bined,²³ which is common for deep networks. **Table 6-8** shows how we use a different prior function depending on what type of input data we're modeling.

Table 6-8. Summary of where to use prior functions

Prior function name	How it's used
L1	Sparse models
L2	Dense models

In the literature we see both L1 and L2 regularizers used at the same time with separate settings for each function. For cases in which we're using early stopping, we might not want to use L2 regularization at all, because early stopping is more efficient at performing the same mechanics as L2.²⁴ L1 regularization has been consistently shown to act as a useful form of feature selection.

In practice, L2 regularization is more frequently used. Here are a couple of ways in which L1 and L2 compare and contrast with each other:

- L2 more heavily penalizes large weights, but doesn't drive small weights to 0.
- L1 has less of a penalty for large weights, but leads to many weights being driven to 0 (or very close to 0), meaning that the resultant weight vector can be sparse.

We can combine L1 and L2 in the one network.



In practice we see L2 regularization give better performance over L1 outside of explicit feature selection.²⁵

As you'll see in the next section, how much data we're using also can have effects on the results of the modeling process.

Max-Norm Regularization

In this form of regularization, we set an upper bound on the L2 norm of the incoming weight vector for each individual hidden unit. This is in contrast to penalizing the squared length of the entire weight vector, as we'd normally do with the L2 norm.

²³ Zou and Hastie. 2005. “Elastic Net.”

²⁴ Bengio. 2012. “Practical Recommendations for Gradient-Based Training of Deep Architectures,” in Muller et al. 2012. *Neural Networks: Tricks of the Trade Second Edition*.

²⁵ Li, Karpathy. “CS231n: Convolutional Neural Networks for Visual Recognition” (Course Notes).

Using a method based on constraints rather than the penalty strategy keeps weights from growing too large regardless of how big the incoming weight update is.

Max-norm regularization allows us to start with large learning rates coupled with a learning rate decay strategy (e.g., AdaGrad) giving us the ability to more completely search the weight-space compared to other methods with smaller learning rates. It has been shown to work well with SGD in deep neural networks even without drop-out.

Dropout

Dropout is a powerful method of regularization that we can use across many types of models. Dropout is a computationally inexpensive way of regularization during model training by removing units from the network. Dropout works with nearly every type of neural network architecture and has been shown to work well with SGD. Dropout works by setting a unit's activations to 0.0 temporarily. For input layer neurons, dropout is done with a probability (e.g., probability of retaining, or removing an activation) between 0.5 and 1.0.



Sometimes we'll use no dropout at all on the input, especially for noisy or sparse datasets.

In hidden layers, dropout is done with a probability of 0.5. By randomly omitting neurons we can prevent coadaptation among detectors, which helps drive better generalization in models on held-out data.



Output Layers and Dropout

It is not common to use dropout on the output layer.

In general, almost all settings for dropout help, short of extreme settings. A setting of 0.5 for dropout is common and has been seen to work well on a range of networks and goals. Dropout in all hidden layers works more effectively than in only one hidden layer. Dropout also tends to make the activations of hidden units sparse even when other regularization techniques are absent, leading to sparse representations.²⁶

²⁶ Srivastava et al. 2014. “[Dropout: A Simple Way to Prevent Neural Networks from Overfitting](#).”

We can apply dropout (with probability 0.5) in DL4J by adding the following to our network configuration:

```
.regularization(true).dropout(0.5).
```



The Dropout Probability

In DL4J, the dropout probability (set by using the `dropout(double)` configuration option) is the probability of retaining an activation. That is, `dropout(0.7)` means that an activation is retained 70 percent of the time, and set to zero 30 percent of the time. Note that in DL4J `dropout(0.0)` is used to mean “disable dropout for this layer” (as 0 percent probability of retaining an activation doesn’t make sense).

According to the original dropout paper by Srivastava et al., the effect of dropout doesn’t depend strongly on the dropout probability. In most cases, just set it to 0.5 and move on. However, if you have a very large network (relative to the amount of training data), you might want to use a lower probability of retaining an activation (which corresponds to stronger regularization).

More notes on using dropout:

- Dropout is often combined with other regularization methods, such as L2 regularization, in practice.
- As a general rule, you should avoid using dropout on the first layer in your network (disable it by setting `dropout(0.0)` on that layer): this will avoid wiping out important parts of your input dataset.



DL4J, Dropout, and Recurrent Neural Networks

For Recurrent Neural Networks, dropout is generally applied only on the activations into the layer and not the recurrent activations within the layer. Applying dropout on the recurrent activations can make it too difficult to learn temporal dependencies. DL4J applies dropout on the input activations only, when applied to a Recurrent Neural Network.

Issues with dropout

Dropout has been seen in research to be less effective as the number of training records rises up into the tens of millions. Dropout has also been shown to increase training time by a factor of two to three times than the same network architecture with no dropout enabled. Noise in the gradients for learning is higher with dropout than standard SGD.

A way to combat the gradient noise is to use a higher Momentum setting (up from 0.9 to 0.95 to 0.99). In some cases, this will cause the weights to become large, which we manage through the use of max-norm regularization.²⁷



Dropconnect

Dropconnect simply sets some weights (temporarily) to zero. Dropconnect is related to dropout, but isn't really a variant of dropout—they operate on different things.

Other Regularization Topics

Here are a few other items to keep in mind about regularization:

*Stochastic pooling*²⁸

This is a regularization technique for CNNs (specifically) by which we use a form of randomized pooling for building ensembles of CNNs. Each network attends to a different spatial location of each feature map.

*Adversarial training*²⁹

Inputs that you can create by using small but intentionally badly formed variations of the training dataset. This will generate a network model that gives the wrong answer with high confidence. In practice this has been shown to reduce overfitting and result in models resistant to adversarial examples.

*Curriculum learning*³⁰

This variation of training involves beginning with easier-to-learn examples from the training set for training mini-batches earlier and then progressively training on more difficult examples over time. This strategy results in faster training and convergence to better solutions while acting as a regularizer for the network, as well. Results have shown mixed success.

²⁷ *Ibid.*

²⁸ Zeiler and Fergus. 2013. “[Stochastic Pooling for Regularization of Deep Convolutional Neural Networks](#).”

²⁹ Goodfellow, Shlens, and Szegedy. 2014. “[Explaining and Harnessing Adversarial Examples](#).”

³⁰ Bengio et al. 2009. “[Curriculum Learning](#).”



Parallelization and Regularization

You should prefer to remove noise from the weights in the parameter vectors during learning due in part to the use of learning rates. One way to remove some noise from the final weights is to average the weights across several updates. You also can accomplish this through the parameter averaging effect in the parallel runtime modes described later in the chapter.

In some cases of parallelization, we'd use no prior function as a regularizer because parameter averaging achieves the intended effect itself.

Working with Class Imbalance

Often, we work with training datasets in machine learning for which the classes are significantly unbalanced with respect to the number of each type of class. This typically occurs in domains in which the event we're most interested in predicting occurs far less often than when the event does not occur. Examples of this would be predicting whether someone will click an ad on a web page.



PhysioNet and Predicting Intensive Care Unit Death

The [PhysioNet Challenge](#) dataset mentioned in [Chapter 1](#) is a great example of a challenging unbalanced dataset. Getting a lot of data on a (hopefully) rare event such as Intensive Care Unit death can be a challenge.

For the majority of learning methods, if we train the model on data that is 99 percent negative and 1 percent positive the model will just learn to always predict the dominant class (e.g., negative). We might not initially think anything is wrong because the average correct score will appear exceptionally high and we might think the model is just highly accurate. We end up in this situation because we're minimizing error in our learning process and we can achieve 99 percent correct by always predicting positive.

A good practice for building models in imbalanced data situations is to not optimize for percent predictions correct, strictly; instead, we should focus on model evaluations such as F1 score or Area Under the Curve (AUC). We need our model to make some positive predictions but hopefully when it's the most appropriate.

A Word on Model Calibration

For the domains in which we see a large class imbalance, we often see unequal consequences for incorrectly predicting certain classes. An example of this would be not detecting a serious disease when a patient actually has the disease (false negative) and the patient consequently dying. In this example, the consequences of not detecting a fatal disease are much greater than predicting falsely that the patient has the disease. From this perspective, better understanding model evaluation is crucial for the real-world application of machine learning.

To deal with the issue of model evaluation, we use a method called *model calibration*. Model calibration is generally based on deriving a transformation that changes the values or probabilities to better estimates. [Table 6-9](#) summarizes these calibration techniques.

Table 6-9. A taxonomy of calibration problems^a

Type	Task	Problem	Global/local	What is calibrated?
CD	Classification	Expected class distribution is different from real class distribution	Global or local	Predictions
CP	Classification	Expected/estimated probability of right guesses is different from real proportion	Local	Probability/confidence
RD	Regression	Expected output is different from real average output	Global or local	Predictions
RP	Regression	Expected/estimated error confidence intervals or probability density functions are too narrow or too broad	Local	Probability/confidence

^a Bella et al. 2009. “Calibration of Machine Learning Models.”

As noted in the original paper, types CD and RD need to modify predictions in order to calibrate the results. Although we don’t further cover these methods in this book, we felt compelled to note the topic for further study.

There are two main ways to deal with class imbalance during the training process:

- Subsampling from the larger class(es) or oversampling from the more infrequent class(es)
- Weighting the training instances based on a weighted loss function

Let’s now take a closer look at each of these.

Methods for Sampling Classes

Our first approach for dealing with class imbalance is to sample the larger class such that we are left with a balanced training dataset (e.g., 50 percent positive, 50 percent negative). A criticism of this method would be that we're throwing away good data but the upside is that we're making training more efficient.

One approach is to use *post scaling*. This is the process by which we train a network normally but scale outputs following the training. The results are less consistent compared to other methods.

Another approach is called *probabilistic sampling*. This is when we select samples by first randomly (e.g., through some probability) selecting a class and then randomly selecting a sample in the class.

The last approach we'll note is *super sampling* where we duplicate the lower-frequency classes to supersample them up to the count of the other classes.³¹

Weighted Loss Functions

An alternative method to dealing with class imbalance during train is to use a lower weight to training examples that are in the larger class. This helps balance out the training impact of unbalanced datasets and tends to create models that have a better ability to detect the more rare labels in generalization.

To help you understand this concept further, we'll illustrate it with a small section of code. In the following code snippet, we can see an `INDArray` being created to hold the class weights for the weighting of the classes in the weighted loss function:

```
INDArray weights = Nd4j.create(new double[]{0.1, 0.4, 1.0});
```

We then pass these weights into a new instance of our loss function as highlighted in this code snippet:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(1)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(learningRate)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.RELU)
```

³¹ With regard to sampling variants, with multiple training epochs, there isn't generally much practical difference between the two with deep learning, whereas there is quite a difference between the two for things like SVMs and random forests.

```
.build())
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
    .weightInit(WeightInit.XAVIER)
    .activation(Activation.SOFTMAX)
    .lossFunction(new LossNegativeLogLikelihood(weights))
    .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();
```

The motivation behind the use of a weighted loss function is to change the optimization target to more heavily penalize incorrect predictions on the infrequent class. Ideally, this drives the network away from the poor local optima of always predicting the most commonly occurring class(es).



Understanding Relative Weight Values

A weight of 1.0 is equivalent to no weighting (e.g., {1.0, 1.0, 1.0}). In this configuration, all 1.0s is identical to providing no weight array at all. We want to use smaller weights for frequently occurring classes.

Dealing with Overfitting

Overfitting is the tendency of machine learning workflows to learn the training dataset so well that it performs poorly on unseen datasets. We'd say that this model does not generalize well to larger datasets.

If our model picks up chance occurrence patterns in the training data, our model will perform poorly on data beyond our test cases. If we train our model so that it learns the input data so well it cannot perform well on data beyond our test cases, our model will be not as useful as it could be. In both of these cases, we have overfit our training data and our model has picked up random correlations that will not benefit the model with scoring unseen data.



Better Generalization

Our goal with models in machine learning is to generalize the information contained in our training dataset such that the model will perform well on more data from similar sources.

All machine learning workflows have the tendency to overfit to some degree, the trick is to know when to stop training so that our model generalizes well yet picks up the least amount of overfitting possible. As we model more complex datasets with deep learning networks, we invariably use larger parameter counts in the networks. The trade-off becomes that we must add parameters at a rate that allows us to model more complex datasets while not doing it so fast that we introduce unnecessary overfitting.

To detect overfitting, we can evaluate the performance on a held-out test set. It is important that this test set isn't used for training; otherwise, it won't inform us as to how the network will perform on unseen data. As far as strategies to fix overfitting, we can use one or more of the following:

- Increased regularization (L1, L2, Dropout, DropConnect)
- Early stopping
- A larger training dataset
- A smaller network

Regularization is typically the first step in handling overfitting. L1 and L2 regularization can help (and has already been discussed). Dropout is a frequently used and usually quite effective approach to regularizing neural networks.



Overfitting Rule and Parameter Count

When the model is not allowed to have enough parameters, it will not be accurate. As the model takes on too many parameters, the accuracy on the training data will look good, but the model will overfit and the holdout data accuracy will diverge from the training data accuracy.

Using Network Statistics from the Tuning UI

The network statistics tool in the DL4J suite is a tool with which we can visualize and understand what the neural network is actually doing, in real time. By inspecting the current state (and trajectory) of things like the activations, gradients, and updates, we can identify network training/configuration issues, and implement strategies to correct the issues.

Setting up DL4J's tuning user interface is straightforward:

```
UIServer uiServer = UIServer.getInstance();
StatsStorage statsStorage = new InMemoryStatsStorage();
uiServer.attach(statsStorage);
int listenerFrequency = 1;
myNetwork.setListeners(new StatsListener(statsStorage, listenerFrequency));
```

See the user interface examples for details on how to configure the `StatsListener` to save the stats to a storage device, change the UI port, or post results to a remote UI. Note that we use `listenerFrequency` to set how frequently the information (activations, updates, etc.) is collected. We might want to increase this number to reduce the overhead associated with collecting the stats. Collecting stats information every 10 iterations can be sensible, or even less frequently (if we just want to track a reasonably well tuned network over a longer time period).

DL4J's training UI has a number of pages, the first of which is the Overview page. The Overview page comprises four sections, as depicted in [Figure 6-4](#).



Figure 6-4. The DL4J network statistics tool

Here's what the Overview page offers:

- Upper-left: model score versus iteration (and a moving average of the score)
- Upper-right: network information summary
- Lower-left: update:parameter ratios
- Lower-right: activations, gradients, and updates versus time

Here, the “score” refers to the value of the loss function (MSE, negative log likelihood, etc.), plus any regularization terms (e.g., L1, L2) that are added to the loss function. The second page in the UI is the model page, as shown in [Figure 6-5](#). This allows us to visualize the model structure and get more details for each layer individually.

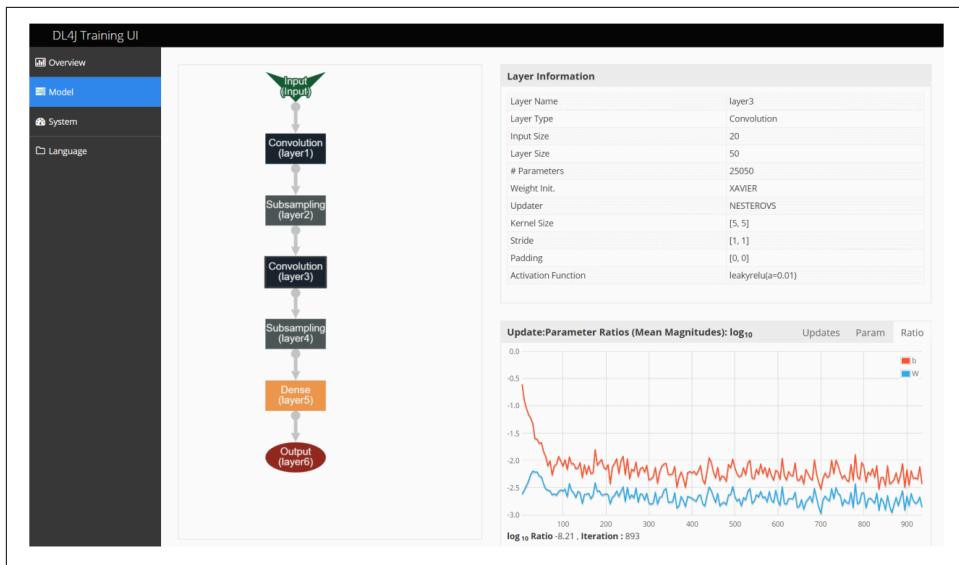


Figure 6-5. The DL4J training UI Model page

The Model page in Figure 6-5 is split into two sections. On the left, there is an overview of the network structure, showing the layers (and vertices, in the case of a ComputationGraph). On the right is a summary table (layer configuration) number of graphs, including the following:

- The ratio of updates to parameters for each parameter type, over time (it also has tabs for the mean magnitudes of the parameters and the mean magnitudes of the updates)
- The layer activations (mean and mean ± 2 standard deviations) over time
- Histograms for each parameter type for the layer
- Histograms of updates (for each parameter type) for the layer
- Learning rates over time for each parameter (note: this will be flat, unless we are using learning rate schedules)



Shutting Down the User Interface Server

We can shut down the UI by calling `UIServer.getInstance().stop();` or by simply terminating the JVM manually, after we are done. The UI server will remain operational until then, even if training has completed.

Finally, another useful tool is to calculate the loss (or accuracy) on a test set; that is, the data not used for training. This is more useful for identifying overfitting, whereas

the UI is used more to identify and fix optimization difficulties. It's common to begin tuning with the UI first, and later add calculation of test set loss/accuracy.

Detecting Poor Weight Initialization

In a nutshell, the ReLU and Xavier initializations are designed based on the network architecture (layer sizes) and assumptions about the activation function to do two things:

- Ensure that the variance of network activations are constant for all layers (i.e., activations don't grow too large or too small in later layers in the network)
- Ensure that the variance of activation (and hence parameter) gradients are constant for all layers; that is, gradients don't grow too large or small by the time they are backpropagated to the earlier network layers.

Unfortunately, it is not possible to guarantee that both of these requirements are simultaneously satisfied in general. Most notably, when adjacent network layers differ in size (e.g., a layer of size 1,024 followed by a layer of size 10, or vice versa), or when the activation function differs significantly from the Xavier/ReLU initialization assumptions (e.g., is not ReLU/tanh/identity, or something similar in shape) we might need to use a different initialization. This is more of an issue in deep networks because they might be able to get away with a suboptimal initialization in networks with only a handful of layers.

There are two ways to detect bad weight initialization. First, we can use the (\log_{10}) Activations Standard Deviation chart on the Overview page, as illustrated in [Figure 6-6](#).

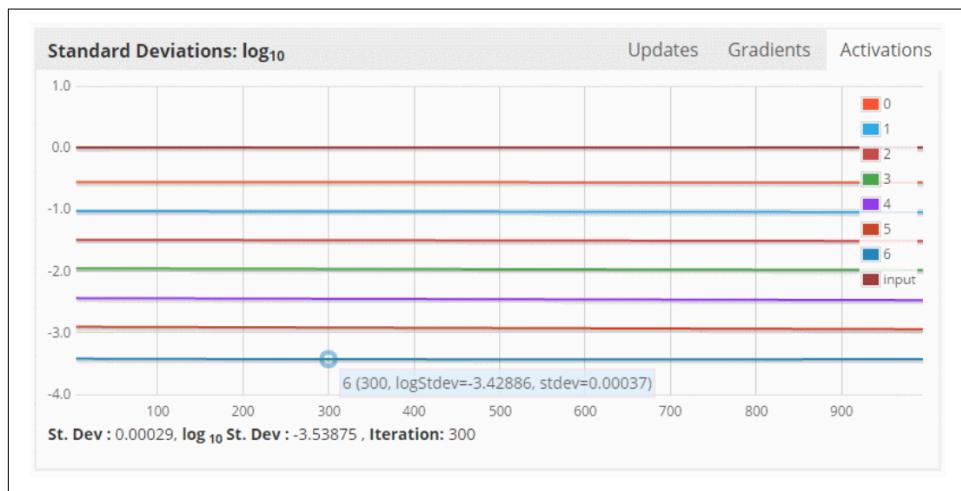


Figure 6-6. (\log_{10}) Activations Standard Deviation chart

This was produced by taking the ReLU weight initialization (with ReLU activation function) and reducing the weight initialization variance by a factor of three. With this (intentionally poor) weight initialization, the activations decrease layer by layer until they have virtually vanished by the final layer. It is also possible to detect this by looking at the charts of layer activations versus iteration, on the Model page, as seen in [Figure 6-7](#).



Figure 6-7. Layer Activations Charts

Activations also can grow too large if the weight initialization is too large. We can identify this by using the same two charts shown in [Figure 6-7](#).



In general, we want the log standard deviation of activations to be somewhere on the order of 1.0 for the hidden layers.

There is one thing you should know, when interpreting these charts: poor data normalization can look somewhat similar to poor weight initialization. Usually, we can identify this by directly looking at the activation charts; for example, we can see that the input standard deviation is 1 in [Figure 6-7](#), $\log_{10}(1) = 0$. This is a good scale for the inputs. Consequently, we also should check our data normalization if we see unusually large or small activations.

Detecting Nonshuffled Data

Neural network training is generally done by using mini-batches, where a mini-batch is defined as a subset of our data (typically 32 to 1,024 examples). To get better results during training, our data should be shuffled. This means that examples should be presented in random order, not (for example) all of one class, followed by all of another class in the same mini-batch. If we're doing a good job shuffling the training data, our training should look something like that depicted in [Figure 6-8](#).

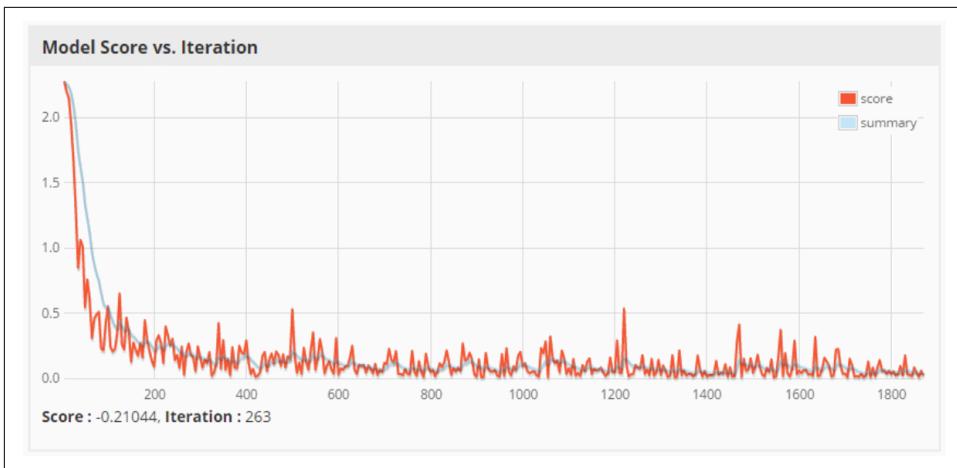


Figure 6-8. Training model score graph from UI

To illustrate how poor shuffling can adversely affect training, let's consider the case of the MNIST dataset previously mentioned in the book. Suppose we train a network to predict digits on MNIST, but instead of presenting the examples in random order (as we normally would) we first sort the examples by the label digit such that we train on mini-batches with only class 0 examples, and then with only class 1, and so on.

Figure 6-9, demonstrates that this poorly shuffled data produces a different pattern in the model score over time.

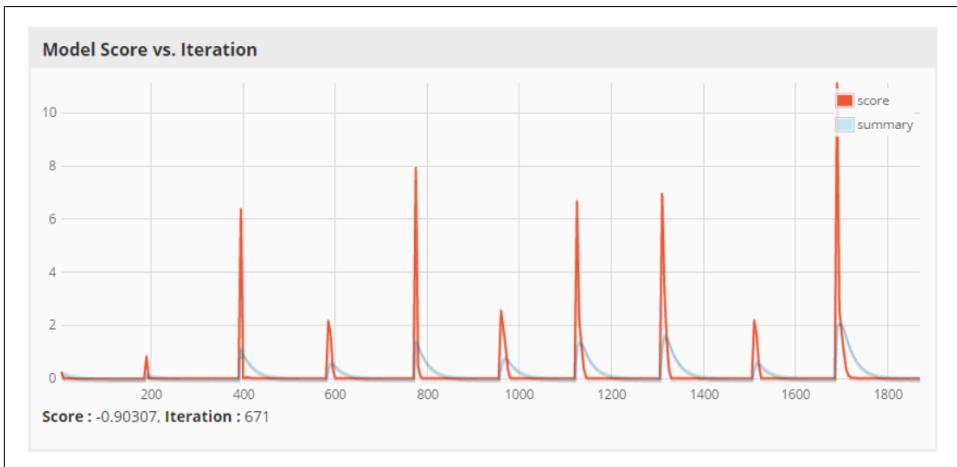


Figure 6-9. Training model scores with spikes

The model being used here normally would train decently on shuffled data and produce a graph similar to **Figure 6-8**. However, because the data was poorly shuffled we

see periodic large spikes in the training scores, which correspond to a switch between digits in the training mini-batches. The takeaway here is that when we see these periodic spikes, we most likely have an issue with how the training data was prepared. This is generally an easy fix in the data preparation pipeline and then we can be on our way.



A Note on the Order of Examples Within a Mini-Batch

It's also worth mentioning that shuffling the examples within a single mini-batch makes no difference. That is, if we have 32 examples in a mini-batch, it doesn't matter at all what the order of those 32 examples are. This is due to the fact that gradients are averaged over all examples in a mini-batch, before being applied to the model.

Detecting Issues with Regularization

Just like with setting the learning rate, setting a too large or too small value for the L1 or L2 regularization parameter value can lead to poor-quality training results. Setting too large a value can result in our weights being driven to 0. If we set too small a value this can contribute to overfitting or unstable learning in some cases. An overly large L1/L2 value (driving our weights to 0) can be detected using the Param tab on the Model page. The Param tab shows the mean magnitudes of the parameters, as illustrated in [Figure 6-10](#).

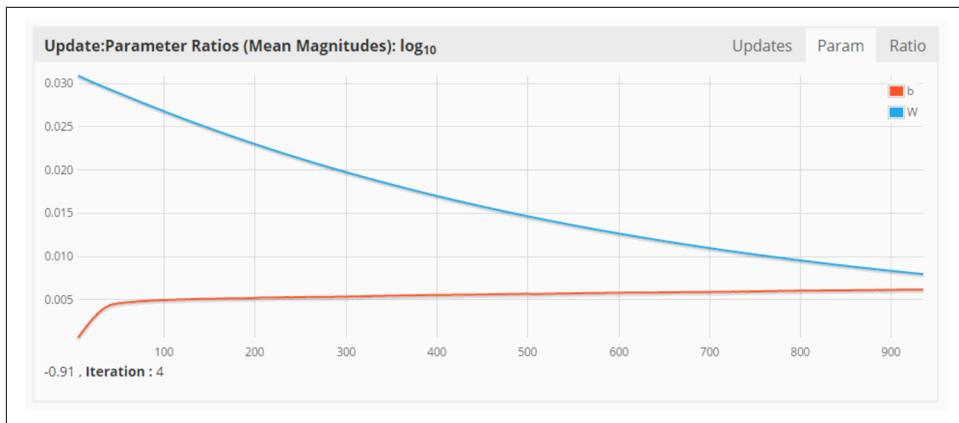


Figure 6-10. Weight values falling toward zero on the UI

We can see the mean weight magnitudes begin higher and then bleed off over the course of iterations, indicating our regularization methods have been too aggressive and we should lower the L1/L2 regularization parameter value.

Another issue we can detect with the same tool is detecting insufficient regularization. We'd like a nice, normal distribution shape for our weights in the Layer Parameters Histogram, as shown in [Figure 6-11](#).

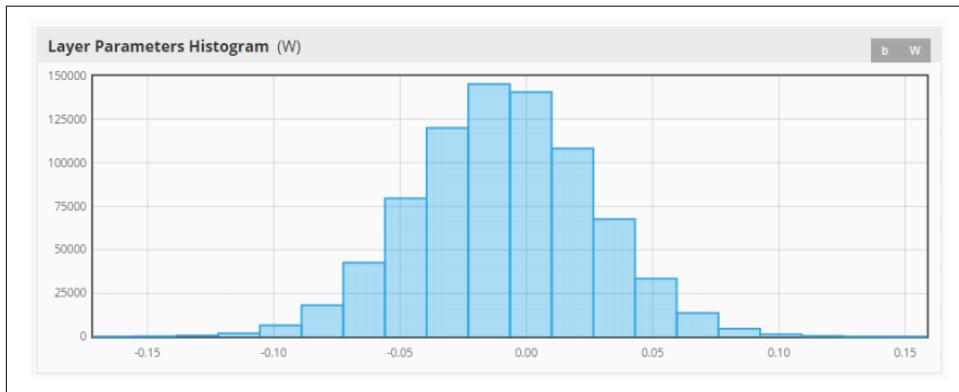


Figure 6-11. Layer Parameters Histogram panel with normal weights

It's also quite common for a small number of parameters to grow quite large when we don't have enough regularization. In this case, the Layer Parameters Histogram might, as depicted in [Figure 6-12](#).

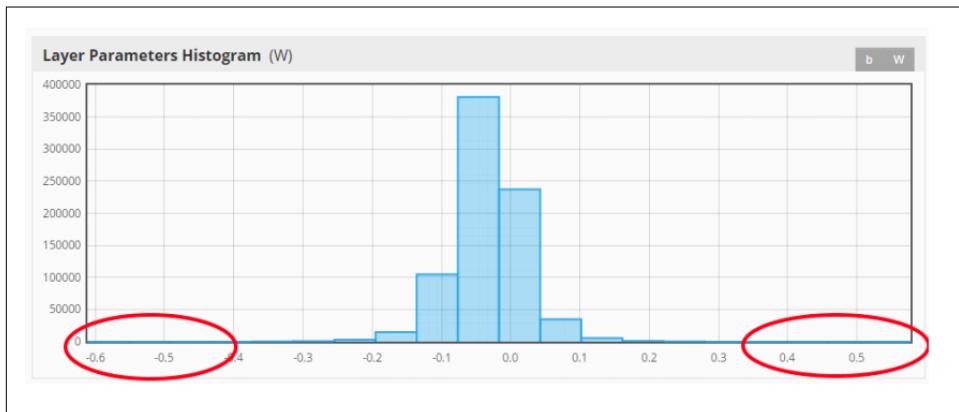


Figure 6-12. Layer Parameters Histogram with a few large weights

In [Figure 6-12](#), it can be a little difficult to see what's happening because there's only a small number that are growing large in this instance (note that the bar barely registers on the histogram). In extreme cases, the largest parameters can diverge toward $\pm\infty$ as training progresses.

Tuning Specific Deep Network Architectures

All you really need to know for the moment is that the universe is a lot more complicated than you might think, even if you start from a position of thinking it's pretty damn complicated in the first place.

—Douglas Adams, *Hitchhiker's Guide to the Universe*

In this chapter, we build on the concepts learned in [Chapter 6](#) on general deep network tuning. We take a deeper look at tuning specific architectures such as the following:

- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks
- Deep Belief Networks (DBNs)

Given that computer vision is one of the more popular applications in the world of deep learning, let's begin by learning to tune the CNN architecture.

Convolutional Neural Networks (CNNs)

CNNs have some general design patterns and then specific convolutional architecture design patterns. We reviewed general network design patterns in [Chapter 6](#). In this section, we review techniques that are relevant to the CNN architecture. Most of the content for this section is focused specifically on laying out patterns of convolutional layers and pooling layers.

The convolution stage uses multiple filters to learn different features from the input to the layer, as outlined in [Chapter 4](#). The output of this stage is transformed with an activation function—the Rectified Linear Unit (ReLU) activation function.



The Detector Stage

Some CNN literature will separate out the detector stage as its own layer or stage in the CNN architecture. This detector stage is simply the activation function for a layer and for DL4J we consider the activation function as part of the layer. For the purposes of this book, we group the concept of the detector stage as part of the convolution stage.

Pooling layers are commonly inserted between successive convolutional layers. We want to follow convolutional layers with pooling layers to progressively reduce the spatial size (width and height) of the data representation. This progressively reduces the number of parameters and needed computation in the network. Reducing the parameter size in this way also helps with overfitting. Most other topics in tuning, such as output layer strategies, have already been covered in [Chapter 6](#).

Common Convolutional Architectural Patterns

We often see a single convolutional layer between every pool layer. We see this pattern demonstrated in the following:

- Input layer
- Convolutional layer
- Pooling layer
- Convolutional layer
- Pooling layer
- Fully connected layer
- Another (possible) fully connected layer

Sometimes, we'll see multiple convolutional layers in a sequence before we see a pooling layer.



In larger networks it's a good idea to see two convolutional layers stacked before every pooling layer. This allows the larger networks to develop more complex features before we downsample with the pooling layer.

To further understand the architectural pattern, let's revisit the LeNet example from [Chapter 5](#). This network, first published in 1998 by Yan LeCun, is one of the most famous convolutional architectures. It has been shown to successfully model MNIST, as we did in [“Modeling Handwritten Images Using CNNs” on page 182](#). Here's a list of the specific layer architecture:

- Input layer
- Convolutional layer: 20 filters [5×5]
- Max pooling layer: [2×2]
- Convolutional layer: 50 filters [5×5]
- Max pooling layer: [2×2]
- Fully connected layer

Example 7-1 shows this architecture represented in a configuration in Java code from the example in [Chapter 5](#). In the interest of brevity, we've included only the configuration portion of the code here so that you can compare the just-described architecture layout to the specific LeNet configuration.

Example 7-1. LeNet DL4J model configuration in Java

```
/*
   Construct the neural network
*/
log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations) // Training iterations as above
    .regularization(true).l2(0.0005)
    .learningRate(.01)
    .weightInit(WeightInit.XAVIER)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn and nOut specify depth. nIn here is the nChannels and nOut
        //is the number of filters to be applied
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
    .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
    .layer(2, new ConvolutionLayer.Builder(5, 5)
        //Note that nIn need not be specified in later layers
        .stride(1, 1)
        .nOut(50)
        .activation(Activation.IDENTITY)
        .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

```

    .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
          .nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
          .nOut(outputNum)
          .activation(Activation.SOFTMAX)
          .build())
    .setInputType(InputType.convolutionalFlat(28,28,1)) //See note below
    .backprop(true).pretrain(false).build();
}

```

In Example 7-1, each convolutional layer is followed by a pooling layer. The network is capped off with a fully connected DenseLayer followed by an output layer with a softmax activation function. The softmax activation function here is used to detect one of the 10 MNIST digits in the training/test dataset. Here is another method to note:

```
.setInputType(InputType.convolutionalFlat(28,28,1))
```

This method call does a few things, including the following:

- Adds preprocessors, which handle things like the transition between the convolutional/subsampling layers and the dense layer
- Does some additional configuration validation
- Where necessary, sets the `nIn` (number of input neurons, or input depth in the case of CNNs) values for each layer based on the size of the previous layer
 - It won't override values manually set by `InputTypes`
 - It can be used with other layer types, too (Recurrent Neural Networks, MLPs, etc.), not just CNNs

In general, in terms of designing convolutional architectures, we should focus on using multiple smaller convolutional layers as opposed to a single convolutional layer with one large receptive field. This allows our network to pick up more nonlinear dynamics in the data with more expressive features.

For the first convolutional layer, the input depth (`nIn`) must match the data, but the number of channels (`nOut`) is a free parameter. The “number of units” generally refers to `nOut`. Ideally, we want this input layer size to be divisible by two more than once so that we have enough information to downsample over time and build features. An example of an input size for a CNN to model CIFAR-10 (discussed in [Chapter 2](#)) would be an input layer size of 32. Most often we'll want to use the ReLU activation function for our convolutional layers.

The input data channels are controlled by the `.nIn()` method, as demonstrated here:

```

.layer(0, new ConvolutionLayer.Builder(5, 5)
      //nIn and nOut specify depth. nIn here is the nChannels and nOut
      //is the number of filters to be applied
      .nIn(nChannels)
}

```

You can configure this to the type of input, but it will typically be 1 for monochrome and 3 for RGB. Of course, if the input were not image data and had a multidimensional representation, this number could be much larger.



Some Caveats About First Convolutional Layer Sizing

In general, the input layer unit count depends on the stride/filter size.

An interesting example is AlexNet:¹ $224 \times 224 \times 3$ input, with 11×11 kernels (uncommonly large these days).

In practice, we can crop or scale our data to a size that works for the network configuration we want to use. Better to crop/scale down, though, if possible (not up).

Beyond the first convolutional layer, we'll have a repeating pattern of more convolutional layers followed by pooling layers. In some architectures, such as VGGNet, we'll see multiple (e.g., two) convolutional layers consecutively before we see a pooling layer.

In general, CNN configurations tend to begin with larger filter sizes before progressively decreasing in size.



No Magic Bullet on Convolutional Layer Sequences

As of this writing, there's no one magic convolutional architecture that will be best for all of your image modeling problems. We suggest beginning with a known successful architecture such as LeNet, VGGNet, Inception,² or AlexNet and getting it to work on your intended problem. From there you can tweak the layer sequences and the hyperparameters, using the known architectures as "known good starting solutions" from which to begin.

Our output layer is going to follow the patterns set forth in [Chapter 6](#).

Configuring Convolutional Layers

We want to set up the spatial arrangement of the convolutional layer and then choose the number of filters.

¹ Krizhevsky et al. 2012. ["ImageNet Classification with Deep Convolutional Neural Networks."](#)

² Szegedy et al. 2015. ["Going Deeper with Convolutions."](#)

How we set up the hyperparameters in a convolutional layer dictates how many neurons there are in the output volume and how they are arranged. Here are key hyperparameters:

- Filter size
- Stride
- Padding

For example, if we were to build a convolutional layer with 5×5 filters, the code might look similar to this:

```
ConvolutionLayer convLayer = new ConvolutionLayer.Builder()  
    .kernelSize(5,5).stride(1,1).padding(2,2)  
    .name("first_layer")  
    .nOut(out)  
    .biasInit(bias)  
    .build();
```

In this sample we can see the filter size (.kernelSize) is 5×5 and we take the integer parameters for both the stride (1,1) and padding (2,2) parameters. We use the .name() method to give this convolutional layer the name "first_layer" and we set the number of filters by using the .nOut(out) method call. The output of the layer itself is more complex than just the number of units out. Later on, we'll work through the calculations for projecting the output volume size.

Setting the stride for filters

For stride, we allocate depth columns around the spatial dimensions (width, height). As our stride increases, our receptive fields overlap less and the output volume has smaller dimensions spatially. We see this parameter being set for the convolutional layer in the previous example, in the line:

```
.stride(1,1)
```

This means our filter will “slide” 1 input grid cell over and 1 input grid cell down each time the filter is applied (for more on this, see [Chapter 4](#)). If our stride is 2, the filter will jump or slide 2 pixels at a time.



Strides Above 2

Stride settings above 2 are uncommon and rare in practice, especially with smaller filter sizes. Strides larger than the filter size should be avoided entirely.

As the stride value goes up, the output volume spatially goes down. It's been demonstrated that smaller strides (e.g., 1) in convolutional layers work better in practice.

This lets the pooling layers do the downsampling work and allows the convolutional layers to focus on just transforming the volume with respect to depth.

Using padding

In some components of certain architectures (like Inception) we want to preserve the spatial size of the input volume (covered in further detail coming up). To do this, we use zero-padding to control the spatial size of the output volumes. We often want to pad the input volume with zeros such that the convolutional layer does not change the spatial dimensions of the input. Zero-padding also preserves input information along the borders.

Choosing the number of filters

Each filter can look for something different in the input training data. As data variation and complexity rises, logically we'll need more filters to capture the relevant features. [Figure 7-1](#) shows learned filters from the first convolutional layer in a network.

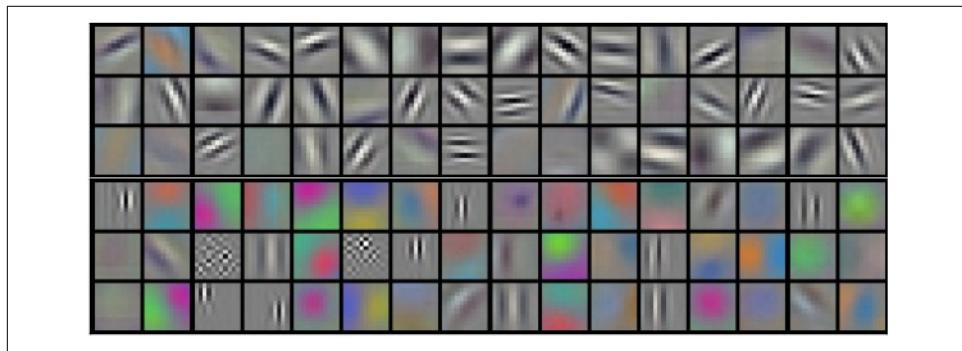


Figure 7-1. Visualized 96 learned filters, $11 \times 11 \times 3$ size³

We want to be judicious with choosing a filter count for a convolutional layer because computing the activations of a single convolutional filter is more expensive than traditional multilayer neural network layers. In DL4J, we configure the filter count with the `nOut(int)` option in the configuration.

As we go further along in the progression of layers in a CNN, the activation map size decreases. The layers closer to the input layer will tend to have fewer filters. As we move toward the output layer in a CNN, we tend to see more filters.

Although there is no specific heuristic to set the number of filters per convolutional layer, we should mention that some of the larger CNN architectures [start around 64](#)

³ Krizhevsky, Sutskever, and Hinton. 2012. “[ImageNet Classification with Deep Convolutional Neural Networks](#).”

and end around 1,024 (at the high end) or so, increasing as we progress further from the input. Table 7-1 presents filter counts for three commonly used CNN architectures.

Table 7-1. Filter count examples

CNN	Convolutional layer filter count progression
LeNet	20, 50
AlexNet	96, 256, 384, 384, 256
VGGNet	64, 128, 256, 512, 512



Computation Cost Versus Data Preservation

We want to keep computation cost roughly the same across all of the layers, so the product of the number of features and the number of pixel positions for each layer needs to be roughly the same. This means that as we build up features from the input, we tend to lose input data through pooling layer downsampling. This allows us to preserve information about the input but in a different form (features) all the way to the end of the network.

Configuring filter size

Smaller filters in multiple stacked convolutional layers tend to perform better⁴ as opposed to big filters in fewer convolutional layers.⁵

As the filter size grows is larger, the computational cost grows as well as we have to look at larger regions of the input per filter. Convolutions with larger spatial filters (e.g., 5×5 or 7×7) tend to be disproportionately expensive in terms of computation. For example, a 5×5 convolution with n filters over a grid with m filters is $25/9 = 2.78$ times more computationally expensive than a 3×3 convolution with the same number of filters.

We want to select a feature size that is obviously smaller than the input volume spatial dimensions yet big enough to capture relevant features. For our convolutional layers, we want to use small filters (3×3 or 5×5) and a stride of 1.⁶ For comparison, Goo-

⁴ Simonyan and Zisserman. 2014. “Very Deep Convolutional Networks for Large-Scale Image Recognition.”

⁵ Ciresan introduced the concept of small filters in 2011, but the networks were not as deep as others of the time. Goodfellow applied a similar approach in 2014 to street sign recognition. GoogleLeNet (Szegedy) also used the approach to win ImageNet in 2014.

⁶ A good example is MNIST input images have a spatial dimension of $[28 \times 28]$ and generally have $[5 \times 5]$ -sized filters on the first convolutional layer.

gLeNet⁷ is a relatively complex CNN and the filters vary between 1×1 , 3×3 , and 5×5 in size.

Another popular network, VGGNet,⁸ has some interesting properties that have proven successful. The convolution layer filter size is usually 3×3 , which is the smallest size that retains the concept of capturing pixels (e.g., left, right, up, down, and center).



Comparison of Filter Size Versus Consecutive Layers

Stacking three convolutional layers with filters of 3×3 and pooling is similar to a single convolutional layer with a 7×7 filter.

Adding more convolutional layers and filters allows us to take advantage of more regularization in the learning process. More layers apply more nonlinear transformations and reduce the amount of parameters applied.

Another approach to get going on layer sequences and filter shapes is to begin with successful architectures like VGGNet or GoogLeNet and draw inspiration from patterns⁹ observed in those networks.



A Caution on Filter Size from the v3 Inception Paper

We note the following:

The above results suggest that convolutions with filters larger 3×3 might not be generally useful as they can always be reduced into a sequence of 3×3 convolutional layers.¹⁰

Convolution mode and calculating spatial size of output volume

Here's the general formula we use to calculate the spatial size of the output volume as a function of the input volume size:

$$\text{Output volume size} = (W - F + 2P)/S + 1$$

Table 7-2 defines the variables.

⁷ Also called "Inception," responsible for setting the new state-of-the-art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC 2014)

⁸ Simonyan and Zisserman. 2014. ["Very Deep Convolutional Networks for Large-Scale Image Recognition."](#)

⁹ Filter/kernel sizes in networks such as AlexNet are a bit big by modern standards. As always, it might be necessary to test and keep in mind that we can set our image size after the fact to help things to work better (e.g., crop or scale to some size).

¹⁰ Szegedy et al. 2015. ["Rethinking the Inception Architecture for Computer Vision."](#)

Table 7-2. Variables for spatial size of output volume

Variable	Description
W	Input volume size
F	Receptive field size of convolutional layer neurons
S	Stride setting
P	Zero-padding setting

The result of this equation should be an integer; if it is not, the configuration is incorrect. If the result is not an integer, we might be able to fix it with padding, a different filter size, or by using the Truncate mode (described in a moment). We want to be mindful that convolutional layers preserve the spatial size of their input while the pooling layers reduce the spatial size of input through downsampling.



Keeping Track of Input Volumes

If we use a stride greater than 1 or we don't zero-pad the input in convolutional layers, we need to keep track of input volumes through the convolutional network. We need to ensure that all the strides and filters balance out across our network.

DL4J does have some input validation that can catch many invalid configurations for this. However, to know how to fix issues (or to know what's valid in each layer), we still need to track the volumes throughout. Also note that there are some configurations that any amount of padding won't fix.

In DL4J, the `ConvolutionMode` defines how convolution operations should be executed for the convolutional and subsampling layers, for a given input size and network configuration (specifically stride/padding/kernel sizes).

Currently, **three modes** are provided:

1. Strict
2. Truncate
3. Same

With Strict mode, the output size for the convolutional and subsampling layers are calculated as follows, in each dimension:

$$\text{outputSize} = (\text{inputSize} - \text{kernelSize} + 2 * \text{padding}) / \text{stride} + 1.$$

If `outputSize` is not an integer, an exception will be thrown during network initialization or forward pass.

With Truncate mode, the output size for the convolutional and subsampling layers are calculated in the same way as in Strict in each dimension. If `outputSize` is an integer, Strict and Truncate are identical. However, if `outputSize` is not an integer, the output size will be rounded down to an integer value.



Consequences of Rounding Down

The major consequence of this rounding down is a border/edge effect will be seen if/when rounding down is required. In effect, some number of inputs along the given dimension (height or width) will not be used as input and hence some input activations can be lost/ignored. This can be problematic higher in the network (where the cropped activations can represent a significant proportion of the original input) or with large kernel sizes and strides.

The Same mode operates differently than Strict/Truncate, in three key ways:

- Manual padding values in the convolution/subsampling layer configuration is not used; padding values are instead calculated automatically based on the input size, kernel size, and strides.
- The output sizes are calculated differently (more on this in a moment) compared to Strict/Truncate. Most notably, when `stride = 1` the output size is the same as the input size.
- The calculated padding values can be different for top/bottom and left/right (when they do differ: right and bottom might have 1 pixel/row/column more than top/left padding).

Configuring Pooling Layers

Pooling functions replace the output of a layer at a specific spot with the summary statistic of the nearby connected neuron outputs. This helps the model representation be more invariant to small translations, in the input data. Pooling layers themselves don't have parameters to deal with because they compute a fixed function of the input. Zero-padding is not commonly used with pooling layers.



Small Variation Invariance Through Pooling

A model that has the ability to be invariant to small variations, or invariant to local translations is a useful aspect when training on image data.

The use of pooling has the conceptual effect of an infinitely strong prior where the function learned by the convolutional layer must be invariant to small variations. This gives us a much stronger convolutional network in terms of statistical efficiency of the network.

We typically set the max pooling shape (e.g., spatial dimensions: width, height) of the downsampling operation in pooling layers as $[2 \times 2]$ as illustrated in the following code snippet:

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
      .kernelSize(2,2)
      .stride(2,2)
      .build())
```

We need to keep in mind that if we increase the pooling spatial dimensions too quickly we run the risk of losing too much information.

Max-pooling tends to occur in two variations in pooling layers in practice:

- Receptive field size: 3, Stride: 2 (overlapping pooling)
- Receptive field size: 2, Stride: 2 (more common variation)



If we increase our pooling receptive field beyond these settings we tend to lose too much information.

Max-pooling has become more popular recently than other variations such as average pooling or L2-norm pooling. Max-pooling has shown to work better in practice than these other variations.

Transfer Learning

It's rare to see a CNN model trained from randomly initialized parameters due to the amount of processing power/time needed for the task. In [Chapter 4](#), we learned how one driver of the popularity of deep learning has been the availability of quality large-scale datasets such as ImageNet (1.2 million labeled training images). Although benchmark training datasets such as ImageNet have become better, we seldom get a large labeled training dataset to perform largescale image training with CNNs. One way to augment this issue is to begin with a previously trained CNN model that had

good results and then train it further on a specific image dataset. This is referred to as transfer learning.¹¹

An alternative to training from scratch

CNNs have been shown to learn general visually generic features on the first few layers and then gradually build up dataset-specific features on later layers. These early layer features are similar to Gabor filters and color blobs and are the “building blocks” of vision processing in general. Following are two use cases of note in transfer learning:

- Fine-tuning an existing model
- Using an existing convolutional model as a feature extractor

Although both of these methods use a pretrained model typically on ImageNet, the way the final tuning is done is the main difference. We give some details for each below.

Fine-tuning an existing convolutional model

In this variant of transfer learning, we see a CNN model trained on a large dataset such as ImageNet, and then we replace the last “classifier layer” with something specific to the fine-tuning dataset. Some variants will continue to perform backpropagation on all of the layers, and others will update only the later layers. This is because many of the earlier-layer features are general to all types of vision processing and there is less need to update them. The later layers are focused on combining these lower-level features in task-specific ways and are more relevant to train on the domain-specific dataset.

Using an existing convolutional model as a feature extractor

In this version of transfer learning, we take the last fully connected layer—output layer—and consider the rest of the CNN as a “feature extractor” for the smaller dataset. This is also relevant because a pretrain dataset such as ImageNet would have 1,000 different class outputs in the output layer and would possibly be not as relevant for a domain-specific task.

¹¹ Yosinski et al. 2014. “How transferable are features in deep neural networks?”



Training ImageNet from Scratch

Training a CNN can take up to two to three weeks to train across multiple graphics processing units (GPUs) on ImageNet. It's common for practitioners to release their model for others to use as pretraining starting points to continue training their own models. The Caffe project has a [model zoo](#) from which we can download many pretrained models, as well.

When to consider trying transfer learning

Consider trying transfer learning in the following (or combinations of) situations:

- Training dataset is small
- Training dataset shares visual features with the base dataset

In other cases, you might find success using transfer learning.



A Note About Learning Rates in Transfer Learning

For fine-tune activities in transfer learning, we recommend using a lower learning rate for the smaller fine-tune dataset as the pretrained weights are already probably good.

Recurrent Neural Networks

Recurrent Neural Networks share some similarities with other architectures such as CNNs, but they have their own set of challenges and hyperparameters. As with most neural networks, there is no sure way to instantly find the best settings. We commonly see hyperparameters that stall the learning process or even make it worse than previous variations. We remind you that for all networks, it's always going to be a game of trial and error.



DL4J and Long Short-Term Memory Networks

DL4J currently supports the most popular blend of Recurrent Neural Networks: the Long Short-Term Memory (LSTM) model. Other variants of Recurrent Neural Networks are actively under development.

DL4J also supports the bidirectional LSTM.

The next most common variant of Recurrent Neural Networks are Gated Recurrent Unit (GRU) and “vanilla” Recurrent Neural Networks.

Network Input Data and Input Layers

Traditionally, input data to a canonical feed-forward neural network is either a single vector (one-dimensional) or a matrix (two-dimensional, containing a mini-batch of vectors for training). Recurrent Neural Networks have a third dimension to the input data representing the time dimension of the input time-series data. In DL4J, we represent this input with the following parameters (see also [Figure 7-2](#)):

- Number of examples
- Input size (number of columns)
- Time-series length

Constructing volumetric input data for Recurrent Neural Networks takes more effort than traditional networks such as multilayer perceptrons. As we discussed previously in the book (and as shown in [Figure 7-2](#)), there are three dimensions for input:

- Mini-batch size
- Number of columns in our vector per time-step
- Time-series length

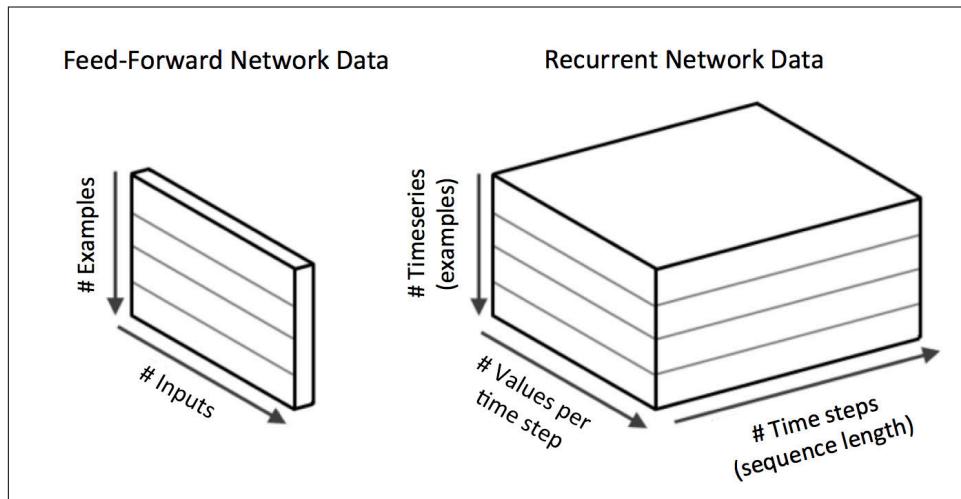


Figure 7-2. Representing 3D input for Recurrent Neural Networks

To understand more about how to do this easily, see [Appendix E](#) and [Appendix F](#). We'll also look at general patterns of time-series vectorization in [Chapter 8](#).



Standardization

It generally helps all neural networks to standardize the input data (e.g., zero mean, unit variance). This helps transform the inputs into a range more suitable for the standard activation functions.¹²

Standardization helps the relationship between the inputs and the targets to be as simple and localized as possible. Keep this in mind for only *real-valued* inputs. Don't do this for one-hot (categorical) inputs.

There are some basic variations in how we train and tune Recurrent Neural Networks.

Output Layers and RnnOutputLayer

The output of a Recurrent Neural Network has three dimensions:

- Mini-batch size (number of examples)
- Output size (number of columns)
- Time-series length

Given that the matrix representation in DL4J is based on the `INDArray` class, we see that the values match up to the indexing scheme (i, j, k), as listed in [Table 7-3](#).

Table 7-3. Understanding variables in a Recurrent Neural Network input matrix

Variable	Description
i	Index of example in mini-batch
j	Index of column per time-step
k	Time-step index

In DL4J, we use the `RnnOutputLayer` for the final layer with many Recurrent Neural Networks for the following tasks:

- Regression
- Classification

`RnnOutputLayer` handles functions such as these:

- Score calculation
- Error calculation (prediction versus actual) given a loss function

¹² Graves. 2012. “Supervised Sequence Labelling with Recurrent Neural Networks” (PhD Thesis).

We can see `RnnOutputLayer` is functionally similar to the standard `OutputLayer` in DL4J for canonical feed-forward networks, yet it can deal with three-dimensional output. Configuration for the `RnnOutputLayer` follows the same design as other layers. For classification, we set the last layer in a `MultiLayerNetwork` to a `RnnOutputLayer`, as shown in [Example 7-2](#).

Example 7-2. Configuration for the `RnnOutputLayer`

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .activation(Activation.SOFTMAX)
    .weightInit(WeightInit.XAVIER)
    .nIn(prevLayerSize)
    .nOut(nOut)
    .build())
```

To get a better idea of this layer in action, check out the Recurrent Neural Network example in [Chapter 5](#), in which we classified sensor readings.

Training the Network

Recurrent Neural Network training can be computationally expensive. In general settings, we see Stochastic Gradient Descent (SGD) as a good baseline optimization algorithm for Recurrent Neural Networks. Research has shown Hessian-free optimization to be helpful for training Recurrent Neural Networks, as well.¹³

Initializing weights

Initializing weights is important in training Recurrent Neural Networks. Good initialization of weights has been shown to create hidden units that can convey information over long enough distances to model long-term dependency tasks.¹⁴

For most Recurrent Neural Network layers in DL4J, we recommend using the Xavier initialization,¹⁵ as demonstrated in the following code snippet:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
```

¹³ *Ibid.*

¹⁴ Sutskever. 2013. “Training Recurrent Neural Networks” (PhD Thesis).

¹⁵ Orthogonal weight initialization for Recurrent Neural Networks has shown promise, as well.

```
.l2(0.001)  
.weightInit(WeightInit.XAVIER)
```

Backpropagation through time

When our recurrent network is dealing with long sequences with many time-steps, we find that truncated backpropagation through time (BPTT) works well. BPTT is an extension of the standard backpropagation algorithm for recurrent neural networks (as introduced in [Chapter 4](#)). Truncated BPTT reduces the computational complexity of each parameter update in a Recurrent Neural Networks. In the code example that follows, we see that using truncated BPTT with DL4J and recurrent neural networks is easy:

```
.backpropType(BackpropType.TruncatedBPTT)  
.tBPTTForwardLength(100)  
.tBPTTBackwardLength(100)
```

This added setting to your network will cause the training to use truncated BPTT with forward and backward passes of lengths set by the parameters for each as just shown. Here are some other notes about usage in DL4J:

- DL4J will use full BPTT as the default unless you specify truncated BPTT.
- Typically, we want to set the length option between 50 and 200 time-steps (application-dependent, of course)
 - Forward and backward time-step lengths tend to be the same length
 - Backward time-step length can be shorter but not longer

The truncated BPTT time-step lengths need to be shorter than or equal to the length of the input time-series.

Regularization

Dropout is a common way to use regularization with Recurrent Neural Networks. In Recurrent Neural Networks, dropout is applied to only a subset of connections in LSTMs (nonrecurrent connections). This variant of dropout has been shown to help.¹⁶ Dropout as implemented for Recurrent Neural Networks in DL4J is done on the input connections/activations only, not the recurrent connections.



Standard Dropout and Recurrent Neural Networks

Standard dropout has been shown to not work well with Recurrent Neural Networks as the recurrence amplifies noise, which hurts learning.

¹⁶ Zaremba, Sutskever, and Vinyals. 2014. “Recurrent Neural Network Regularization.”

Recurrent Neural Networks are more sensitive to learning rate and momentum settings than multilayer perceptron networks.¹⁷

Debugging Common Issues with LSTMs

Although it is fair to say that neural network tuning involves a lot of trial and error, after a while we begin to see some patterns that can help us tune things faster. We've seen too many people attempting to tune their networks by blindly changing things, and it's not the best path. Sometimes, gradients become occasionally very large (called "exploding gradients," and are visible on a mean magnitude graph for updates). One side effect with exploding gradients is loss function values that increase for multiple mini-batches. The real danger here is that with large gradient values we end up making large changes to the parameters and potentially damaging the learned features in the model. To help with this we add gradient clipping or gradient renormalization. The exploding gradients issue is a common drawback with vanilla Recurrent Neural Networks.

Exploding and Vanishing Gradients

Recurrent Neural Networks are known to have issues with the "vanishing gradient problem" in addition to "exploding gradients." The vanishing gradient problem issue occurs when the gradients become too small and make it difficult to model long-range dependencies (10 time-steps or more) in the structure of the input dataset.

The most effective way to get around the vanishing gradient issue¹⁸ in recurrent neural networks is to use the LSTM variant of Recurrent Neural Networks, which DL4J supports. Other methods to deal with this issue include the following:

- Combining short and long paths in unfolded flow graphs (time-delay neural networks)
- Leaky units and a hierarchy of different time units
- GRUs
- Better optimization methods
- Gradient clipping (for exploding gradients)
- Regularization to encourage information flow
- Using L1/L2 penalties

¹⁷ Sutskever. 2013. "[Training Recurrent Neural Networks](#)" (PhD Thesis).

¹⁸ LSTMs help solve the vanishing gradient issue but do not inherently solve the exploding gradient issue.

Padding and Masking

Padding involves adding zeros (“pad”) to the end of any time-series that is shorter than the longest time-series in the mini-batch. This allows the training data to be a rectangular array (matrix) which is required for training while training on time series of different lengths in the same mini-batch. Masking involves two additional arrays that indicate for each value whether the input or output was originally present in the input or was a padded value.

Padding and masking in DL4J allows us to support the follow variations on recurrent model training:

- One-to-many
- Many-to-one
- Variable length time-series (in the same mini-batch)

Figure 7-3 presents examples of each of these.

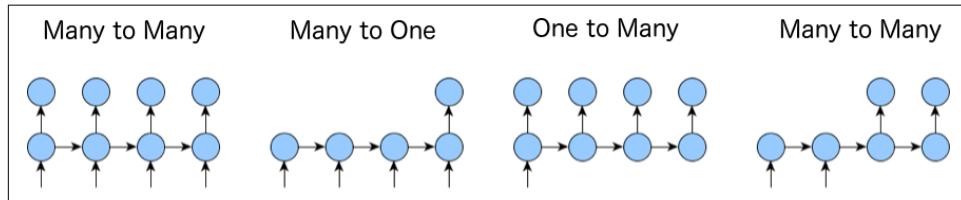


Figure 7-3. Different variations of Recurrent Neural Network training

Padding and masking help Recurrent Neural Networks with inputs or outputs that do not occur at every time-step.



The Usefulness of Padding and Masking

In the absence of masking and padding, we can only work with the many-to-many training case. In this scenario, all input records are the exact same length and have inputs and outputs at all time-steps.

Applying padding and masking to volumetric input

Training Recurrent Neural Networks in DL4J involves the following parameters for the shape of the mini-batch:

- Mini-batch size
- Input size
- Time-series length

The padding array involves the following parameters for both the input and output:

- Mini-batch size
- Time-series length

For each value in the two-dimensional array, we have either a 0 or 1 for whether the given time-step in the sequence was originally absent or present, respectively, in the input data before padding. The masking arrays for the input and output data are stored separately.

If we use a masking array with all 1s, it has the same effect as no masking array at all. In the case of many-to-one, we'd have a single masking array for the output only. In DL4J these masking arrays are created during the data import stage (e.g., `SequenceReaderDatasetIterator`) and are represented in the `DataSet` object. The `MultiLayerNetwork` object in DL4J knows how to deal with the masking arrays if they exist during training.

Evaluation and Scoring With Masking

We use mask arrays when we're doing scoring and evaluation—that is, when we're evaluating accuracy of the model—with Recurrent Neural Networks. In the many-to-one cases, we have only a single output for each example, and the evaluation needs to take this into consideration.

Classification using the evaluation class

We use output mask arrays during evaluation by passing them to the `Evaluation` object, as shown in [Example 7-3](#).

Example 7-3. Evaluation setup

```
Evaluation.evalTimeSeries(INDArray labels, INDArray predicted, INDArray outputMask)
```

[Table 7-4](#) provides explanations for each of the variables in [Example 7-3](#).

Table 7-4. Explanation of variables

Variable	Description	Dimensions
labels	Actual output of training data	3-dimensions
predicted	Generated labels from network	3-dimensions
outputMask	Mask array for output	2-dimensions

We notice that the input mask array is not required for evaluation, because only the network outputs are relevant at evaluation time.

Scoring new data with MultiLayerNetwork

We can also use mask arrays for score calculation of models. Score calculation is different from the use we saw previously. We are calculating the value of the loss function value, not metrics such as accuracy or F1. However, in both cases, we want only to score/evaluate those time-steps that are actually present, not those that are simply padded time-steps. We can score time-series data by using the `MultiLayerNetwork` class, as shown in the following code example:

```
MultiLayerNetwork.score(DataSet)
```

Just as in the previous example of masking in DL4J, if the `DataSet` contains an output masking array, it will automatically be applied to calculating the score (loss function) for the network.

Variants of Recurrent Network Architectures

There are some cases for which the user might want to combine different types of architecture layers to model different mechanics of data, such as video data. In some cases, we might want to combine LSTM and CNN layers to model a sequence of images (e.g., video frames). DL4J provides special preprocessor classes to help in these architecture variants such as `CnnToRnnPreProcessor` and `FeedForwardToRnnPreProcessor`.



You can add Recurrent Neural Network preprocessors manually, but in many cases, they will be added automatically by the network.

Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBMs) are a type of neural network used to learn features from datasets in an unsupervised manner, by mapping input data to a hidden state and then attempting to reconstruct the input from the hidden state. In this respect, they are similar to other models for unsupervised training such as autoencoders (denoising, compression, and variational variants) though the training procedure for RBMs (contrastive divergence) is quite different than for autoencoders.

Typically we're focused on using RBMs for use cases such as the following:

- Unsupervised feature learning in DBNs
- Data reconstruction
- Recommendation engines



MNIST Example

An example would be when we're learning the MNIST dataset and our input layer has 784 input neurons. We would choose a lower number of neurons for the hidden layer. In this example, we'll select a hidden layer of 500 units (about two-thirds as much as the visible input layer). For the second hidden layer, we'll select a neuron count of 250 neurons.

Let's now think about how we'd set up hidden units for arbitrary problems beyond MNIST.

Hidden Units and Modeling Available Information

We don't do this the same way as we do in discriminative machine learning. With discriminative learning, a training case imposes a constraint on the model parameters equal to the number of bits that it takes to specify the label.



Information and Example Labels

Labels hold at most only a few bits of information, so this doesn't give us a tremendous amount to work with. In some cases, where we'd use more parameters than training cases, we see severe overfitting in the model constructed (potentially, depending on regularization).

The number of bits in the input vector can be several orders of magnitude greater than the number of bits used to represent the label. In this way, we can see that the data input vector has quite a bit more information to learn from than the label.

Using the raw information contained implicitly in the input data, we can take advantage of this formation to more effectively extract features and/or allow for more effective discriminative classification later in the pipeline (when used as part of DBNs). What we want to look at is how much information is available to us to use. How much information is in your training record affects how you select the number of hidden units.



Quick Example

Number of training images: 10,000

Pixels per image: 1,000

Projected globally connected hidden units: 1,000

We can see that the number of hidden (globally connected) units in this example is tied directly to how much information is in the input vector or image in this case. In

RBM networks that are not globally connected or use weight-sharing schemes, we can use more units. Now let's look at a heuristic for choosing a specific number of hidden units for RBMs.



Choosing the Number of Hidden Units

When we are trying to select the number of hidden units for RBMs, our main concern is avoiding overfitting. Given this constraint, our heuristic can be summed up by multiplying the visible units by 0.75 to get the number of hidden units. If our sparsity target is very small, we might get away with using more hidden units. For the cases in which the input dataset contains a large number of input records that are very similar, we can use fewer parameters, as well.

Using Different Units

Binary (or logistic) units in RBMs are typically where we begin with RBMs and DBNs. Depending on the use-case training setup for which the data might not be well-modeled by these units, we have other options to use for RBM hidden and visible units. For alternative situations in data modeling, we list more unit types:

- Multinomial visible units
- Gaussian visible units
- Binomial units
- ReLUs

Multinomial units are for topic modeling, building recommender systems, and building a classifier with a RBM (where to build the classifier we'd throw an output layer on top for classification after unsupervised pretraining). Compared to binary visible units, multinomial visible units replace the logistic function with the softmax function and use Bernoulli hidden units.

When dealing with images or speech, we find that binary visible units give us poor feature representation. In this scenario, we want to use Gaussian units. We want to keep using binary units for the hidden units, however, because using Gaussian units for both will result in worse instability in training.

We use Gaussian units in image and speech applications because logistic units tend to work poorly here. We need a smaller learning rate for these units compared to binary units. In some cases, these units have shown to be unstable during learning. When dealing with sparse data we want to use binomial units.

When we're dealing with continuous data, we use ReLUs. We see a similar parameter count as binary units as when using ReLU units, yet ReLU units are more expressive. ReLU units typically use a smaller learning rate than binary units for **better stability**.

during training. Table 7-5 provides a quick reference for looking up unit types indexed by input data.

Table 7-5. RBM unit types

Data type	Visible unit type	Hidden unit type
Text	Gaussian (Neural Word Embeddings), Binary (if Bag of Words)	Rectified Linear (Neural Word Embeddings), Binary (if Bag of Words)
Audio/time-series	Gaussian (if continuous), Binary (if 0–1)	Rectified Linear (if continuous), Binary (if 0–1)
Image/video	Gaussian (if zero mean and unit variance), Binary (if 0–1)	Rectified Linear (if zero mean unit variance), Binary (if 0–1)

Using Regularization with RBMs

Regularization is an important topic in machine learning because we want to control the size of the weights in our parameter vector. A great reason to use regularization with RBMs is that sometimes, early in training, weights with large values and the values of the associated hidden units are “stuck” on or off. Regularization helps the training process “unstick” these units.

With RBMs, we want to set a sparsity target for the activities of the binary hidden units. This is the desired probability of being active for the binary hidden unit. This factor is typically much less than 1.0.

When training RBMs, a good way to set the L2 prior weight-cost coefficient weight-decay is in the range of 0.01 to 0.00001. With RBMs, we don’t typically apply weight-cost to the hidden and visible biases because they are fewer in number and there is less of a chance of overfitting. In some cases, we’d like the biases to grow large so applying a weight-cost factor would make less sense here. For RBMs’ weight-cost, a great rule of thumb is to begin with a value of 0.0001. Small differences in the weight-cost probably will not change how your training process performs. It also has been shown¹⁹ where dropout has also been applied effectively to RBMs.

DBNs

DBNs involve a two-phase training process: pretrain and fine-tune. Pretraining involves first learning higher-level features from the input data to better initialize our feed-forward network with a good starting point for the fine-tune phase. A DBN uses stacks of RBMs to learn features during the pretrain phase. In Chapter 4, we gave an extended description of the interplay between DBNs and RBMs, refer back to that passage for more detail.

¹⁹ Srivastava et al. 2014. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.”

We want to initialize the weights in a DBN typically with small random values from a zero-mean Gaussian with a standard deviation of about 0.01. As the initial values become larger, we might see faster initial learning but can have the side effect of a **worse final model**. Hidden bias values of 0 are usually fine.

In training neural networks, and, more specifically, DBNs, we see learning rates in the range of 0.001 to 0.1. One side effect of having the learning rate too large is that the reconstruction error generally increases dramatically and the weights increase dramatically. This effect leads to poor models in most cases.



Setting the Learning Rate Visually

With DBNs, a good way to set the learning rate is to look at the histogram of the weight updates and a histogram of the weights. The weight updates should be about 10^{-3} times the weight values.

Using Momentum

Momentum can cause the learning process to move the parameters in a direction that is not the direction of steepest decent in some conditions. This allows the learning process to explore certain neighboring regions of search space that might have opposing gradients without feeling the full effect of the increment in gradient before turning back. It allows us to have smoother changes in learning direction over time, avoiding unstable oscillations.

We can tune Momentum for contrastive divergence used for training RBMs during pretraining. We like to begin with a Momentum setting of 0.5 and increase it to around 0.9 after the reconstruction error has settled down. If this creates too much of a shock in reconstruction error, reduce the learning rate by factors of two until it stabilizes.

When setting up Momentum at the beginning of training, a great place to start is the value 0.5, given that random initial parameter values can put us in odd starting places in search space. Over time, a good implementation will increase the momentum toward 0.9. As the reconstruction error rises or we near a terminating condition, the Momentum will begin to fall toward 0.0.

Using Regularization

In the previous section on regularization for RBMs, we've already covered the major reasons for using regularization (and this extends to pretraining DBNs).

We can view unsupervised pretraining with RBMs as a form of regularization. Unsupervised pretraining amounts to a constraint on the region in parameter space where our algorithm can scan. Evidence points to unsupervised pretraining as a data-dependent regularizer when there is enough neurons or parameters. However, unsupervised pretraining has also been shown to hurt generalization in some cases. We see an example of this when the training dataset size is relatively small (where small is less than 100,000 rows of data).



Sparsity and DBNs

Sometimes, looking at features that are only rarely active is a good way to improve model performance. We can set a sparsity target, which is the probability of binary hidden units being active (e.g., the sparsity setting is much less than 1.0). Research has shown good values for the sparsity target in RBMs to be between 0.01 and 0.1. The decay rate of the factor should be set between 0.9 and 0.99.

Dropout

When dropout is used for pretraining²⁰ we want to use a small learning rate with no weight constraints to avoid losing the feature detectors discovered by the pretraining technique. Dropout is also effective at the fine-tune phase of DBNs.



MNIST and Dropout

In a published account of using dropout on the MNIST dataset, a 50 percent dropout setting in the hidden layers gave performance improvement over using no dropout. Performing an additional 20 percent dropout on the input units gave further improvement. The settings of 20 percent on the input units and 50 percent on the hidden units was often found to be optimal for the dataset.

Determining Hidden Unit Count

The main question we face when considering the hidden unit count for DBNs is how many bits it would take to describe each input training vector (to produce a good model)?

²⁰ Hinton et al. 2012. “Improving neural networks by preventing co-adaption of feature detectors.”

Under this context, the main issue we face is overfitting, which we've discussed previously with regard to methods for control. To determine the number of hidden units over progressive layers, take the previous number and use a number of parameters an order of magnitude smaller.

CHAPTER 8

Vectorization

*New York City, Old St. Joe, Albuquerque, New Mexico
This old rig is humming and rolling and she's doing fine
If somebody wants to know what's become of this so and so
Tell em' I'm somewhere looking for the end of that long white line*

—Sturgill Simpson, *Long White Line*

Introduction to Vectorization in Machine Learning

This chapter is meant to serve as a set of guidelines for vectorizing different kinds of data used in the machine learning landscape. You might be wondering why we've taken a detour off into the land of vectorization in a book about deep learning. The main reason is that most machine learning books focus purely on the algorithms themselves and less so on the complete lifecycle of data mining. We want to experiment with data as fast as possible in our machine learning tools, and we end up spending far too much time on topics like custom vectorization of text data.

In our professional experience working with enterprise customers we've seen situations in which we'd talk about implementing text classification techniques only to have the exercise derailed by needing to diverge into a long discussion about the basics of converting text to vectors. Companies have a lot of simple data sources like spreadsheets that can be exported into comma-separated values (CSV) format yet still need to be transformed into vectors. We also find ourselves trying to explain the myriad ways textual data can be vectorized. Depending on the tools involved and the desired classification algorithm, the customer might not even have a good way to attempt text vectorization without extensive programming extraneous to the actual statistical modeling itself.



Raw Data and Automated Feature Learning

As discussed in [Chapter 3](#), that a key aspect of deep learning is moving from *feature engineering* toward *automated feature learning*. Automated feature learning is a useful aspect of deep learning, yet we still have to get the raw data into a form in which our tools can operate. Techniques of raw data vectorization allow us to do just that.

Over the years, it's become apparent that vectorization techniques (along with the data-handling process itself) is core to the process of data science and many times grossly overlooked. When putting together this book, we felt a strong need to give you a solid treatment of vectorization in a way that supported the deep learning modeling process and didn't distract with unwieldy side programming exercises for creating vectors. The other aspect is that as we move from the theory side of the book toward the practical side, we want to set up that transition with a focus on handling vectors in an approachable way, directly developing lots of great data to feed our deep learning models. We want to get you involved in modeling data as quickly as possible, and bridging this gap is a great way to kick that off.

Working with canonical machine learning datasets is a great example of how unwieldy this can become. The vectorization phase of machine learning can last from hours to days, depending on your comfort level with programming and vectorization. This impedance factor tends to slow down many new users beginning work with statistical models.

Why Do We Need to Vectorize Data?

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 numerical vector (or in some cases, a multidimensional array of numbers). Neural networks still need to represent the input data as vectors and matrices because they cannot work directly on text, graph, and other nonvector/matrix representations.

There are many different ways to handle the vectorization, and we can apply many preprocessing steps giving us different grades of effectiveness on the output models. Often, our modeling exercise is dependent on how well we vectorize our input data. Input data can come in many forms, including:

- Columnar CSV data
- Text documents
- Image data
- Audio data

- Video data
- Sequential data

In the following input data, we see an example of columnar CSV data in the [raw iris dataset](#) from the University of California Irvine repository:

```
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
```

An example of a text document might be a passage from *Go Dogs, Go!*:

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

Both cases involve raw data of different types, yet both need some level of vectorization to be of the form we need to do machine learning. We want our machine learning algorithm's input data to look more like the following serialized sparse vector format:

```
1.0 1:0.7500000000000001 2:0.4166666666666663 3:0.702127659574468
4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.4583333333333326 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 2:0.583333333333334 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 1:0.9166666666666667 2:0.6666666666666667 3:0.7659574468085107
4:0.5652173913043479
0.0 1:0.0833333333333343 2:0.583333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.958333333333335 2:0.7500000000000002 3:0.723404255319149
4:0.5217391304347826
0.0 2:0.7500000000000002
```

We can break down the process of converting raw data to machine learning-ready vectors into two phases:

1. Vectorization
2. Normalization

We cover each phase in the sections that follow. We'll also look at the concepts of data vectorization and normalization in three key areas of deep learning:

- Sequential data
- Image data
- Text data

We call out these data types specifically because of their relation to deep learning. We have shown in this book how Recurrent Neural Networks are adept at dealing with sequence data, so illustrating key sequential vectorization techniques is valuable. Deep learning has also shown to be valuable in the domain of image analysis with Convolutional Neural Networks (CNNs), so we also look at the subtopic of image vectorization. We close with a review of text vectorization methods and a look at how you can use Word2Vec in vectorization.



Deep Learning and the Changing Nature of Data Preparation

Deep learning has shown that we can do fewer of these specific feature engineering preprocessing steps. We still need to get the data into a vector form, but deep learning is adept at performing feature selection and dimensionality reduction in the process of learning structure in the dataset.

Many machine learning practitioners of the past decade have preferred to handcraft the features in their input vector dataset using expert domain knowledge. Expert domain knowledge is defined as a deep understanding of how the data was generated or how the sources of the data interoperate with the real world to drive certain effects in real life.

Later in the chapter, we'll see how a dataset with a small number of attributes might be a good candidate for handcrafting, but larger datasets such as text documents, images, or audio files need an algorithmic approach. In other cases, we allow a general vectorization technique to automatically generate the vectors, largely due to the complexity of the source data (large amounts of text) making the task difficult. Some examples of algorithmic vectorization are kernel hashing, Term Frequency–Inverse Document Frequency (TF-IDF), and Word2Vec.

Here are some major considerations when building models from raw data:

- What kind of source data are we dealing with?
- What type of model do we want to train with this data?
- What's our approach to vectorizing the data?

- Are we going to hand-encode the features or use an algorithmic approach?
- Raw text tends to be more cumbersome to deal with, so how do we handle it?

Based on the source data type, we need to take into consideration different things during vectorization. For example, columnar data (e.g., a database table) is relatively well known in terms of ways to handle vectorizing each column type. However, what would we do when trying to vectorize multivariate time-series data spread across multiple files? It's these types of questions that lie beneath the architectures of machine learning pipelines in today's applications, and in this chapter, we deal with the realities of building these pipelines. Let's begin with a review of how to deal with columnar data in the context of vectorization.

Extract, Transform, and Load

Many business intelligence (BI) and reporting tools in today's enterprise use a pre-processing phase called Extract, Transform, and Load (ETL). In the world of Apache Hadoop, this is common vernacular, as well. This is the phase in which we typically merge multiple datasets together, filter out unwanted columns, apply any necessary transforms, and then load the data into a place to be used by another application. Much of the vectorization process is performed during the ETL phase of a pipeline, and you need to take into account what actions you will need to perform to build out your pipeline.

Strategies for Dealing with Columnar Raw Data Attributes

Columnar source data—for example, an exported table from a relational database management system (RDBMS)—comes in a variety of shapes and sizes, but we consider each column in the table of data to be an “attribute” of the data. Each attribute of the dataset can further be classified as a type of attribute. Because we're working in the world of the data scientist, we'll use the definitions for columns or attributes of data based on that work. Statistical textbooks typically define four types of attributes in data:

- Nominal
- Ordinal
- Interval
- Ratio

Let's take a quick look at each type.

Nominal

Nominal attributes are sometimes referred to as *enumerated*, *categorical*, and/or *discrete* (e.g., “sunny,” “overcast,” and “rainy”). The term categorical means that these

attributes are set in a finite set of possibilities. They have values that are distinct symbols, and these values serve as labels. The term “nominal” comes from the latin term for “name.” Nominal attributes or labels have no relation to one another, nor is any order implied.

For nominal columns we recommend a one-hot representation of the column value. This will create multiple column entries in the feature vector with the column relating to the value from the source data containing the 1.0 value and the other columns containing a 0.0 value.

Table 8-1 shows example representations for two records for which the first record has a value of “sunny,” and the second record has a value of “rainy.”

Table 8-1. Example of one-hot representation in a feature vector

[Other columns]	Sunny	Overcast	Rainy	[Other columns]
...	1.0	0.0	0.0	...
...	0.0	0.0	1.0	...

Ordinal

Ordinal values are like nominal values except they have order. Ordinal values have rank, giving us a notion of ordering but no concept of distance between the values. We can compare ordinal values between one another, but mathematical operations don’t make sense in the context of these values.

Examples of ordinal values are “hot,” “mild,” and “cool” (which direction these are ordered doesn’t matter, as long as the convention is consistent). Other examples of ordinal values are “low,” “medium,” and “high.” These values end up being converted to integer values (e.g., cool = 0, mild = 1, etc.) but represented by floating-point values at the code level in vectorization.

The differences between ordinal and nominal are not always obvious, because there are subtle differences between them. Some of the older data-mining systems tended to cover only nominal and ordinal types. If the vector representation is for the output vector, we’d want to use a one-hot representation. For input vectors, we might want to convert the column value to a real-valued scale.

Interval

Interval values are ordered and measured in fixed and equal units. A great example of an interval measurement is a specific date or year. We can compare an interval, but adding or subtracting an interval doesn’t make sense. Interval values are already numeric, so we don’t need to do a conversion, yet we might end up applying methods of normalization.

Ratio

Ratio values are measured based on a zero-point and are treated as real numbers. In this context, math operations make sense. With ratio data the scheme defines a zero-point and then a distance from this fixed zero-point. Ratio values are already numeric, so they don't require a transformation. We might choose to apply a method of normalization, however, which we cover in the next section.

Feature Engineering and Normalization Techniques

Vectorization has been around for a while, and practitioners have developed patterns for building the canonical “flat” variety of vectors. Often we see these vectorization patterns applied in machine learning pipelines for methods such as logistic regression, random forests, and so on.

In the classical variant of vectorization, we create a fixed-size n -length vector (where $n - 1$ is our feature count, and the last slot is dedicated to the label value) and then set the values of each indexed cell according to some heuristic that we devise to represent our data. This is generally appropriate for a network such as multilayer perceptron (for which we're *not* using mini-batching). In many situations, in deep learning, however, we're building more complex n -dimensional matrix data structures for input to our deep networks. As we progress through this chapter, we'll see more advanced methods for creating input tensors for Recurrent Neural Networks and four-dimensional tensor input for CNNs. For the purposes of this section, we focus on how we engineer individual feature values in these input tensors and then how to apply methods of normalization to these values.

The process of vectorization is based on selecting attributes and finding a dimension in a feature vector to which to assign the feature. This is done in different ways depending on whether we're working with CSV, text, image, or time-series data. We still need to select which parts of the input data we want to work with and collect that in the input vector. We can undertake the process of taking n attributes and turning them into m features in our output vector in a number of ways, which is referred to as *feature engineering*. Raw source data typically needs transformation before it's ready for modeling. Raw source data can take on a range of manifestations:

- Raw text such as a document in a text file
- A file containing a tweet string per line
- Binary time-series data in a custom file format
- Preprocessed datasets with a mixture of numeric and string attributes
- Image files
- Audio files

Depending on the condition and source of the raw data, attributes can be in numeric or string form. For most cases, the values of attributes are numeric and continuous.

These attributes measure numbers, either real or integer values. The term “continuous” is conflated in meaning in this setting.

Data Cleaning and ETL

Data cleaning is a pragmatic step here and takes a good chunk of time for most data scientists. Data cleaning tends to occur during the ETL phase of machine learning and is typically done in the ETL pipeline with scripts or Hadoop jobs. Depending on the type of attribute and the specific semantics of the scenario, we'll want to use placeholder values. Every column in the dataset having a value does not completely absolve us of examining the data further. We can also encounter inaccurate values due to a variety of reasons. Many times, the data was gathered over time and some column's values were not relevant and received a dummy value potentially being inaccurate. Again, studying the statistics of the dataset and graphing attributes are good best practices.

We can apply the techniques of feature engineering in the hand-coded vectorization methods of vectorizing columnar data or when we're working with more complex data such as time-series or image data. These techniques include the following:

- Taking the values directly from the attribute unchanged
- Normalizing an attribute to create a feature
- Binarization of features
- Dimensionality reduction

We provide some details below on these techniques.

Feature copying

To create vectors from this raw input data, we need to select the proper “features” of the data that we think are most relevant to the model. Feature selection has long been held as a key to building a successful model. The number of features in our produced vector typically will not match up to the number of attributes in the source data. Many times the source data will be joined together with other datasets and then a subset of attributes of the resultant denormalized view of the dataset will be used to derive the final set of features in our vector.

The most common way to generate a feature is to simply copy an attribute that is already numeric and in the correct range. This scenario is, alas, not very common. More often, we need to do some basic transformation on the attribute.

Dealing with Missing Values

Many times with data, we encounter missing values in our raw input data that are frequently indicated by out-of-range data entries (possibly -1 in a numeric entry or a 0 in a numeric attribute that can never hold a zero-value). For nominal attributes, missing values are commonly represented by blanks or dashes. Missing values can occur for many reasons and understanding the mechanics of the source data can help us triage why values are missing. A good practice is to study the statistics and properties of the dataset before we begin building vectors by looking for outliers. A simple method for this is to graph the various attributes and look for things that don't look right; for example, year attributes that have a 0 value.

Here are some basic methods to deal with missing values:

- Filter out records with missing values (caveat: can introduce bias if values are not missing at random!)
- Set to 0 if missing (caveat: can be OK sometimes; makes the data “noisy”)
- Set to most common value for the column (there are other variants of deciding this replacement value)

Normalization

Normalization methods scale the input training data to be in a range—for example, $[0, 1]$, $[-1, 1]$, and so on—after we've converted the raw data into a vector representation. We care about normalization because it can affect the activations in a neural network. If the inputs to a network are too large, we see activations that are too large, potentially adversely affecting training. Conversely, if our inputs are too small, our activations might end up being too small, as well. Calculated gradients in the network can be affected in a similar fashion.



Some Hyperparameters Assume Normalization on Data

We make other design decisions such as weight initialization schemes (e.g., Xavier weight initialization) in the context assuming that normalization will be applied to the input data.

In the domain of normalization we want to transform the data so it is more consistent. There are **two basic tools** we can combine in different ways to perform normalization:

- Centering
- Scaling

Each of these methods are focused on making the training process easier. With centering, we are transforming the feature so that it is centered on the origin, or, in most cases, the mean. Scaling is a method by which we rescale the feature such that it has a variance of 1 across the entire dataset, or each feature has a maximum absolute value of 1 across the training dataset.

Here are the specific methods of normalization that use the above core techniques:

- Standardization
- Min-max scaling
- Whitening
- Principal Component Analysis (PCA)

We see both centering and scaling applied in common normalization practice standardization to transform the data to have a zero mean value and unit variance. Min-max scaling is considered a variant of the basic scaling technique, and can help a learning algorithm operate more efficiently. We'll touch briefly on whitening and principle component analysis later in this section.



Drawbacks to Normalization

Many datasets we see in practice have outliers. When we normalize a dataset we scale the nonoutlier data to a small interval, and this is something for which you should be vigilant.

Quick Review: Means and Variances

Often, we'll use statistical measures in the transformation of our vectors. These include using the sample mean (e.g., just "mean") or the sample variance (or just "variance") of a feature.

The mean of a feature (or column in dataset) is just the average of the values for the specific feature. For N real-valued numbers x_1, x_2, \dots, x_N , we calculate the mean as follows:

$$\mu = \frac{1}{N} \sum_n x_n$$

The variance of a feature measures how much the values for a feature vary around their mean:

$$\sigma^2 = \frac{1}{N-1} \sum_n (x_n - \mu)^2$$

where μ is the sample mean, as just defined.

Standardization and zero mean, unit variance. When we “standardize”¹ a feature column value, we subtract a measure of location (minimum, maximum, median, etc.) and then divide by a measure of scale (variance, standard deviation, range, etc.). We want to rescale features such that they will have properties of a standard normal distribution with the following:

$$\mu = 0 \text{ (the mean equals 0)}$$

$$\sigma = 1 \text{ (standard deviation equals 1)}$$

For each feature, we want to transform the value to have zero mean and a standard deviation from the mean of 1.0. To do this, first we need to calculate the mean and standard deviation for each feature. The mean is then subtracted from each feature and the result is divided by the standard deviation of the feature, as seen here:

$$z = \frac{x - \mu}{\sigma}$$

This is also referred to as *zero mean, unit variance*. This equation will center the feature value on 0 with a standard deviation of 1.0, producing values in the range of $[-1, 1]$.



Zero mean, unit variance is the most commonly used method for standardization. We typically see it used with dense vectors in neural networks.

Optimization methods such as Stochastic Gradient Descent (SGD), can greatly benefit from methods such as standardization. Many times when we see features on different scales, we also see some parameters being updated faster than others because the input features are on different scales. Standardization also helps methods that compare similarities between features based on a distance metric.

As we transform data fields from attributes to features in our vector, we also need to consider how sparse (i.e., lots of zeros in our vectors) the vector is. When we have sparse data, we want to scale features (or “normalize” as previously mentioned) to fall between (0, 1) to represent probabilities. In contrast, when we have dense vectors we want to first perform a preprocessing step to zero the mean and unit variance before we complete the step to scale resulting values (now in the range of $[-1, 1]$).

¹ In some literature, standardization is referred to as *z-score normalization*.

The primary difference here between sparse and dense data preprocessing is that with dense data, we first do zero unit/mean operations and then do scaling, whereas with sparse, we just do scaling. Obviously, for the cases in which our algorithm needs the data in the range of (0, 1), we'd want to scale it to that range (example: RBMs).



Standardize with Caution

Naively applying standardization to a sparse vector might not be a good idea. Min-max (described in the next section) *might* be better in some cases. With sparse, we usually want to keep 0 as 0 post-normalization, whereas standardization might not do that.



A Broader Representation with Standardization

The range of [-1, 1] gives a broader representation of our data than [0, 1] in basic normalization due to there being more bits of information in the floating-point representation.

Min-max scaling. Another way to normalize data is to use min-max scaling, in which we scale each feature to a fixed range (e.g., the range [0, 1] is common here). The most basic min-max scaling is performed with the following equation:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

The trade-off we're making with scaling compared with standardization is that we end up with smaller standard deviations. Min-max normalization is more sensitive to outliers, because a single outlier (small or large) affects the min-max values and can have a big effect on the normalization. This ends up being less of an issue with standardization.

Min-max scaling tends to be popular in image processing for which pixel intensities need to be normalized into a specific range (e.g., RGB values are [0, 255] and need to be scaled to [0.0, 1.0] for input vectors).



Conflating Terms in Normalization

Some texts call min-max scaling simply “normalization” and consider “standardization” as another technique. In this book, we’re calling them both methods of normalization and delineating them specifically as min-max scaling and standardization, respectively. This seems to be a more discrete and specific way to express the methods.

Whitening and principal component analysis. A transform of note in the context of deep learning is called statistical whitening² (whitening, for short). This is the process by which we transform data such that it has an **identity covariance matrix**. Whitening also decorrelates the data, helping to find an optimal way to represent the data in an efficient way. The whitening transform is named due to how it changes the input data into a white-noise vector. A known issue with using whitening is that it can exaggerate the noise in our data in certain situations.

Principal component analysis³ is a method to convert data in a dataset (which is possibly correlated) into a dataset of linearly uncorrelated variables. This method uses an orthogonal transformation to perform the transformation and produces a set of principal components.

Dimensionality Reduction

Dimensionality reduction comes up in machine learning literature quite often for situations in which we'd like to find the attributes that matter most to the model in some algorithmic fashion. An interesting dynamic with deep learning and more specifically Deep Belief Networks (DBNs) is that the pretrain phase learns representations of the data by training layers of RBMs. We also see autoencoders encoding raw data into shorter vectors of numbers. This smaller representation then can be used in classification or similarity search algorithms as input.

Applying normalization in Recurrent Neural Networks and CNNs. When our model assumes some structure in the input, our normalization should, as well. When we're working with Recurrent Neural Network variants—for example, Long Short-Term Memory (LSTM)—we want to consider all time-steps when we calculate the mean, standard deviation, minimum, or maximum values for a given input.

When we're working with normalization in the context of CNNs, we want to share mean, standard deviation, and other calculated values across all pixel locations in the image, usually separately for each channel (RGB).

Normalization for regression models. Although we need to use normalization of data for regression, we need to consider regression a special case. In normalization in a classification model, we usually always normalize the features (input); however, for a regression we also often need to normalize our labels (e.g., targets or output data values).

² Kessy, Lewin, and Strimmer. 2015. “Optimal whitening and decorrelation.”

³ Pearson. 1901. “On Lines and Planes of Closest Fit to Systems of Points in Space.”

For regression modeling, the same basic principles apply in the process of normalization. We still want to use methods such as min-max scaling and standardization when we normalize the inputs and outputs for the training process.

Why Do We Normalize the Outputs for Regression?

Consider, for example, the mean squared error (MSE) loss function: if we have values in the range 0 to 1 million, we end up with large errors and much too large gradients.

However, after normalizing the labels (output values), we are no longer predicting the original thing we care about, so we need to undo the normalization for the network predictions. We can perform reverse normalization with min-max and standardization because they are one-to-one mappings.

If we rearrange the normalization equations to undo it:

```
(reverse standardization)  
origScaleOutput = netOutput * sigma + mu  
(reverse min-max scaling)  
origScaleOutput = netOutput * (xMax-xMin) + xMin
```

Binarization

There are other situations when building features from attributes for which the modeling algorithm might require that the data is distributed according to a multivariate Bernoulli distribution.

In this case, we could use *feature binarization* to threshold the numerical features to derive Boolean values. This involves using a filter to produce a feature that includes a 1 or a 0 as its value.

Using DataVec for ETL and Vectorization

As we've established so far in this chapter, the topics of data ETL, vectorization, and normalization are all important topics for the practitioner. We need to consider not only what kind of machine learning pipeline to build, but also how to operationalize that pipeline. Given that most joins and ETL operations are single passes over the datasets, horizontal scalability also comes into play when we want our pipeline to operate faster.



Data Doesn't Need to Be Big to Need More Speed

Even if you don't have terabytes of input data, this doesn't mean that you won't need to consider a scalable solution for the ETL phase of your machine learning pipeline. Many times management or the customer asks, "How can we get this answer faster?" and many times the answer is to scale the pipeline horizontally. Tools such as Apache Hadoop, Apache Spark, DL4J, and DataVec are all part of the answer to modern, scalable machine learning pipelines.

DataVec allows for local mode execution and also the ability to scale horizontally on Apache Spark and Apache Hadoop. In the examples in [Chapter 5](#), there are many in which we used the ND4J API directly to manipulate data to produce the input DataSet objects used by the DL4J library. One of the nice things about DataVec is that it is able to take on certain types of standard data types and automatically produce vectorized DataSet objects. DataVec also includes transforms to collect statistics on the vectorized data and then normalize the data in the DataSet object.

As discussed previously in this chapter, the fundamental case for vectorization in machine learning (and deep learning) is to be able to handle the columnar (CSV) data type. To do this in local mode with DataVec, we'd simply write the following:

```
RecordReader reader = new CSVRecordReader( numLinesToSkip, delimiter );
InputSplit inputSplit = new FileInputSplit( file );
reader.initialize( inputSplit );

//Create the DataSetIterator. We are assuming classification here:

int minibatchSize = 10;           //Number of examples in each minibatch

int labelIndex = 7;              //Index of column that contains the label
int numClasses = 5;              //Number of classes (label categories)
DataSetIterator iterator =
    new RecordReaderDataSetIterator(
        reader, minibatchSize, labelIndex, numClasses );
```

We saw this code initially in [Chapter 5](#) when we built the multilayer perceptron model with the CSV data input. DataVec allows us to specify different types of record readers, couple them with DataSet iterators, and work with the data in a DL4J modeling pipeline. DataVec gives us the ability to quickly specify operations for the following:

- Joins
- Filters
- Normalization
- Standardization

As we progress further through this chapter, we'll discuss the other major data types and lay out notes on how DataVec handles them.



To read more about DataVec, see [Appendix F](#) by guest author Alex Black.

Vectorizing Image Data

Images are another rich source of information that we can mine for information. We define an image as “an artifact that depicts or records visual perception; for example, a two-dimensional picture.” An array of pixel values, as we see in [Table 8-2](#) represent an image stored on a computer. Each pixel has a value indicating how bright the pixel should be and/or what color the pixel should be.

Table 8-2. Pixels in an image visualized as rows and columns

	Column-1	Column-2	...	Column-m
Row-1	p11	p21	...	p-m-1
Row-2	p12	p22	...	p-m-2
...
Row-n	p-1-n	p-2-n	...	p-m-n

The number of bits per pixel dictates how many colors we can show per pixel. A 1-bit pixel can display only binary images (typically black and white pixels). Eight-bit pixels are more common and can display 256 colors (or grayscale) per pixel. For grayscale images, the integer representing the pixel indicates the brightness (closer to white or 255, from black or 0) of the pixel. To represent color images, separate red, green, and blue components must be specified for each pixel (assuming an RGB colorspace), and so the pixel value is actually a tuple of three numbers. Common image file formats are JPG, PNG, and GIF.



Image Formats

Each format can store the image data differently and perform varying levels of compression on the raw data. Images are different from text but can be more straightforward in how we vectorize them.



Working with Video Data

Video data vectorization is a variation of both image vectorization and time-series vectorization. With video data, we have a series of timestamped images. Our video vectorization process involves tracking image vectorization over time and can become more complicated in terms of what we consider an individual vector (e.g., single frame versus subset of image across multiple frames).

Image Data Representation in DL4J

In the context of deep learning, we're most interested in image vectorization for CNNs. In DL4J, we want to convert each image into a three-dimensional tensor representation in an `INDArray` object (for more on this topic, see the [Appendix E](#)) as seen in [Figure 8-1](#).

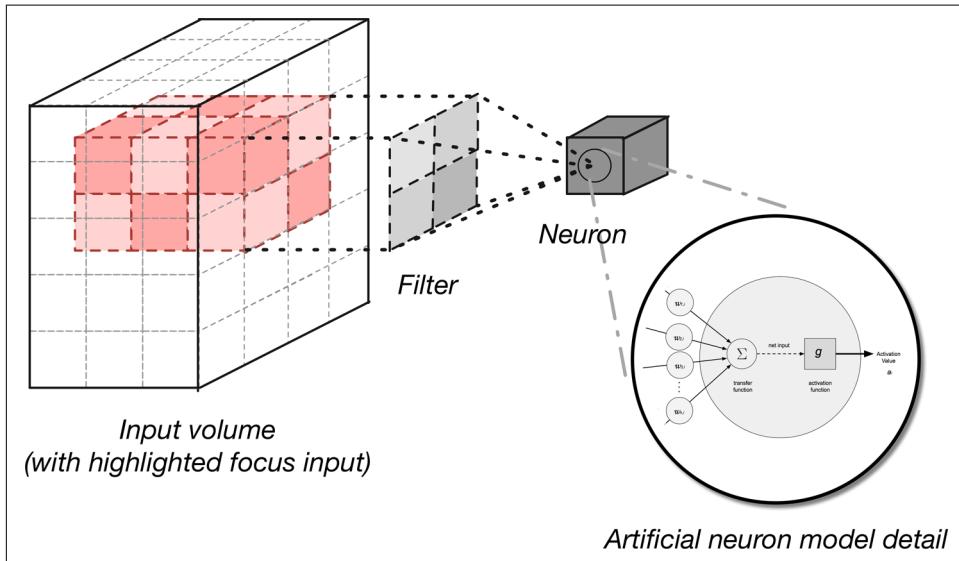


Figure 8-1. Three-dimensional input volumes in convolutional networks

When we create mini-batches of images for training, as we've seen in examples in this book, we're actually creating four-dimensional tensors that have the following dimensions:

1. Mini-batch size
2. Depth
3. Height
4. Width

Where the width and the height map directly to the input image characteristics and the depth is the number of channels (e.g., RGB is typically 3) in the image data.



4D Mini-Batches of Images are the Standard in CNNs

The 4D format is the standard in DL4J for working with CNNs and images.

In the event we want to vectorize data for a multilayer perceptron network, we need to flatten it into two dimensions (e.g., `minibatchSize`, $\text{depth} \times \text{height} \times \text{width}$). To understand this variant of image vectorization, let's continue with our earlier visual grid explanation. With an image, the first thing we need to do is extract the pixel intensities from each location in the array of pixels. The basic idea is to take the concept of an $M \times N$ image and flatten this rectangle into a $1 \times (m \times n)$ array as seen in [Table 8-3](#).

Table 8-3. Flattened image data

	Column-1	Column-2	...	Column- $m \times n$
Row-1	pixel1	pixel2	...	pixel- $m \times n$

The image is composed of a sequence of floating-point values that indicates hue intensity physically, but we tend to think of them as their logical form of the $M \times N$ grid of pixel intensities. To feed this image into any linear algebra vector, we need to arrange these pixel intensities into a vector of values more suitable for machine learning algorithms.

We need to convert this rectangle into a $1 \times (N \times M)$ array per image record in our matrix, as seen in [Table 8-4](#). We perform this flattening routine by using `CnnToFeedForwardPreprocessor`, which is done automatically by using `.setInputType(InputType.convolutional(height, width, depth))` in the configuration of the network. The method `.setInputType()` will realize we're trying to feed a 4D array into a dense layer that requires 2D, and it will add the preprocessor to flatten it for us.

Table 8-4. Flattened mini-batches of images in 2D array

	Column-1	Column-2	...	Column- $m \times n$
Image1	pixel1	pixel2	...	pixel- $m \times n$
Image2	pixel1	pixel2	...	pixel- $m \times n$
Image3	pixel1	pixel2	...	pixel- $m \times n$

Image Data and Vector Normalization with DataVec

A major part of the vectorization in image modeling is getting the raw pixel data out of the file format container and translated into the vector object that our learning algorithms understand. We need to read the raw pixel data, cell by cell, do any needed transforms or normalizations, and then add the value at the corresponding index in the output vector. Many times we simply take the entire set of image pixels and translate them directly into a vector with some transforms per pixel. In more advanced methods of image processing, sections of the image are parsed out and used as vectors by themselves.

DataVec's `ImageRecordReader` is the record reader to use with image data and performs common image operations such as cropping. `ImageRecordReader` supports a wide variety of image formats, including the following:

- JPG
- GIF
- PNG
- TIFF
- BMP

Efficient loading and image operations are provided by JavaCV and OpenCV. The code snippet that follows shows DataVec's `ImageRecordReader` being set up to build a `DataSetIterator`, which allows DL4J to train directly on image data:

```
ImageRecordReader reader =
    new ImageRecordReader(
        outputHeight, outputWidth, inputNumChannels, labelMaker );
reader.initialize( inputSplit );

//Finally, create the DataSetIterator:

int minibatchSize = 10;           //Number of examples in each minibatch
int labelIndex = 1;              //Always value 1 for ImageRecordReader
int numClasses = 3;             //Number of classes (label categories)

DataSetIterator iterator =
    new RecordReaderDataSetIterator(
        reader, minibatchSize, labelIndex, numClasses );
```

Functionality such as this makes DL4J a much easier platform to work with because you can focus on the overall ETL and vectorization pipeline flow and less on the minute details such as extracting pixel values from file formats. We cover the `ImageRecordReader` in further detail in [Appendix F](#), along with methods for image data transforms, normalization, and standardization.

Working with Sequential Data in Vectorization

Sequential data is similar to columnar data except there can be multiple values for a given column in a row. An example of sequential data would be a temperature reader at a specific location taken periodically over time. If we were to add a timestamp to each of these temperature measurements, we could term this sequential data as “time-series data.” Time-series data is defined as a sequence of data points measured typically at successive time intervals. In the world of finance, we see time-series data⁴ in the form of the daily-adjusted close price of a stock on the NYSE. Another example would be sensor readings from phasor measurement unit (PMU) devices on the **smart grid** that measure “phase angle” and voltage 30 times a second.

One of the most common applications is log processing. Systems such as web servers, cell phones, and credit card readers all generate log entries every time an action is performed in response to a request. These devices are a tremendous source of sequential and time-series data. This data doesn’t always fit that well in a traditional RDBMS system, so we often see it stored on storage area network (SAN) drives or Apache HDFS. The much talked-about Internet of Things (IoT) largely consists of these use cases in which we are ingesting and processing time-series data from sensors (webservers, PMUs, cell phones, etc.). Other sources of sequential data we might not think of directly are genomic data and regular character sequences (e.g., sentences).

Many machine learning tools today have poor support for sequential data and the practitioner ends up having to handcode complex ETL pipelines to handle this class of data. We also have a lot of sequential values that can be very noisy and we need normalization and standardization. In the rest of this section, we look at some practical ways to deal with the ETL and vectorization requirements of sequential data with DataVec.

Major Variations of Sequential Data Sources

Sequential data can come in many forms based on who or what device produced the data, where the data is stored, and what format was used to store the data. Here are three major ways we often see sequential data:

- Raw log lines
- Single CSV file of time-series data
- Multiple CSV Files of time-series data

⁴ For a more detailed look at time-series in the wild and in practice, go to <http://www.cloudera.com/blog/2011/03/simple-moving-average-secondary-sort-and-mapreduce-part-1/>.

Inside of these source files, we might see different arrangements of data. We have entities (e.g., a “user,” or other agent for which we’re tracking actions) and then we have N number of values we’re collecting per sequence entry. So, we’ll have a sequence of 1 or more columns for every entity. Some of these variations in how data is stored on disk include the following:

All sequences in a single file

Each line is a CSV set of readings (e.g., multiple columns) from a single source.

Sequences spread across multiple files

Each file in this variant tends to represent readings from a single source.

In the single-file variant, sometimes we see sequential data arranged in a row-oriented format in a CSV file where each line represents a single column for a given source. The line also contains the source’s identifier.

When each source is generating multiple columns of time-series at each step, we’ll often see the data arranged in CSV files, in which each line is a time-step. Each file represents a single source of the multivariate time-series data. Each row will typically have one or more columns to represent the multiple values per time-step for the source.

We should also consider how many columns are measured at each step in the sequence of time-series data. For example, when we’re measuring only voltage on a power grid, we’re only measuring a single column of data per step in the sequence. If for every step (or time-step) we’re measuring both voltage and temperature, we’d need a file format with the ability to represent at least two columns per sequence step.

All of these factors (and more) come into play in the ETL process leading up to the vectorization, normalization, and standardization of sequential data. Eventually we’ll want to get all of this information into an object (e.g., a `DataSet` object) that our modeling tools can understand. DL4J provides support for handling sequential data ETL and through DataVec.

Vectorizing Sequential Data with DataVec

We need to be able to load the raw sequence file from a drive (local, or from another location such as HDFS). This involves such tasks as:

- Reading the file format
- Processing any joins or transformations of the data
- Matching up the labels (if any) to the features for the relevant sources of sequences
- Performing any normalization or standardization needed
- Finally producing a `DataSet` object with data aligned correctly

Ideally, our input file format matches up to a record reader we already have in DataVec (or Spark, if we're using that execution environment). Ultimately, we want to construct a mini-batched set of sequential data with one or many columns in a tensor structure represented by an NDArray in ND4J. We want a 3D tensor structure, such as that depicted on the right side of [Figure 8-2](#).

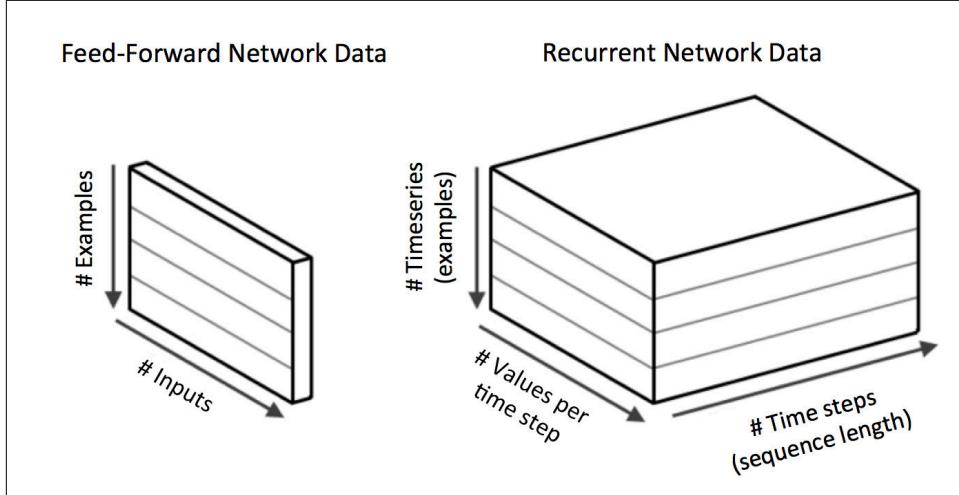


Figure 8-2. Representing sequential data for Recurrent Neural Networks

We want to create a 3D tensor (e.g., 3D matrix) for which the following is true:

- Mini-batch size is the first dimension
- Feature columns are the second dimension
- Time-step values for each column are the third dimension

It can become tricky to perform all of the aforementioned ETL work and then align the data values in the NDArray tensor by hand. From that perspective, we recommend that you use record readers and formats already in DataVec where possible.



Custom File Formats

If you do not have a record reader in DataVec for the particular type of input data, you will have to write some custom code for tensor construction. This can be a nontrivial exercise depending on the complexity of the data.

Let's take a look at some older variants of time-series vectorization before we move on to how we do a few patterns of sequence conversion into tensors with DataVec.

Converting time-series to a single vector

For most machine learning methods, we focus on representing each record as a vector or row in a matrix. Each row represents a single record, and a group of rows represents a mini-batch of these vectors to train on, as illustrated on the lefthand side of [Figure 8-2](#).

The major drawback is losing the time aspect of the input data. Most of the time, before Recurrent Neural Networks, we didn't care because most machine learning techniques did not have a way to take the sequence of values (or the time domain) into account anyway. Most of the time the practitioner was tasked with performing exotic methods of vector creation through hand-engineering. Although in some cases this is effective, as discussed in [Chapter 5](#), this path can lead to significant technical debt in terms of durability in the machine learning pipeline.

Symbolic Aggregate Approximation

There are a few ways to handle time-series data in the world of flat vector representation. One method to deal with noisy time-series data is to run a low-pass filter over it as a preprocessing step. [Symbolic Aggregated approXimation \(SAX\)](#) preprocessing is a great way to reduce noise in time-series data. Dr. Eamonn Keogh's team at the University of California, Riverside, developed SAX. SAX is a symbolic representation of times-series with some unique properties. SAX represents a time-series T of length n in d -dimensional space. It is essentially a low-pass filter with a lower bounding of the Euclidean distance. SAX performs a measure of dimensionality reduction on the raw time-series data, which helps learning algorithms.

The successor to SAX is indexable Symbolic Aggregate approXimation (iSAX), which adds some additional capabilities. iSAX modifies SAX to facilitate “extensible hashing” and a multiresolution representation and allows for both fast exact search and ultrafast approximate search. The net effect is that iSAX’s representations are more flexible in representing different distributions of values and it allows for an interesting index over the time-series that is handy in many applications like fast approximate search.

Converting sequential data to a DataSet object in local mode

In the world of Recurrent Neural Networks (and LSTMs), we want to produce a 3D tensor represented by a `DataSet` object. To illustrate this concept, let's consider the case in which we're modeling data that has each entity's sequence data separated into a distinct file. There is also a corresponding file for the label for each entity, as well. For example, we'd have the following:

- The file for the sequence data for entity 0:
 - *train/features/0.csv*
- Label for sequence data entity 0:
 - *train/labels/0.csv*

This data has only one column of data per measured step, so we consider it univariate. With DataVec, we can use the `CSVSequenceRecordReader` as our record reader, and then the `NumberedFileInputSplit` class will handle the file and directory naming schemes for us. We use the `SequenceRecordReaderDataSetIterator` class to produce our `DataSet` objects, which will use the record reader and input split objects, as shown in the following code example:

```
//Note that we have 450 training files for features: train/features/0.csv
//through train/features/449.csv

SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
    .getAbsolutePath() + "%d.csv", 0, 449));

SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
    .getAbsolutePath() + "%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;

DataSetIterator trainData =
    new SequenceRecordReaderDataSetIterator(trainFeatures,
    trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

//Normalize the training data
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData);           //Collect training data statistics
trainData.reset();

//Use previously collected statistics to normalize on-the-fly.
//Each DataSet returned by 'trainData' iterator will be normalized
trainData.setPreProcessor(normalizer);
```

Here we've put together the aforementioned objects to create a `DataSetIterator`, which will give us mini-batches of `DataSet` objects. These mini-batches of `DataSet` objects will contain the data from the source files (and label files) aligned correctly. Toward the bottom of the code, we also can see how normalization and standardization might be applied to the data while in flight in the DataVec pipeline.

Here are other variants of the record reader in DataVec:

CSVNLinesSequenceRecordReader

This is another version of the previously mentioned CSV sequence reader. Each sequence is exactly N lines long, all in one file, one after another.

RegexSequenceRecordReader

This works well for log data for which we need regular expressions to parse the columns.

In terms of dealing with file formats, earlier we demonstrated the use of the `NumberedFileInputSplit`, which understands how to read numbered files split out in directories. Obviously, there are many other ways in which files can be laid out on a drive, and we'd want to subclass the `FileSplit` class to further deal with those scenarios.

Another topic to consider here is building custom `DataSets` from sequential data for which we don't already have the needed built-in functionality in DataVec and then running DataVec on Spark. We cover the custom construction of `DataSet` objects in the next section. We cover vectorization of sequential data on Apache Spark in [Chapter 9](#).

Building custom `DataSets` from sequential data

The easiest way to think about how to create the needed data structures for training an LSTM in DL4J is to realize we need only the following:

- An input `NDArray` holding the training data
- A label `NDArray` holding the label data
- Two masking `DataSets` representing our masking information

We combine these four `NDArray` objects together to construct the needed `DataSet` object DL4J expects:

```
DataSet d = new DataSet( input, labels, mask_in, mask_labels );
```

Really, the trick is in laying the data in each of the `NDArrays` in the correct spot before we put them all together in a fashion similar to the preceding line. An example of this might look like the code snippet here:

```
//Allocate space: { mini-batch size, number columns, number timesteps }
INDArray input =
    Nd4j.zeros(new int[]{ miniBatchSize, inputColumnCount, maxTimestepLength });
INDArray labels =
    Nd4j.zeros(new int[]{ miniBatchSize, outputColumnCount, maxTimestepLength });
INDArray mask = Nd4j.zeros(new int[]{ miniBatchSize, maxTimestepLength });

for (int miniBatchIndex = 0; miniBatchIndex < miniBatchSize; miniBatchIndex++) {

    for (int curTimestep = 0; curTimestep < endTimestep; curTimestep++) {
```

```

    //      input -> set the column index for the character id-index ->
    //at the current timestep (c)
    input.putScalar(new int[]{ miniBatchIndex, columnIndex, curTimestep }, 1.0);

    // do more column input here ....

    // now setup the masks, setting a 1.0 for the timestep wherever there is data
    mask.putScalar(new int[]{ miniBatchIndex, curTimestep }, 1.0);

    //      labels -> set the column index for the next character id-index ->
    //at the current timestep (c)
    labels.putScalar(new int[]{ miniBatchIndex, nextValue, timestep }, 1.0);
}

}

INDArray mask2 = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
Nd4j.copy(mask, mask2);
return new DataSet(input, labels, mask, mask2);

```

Obviously, by the time we'd execute this code, we'd have performed all of the ETL and vectorization needed for the raw data. At this point, we're just aligning the data in the tensor data structure. Here are the main things to note about this code:

- We need to use ND4J to construct our input matrices.
- The two-loop pattern is generally a coherent way to walk through the 3D data structure and set up our data.
- We then mostly make calls to `INDArray.setScalar()` to set values.

The trick we'll note here is how we lay out the label matrix data. If we're doing character prediction, as in the Shakespeare example from [Chapter 5](#), we're putting the next character value at the current time-step in the labels matrix. If we're doing a sequential data classification model (e.g., classifying an anomaly in log data, for example), we might put the class at every single time-step in the labels matrix. How we lay out the data is a bit of an art and there are multiple strategies for treating data that occurs at uneven time-steps. The strategy for aligning time-steps is a decision we leave to your discretion.

Using Masking in Tensors

In DL 4J, we use masks for both training data and labels.

We set the mask time-step value to 1.0 for every time-step containing training data in the training data mask. We set the mask time-step value to 0.0 for all other time-steps. We normally use the same mask for both the input data structure and the label data structure.

Typically, we see this in code where we're looping over the tensor data structure and we set the mask entry for each time-step as we go along. This might look like the following:

```
INDArray mask = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
...
for (...) {
    ...
    mask.putScalar(new int[]{ miniBatchIndex, timeStep }, 1.0);
```

Notice that we're setting only a mini-batch index and then a time-step index in the mask data structure. We don't need to specify which columns have data, just that any column in this specific time-step could have data.

Working with Text in Vectorization

Text can be unwieldy at first glance, given that we can have any number of words in a document or passage and the number of words across a corpus will not be the same. If each document doesn't have a consistent "attribute" count, the simplest method of feature engineering (simply copying the attribute value over) will not work. We need methods to convert a variable number of attributes to a consistent number of features. As we'll see momentarily, there are a number of ways to do this. In this section, we review some of the older methods of text vectorization and then compare those to more modern methods such as Word2Vec. Let's begin by looking at the fundamentals of free text and the [Vector Space Model \(VSM\)](#).

The VSM is a common way of vectorizing text documents. In this model, every possible word is mapped to a specific integer. If we have a large enough array, every word fits into a unique slot in the array and the value at that index is the number of the times the word occurs. Most often our array size is less than our corpus vocabulary, so we have to have a vectorization strategy to account for this.

To model text we go through several stages:

1. Sentence Segmentation: We can skip directly to tokenization depending on use case.
2. Tokenization: We use this to find individual words.
3. [Stemming](#): This finds the base or stem of words (optional).
4. Lemmatization:⁵ This collapses the different inflectional variants of a lemma (optional).
5. Removing stop-words (optional).

⁵ In some languages stemming is “light” version of lemmatization.

6. Vectorization: We take the output of the process and make an array of floating point values.

With the VSM, we assume that words are dimensions and by extension are orthogonal to one another. This is used in a similar fashion to how a point's x and y values would be considered independent. But for text this is actually not true, because words can have co-occurrence mechanics. An example of this would be the name of a product or a team such as the "Tennessee Volunteers." The probability of "Tennessee" and "Volunteers" occurring together is higher so these words are not truly independent. For vectorization, we can use multiple strategies. In the following sections, we list a few of the most common methods in the text vectorization space in order of simplicity, from most simple to more advanced.

Bag of Words

Bag of words is a list of words and their word counts. It is the simplest vector model but can end up using a lot of columns due to the number of words involved. Typically we want to normalize the counts of the words per document to help our learning algorithm, which give us probabilities for word occurrences in the document in the vector representing the document. The bag-of-words model gives us a simplified representation and is often used in Natural Language Processing (NLP) and information retrieval (IR).

A group of words or a document is represented as a bag, or multiset, of its words. Grammar and word ordering is ignored, but we still track how many times the word occurs in the document. The bag-of-words vectorization technique has been used most frequently in the document classification and IR domains.⁶

In [Table 8-5](#), our vector has an index for every distinct word in the document. If the word "apple" corresponds to the 0 index, every time the word "apple" occurs in the document, we increment the value at this index. This ends up becoming a word-counting exercise in which we define a distinct set of words and then count how many times they occur. When we have this distinct word count, we can build our feature vector based on how the words' index maps into the vector. In this vector form, we maintain the number of times a word occurs, but we lose the word-ordering information. We also can consider this representation to be equivalent to the histogram representation.

⁶ You can find one of the earliest references of the bag-of-words model in Zellig Harris's 1954 article on "Distributional Structure." (Orderless document representation: frequencies of words from a dictionary by Salton & McGill also appears in 1983.)

Table 8-5. Illustrating the vector space model visually

	T ₁	T ₂	...	T _t
D ₁	w ₁₁	w ₂₁	...	w _{t1}
D ₂	w ₁₂	w ₂₂	...	w _{t2}
...
D _n	w _{1n}	w _{2n}	...	w _{tn}

The bag-of-words model is commonly called a *term frequency* vector. We can use this term frequency in more elaborate vectorization schemes such as TF-IDF. TF-IDF multiplies the frequency of the word within a document by the rarity with which it appears in the corpus overall. In other variations of the bag-of-words model, we use only 0 or 1 as the vector entry value to indicate that the word occurred or did not occur. The bag-of-words model requires multiple passes of the dataset, which can be expensive, especially with larger datasets.

A drawback of the bag-of-words method is that it cannot capture phrases and multi-word expressions. Bag of words, without specific preprocessing, also does not account for potential misspellings or word variations that we might want to combine into a single word. N-grams and other preprocessing techniques alleviate both of these concerns.⁷

TF-IDF

TF-IDF is a great way to fix some inherent issues with the bag-of-words model. Words do not occur with the same frequency across all documents or in the same document. We use TF-IDF to measure the relative weight of how frequently the word occurs versus how distinctive that word is as a marker.

TF-IDF gives us a numerical statistic to indicate how important a word is to a document with respect to a collection of documents. We often consider TF-IDF as a weighting factor in IR and text mining. TF-IDF values increase proportionally to the number of occurrences of a word in a document, but the values are dragged down at the same time by the frequency of the term across the entire corpus. This helps TF-IDF account for words that might appear important in the document but are actually common across the entire collection of documents.

TF-IDF has proven useful because it allows us to use the information about how often a word occurs in a document while considering the frequency of the word in the corpus to control for the facet that some words will be more common than others. The TF-IDF weight is composed of term components, term frequency (TF), and

⁷ Another vectorization variation on the bag-of-words model is the hashing trick in which we use a hashing function to map each word to an index in a fixed-sized vector.

the inverse document frequency (IDF), in which the former (TF) is divided by the latter (IDF). To produce our TF-IDF number, we begin by calculating the “term frequency” portion of the measurement.

Weighting Terms

Search engines often use TF-IDF as a mechanism to score and rank a document’s relevance, given a user query. It’s also effective as a stop-words filter for text summarization and classification. With TF-IDF, stop-words will end up with a small weight, and the terms that occur infrequently relative to the entire corpus are assigned larger weights.

These more important words generally have high TF and larger IDF values, so the product of the two factors is a larger value. In this classic version of TF-IDF, we see that stop-words are assigned a small weight, and the terms that are rare are assigned a much larger weight. This allows us to “see” the words of relatively high value, such as the topic of a document. Important words will have both large TFs and large IDFs, which produces a relatively high TF-IDF score for them.

Besides search engines, TF-IDF also is very useful in the fields of text summarization and classification. This vectorization process is more accurate than the basic bag-of-words model but computationally more complex because we need to look at how often the word occurs in both the document and then the whole corpus. This involves more than one pass over the dataset and some preprocessing passes.

TF

TF is defined in its simplest form as the number of times a word appears in a group of documents. Much like bag of words gives us an indication that a word occurred in the corpus, TF gives us information on how often the word occurred. For more complex forms of term frequency, we like to normalize our counts with respect to the length of documents. Most documents will not have the same length; to account for this, we divide the term frequency by the document length (total count of all terms in the document) as a way to normalize the counts, giving us a more accurate measure.

t = term

d = document

$$tf_{t,d} = \frac{count(t)}{count(alltermsind)}$$

IDF

The next step for calculating TF-IDF is to measure how much information a word provides by calculating the word's **IDF**. We'd like to measure how common or rare this word is and then include that information in our vector with this part of the formula. The intuition is that we want terms that occur in few documents to be assigned a higher value in the vector than terms that occur in many documents across the corpus.⁸

To determine the “document frequency,” first we calculate the fraction of documents containing the word by dividing the total number of documents in the corpus by the number of documents containing the word. This gives us information regarding how rare the word is from the vantage point of looking at the entire group of documents we're considering. To complete the IDF, we then take the logarithm of this derived fraction.

The IDF value in this form is still not ideal, because it masks the effect of TF on the final term weight. To reduce this problem, a usual practice is to use the logarithm of the IDF value instead ([Table 8-6](#) lists the variable definitions):

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

Table 8-6. Components of IDF

IDF variable	Description
df_t	Document frequency; number of documents in the corpus that contains a term t
N	Total number of documents in the input corpus

Computing the full TF-IDF score

Thus, the TF-IDF weight for a word becomes the following:

$$tfidf_{t,d} = tf_{t,d} \times idf_t$$

A high weight for a TF-IDF calculation is driven by words that have relatively high TF and relatively low document frequency with respect to the entire document corpus. Common terms tend to be filtered out because they will have low TFs and high document frequencies. If we look closely at the IDF part of the weight, we see that its value is greater than or equal to 0.

⁸ The IDF was originally introduced in 1972 by Karen Sparck Jones as “term specificity.”

This is due to the value that is input to the logarithm function always being greater than or equal to 1. We can see that as a term occurs more in various documents, the ratio inside the logarithm approaches 1. This gives us an IDF value closer to 0 and by extension a TF-IDF value closer to 0. TF-IDF has shown to be more effective than plain bag of words in practice, but through preprocessing phases, we can gain even more effectiveness through a technique called *n-grams*.

DL4J includes an implementation⁹ of TF-IDF as part of its text-processing tools. A quick example of how we'd use the DL4J implementation of TF-IDF is shown here.

```
File rootDir = new ClassPathResource("tripledir").getFile();
LabelAwareSentenceIterator iter = new LabelAwareFileSentenceIterator(rootDir);
TokenizerFactory tokenizerFactory = new DefaultTokenizerFactory();

TfidfVectorizer vectorizer = new TfidfVectorizer.Builder()
    .setMinWordFrequency(1)
    .setStopWords(new ArrayList<String>())
    .setTokenizerFactory(tokenizerFactory)
    .setIterator(iter)
    .build();

vectorizer.fit();
```



Removing Stop-Words

A common preprocessing step in TF-IDF is removing stop-words. Stop-words dominate raw text in terms of quantity, which can greatly affect the value of the TF-IDF weights. In TF-IDF, we remove the stop-words to achieve a more accurate measure between two document vectors. Examples of stop-words include the following:

- a
- an
- who
- the
- what

Removing these words improves our TF-IDF measure considerably as initial distance values between two documents is dominated by the weights of these stop-words.

⁹ Apache Lucene has a quality implementation of TF-IDF that is used in many places in the JVM landscape, as well.

N-Grams

An n-gram is a contiguous sequence of n items from a given sequence of text or speech. N-gram vectorization techniques can give the vectorization process an idea of how often groups of words occur together in the corpus. A basic TF-IDF implementation will not pick up the information contained in neighboring words during its vectorization process, and can miss dependent information in identifiers such as “Wall Street” or “Coca Cola.”

We use n-grams for tasks in which we have a sequence of elements such as words, and we want to identify or predict the next word. We see that certain words are more probable in certain contexts than others, and we can take advantage of this information. N-grams give us a way to model the context of how a word is used by including some of its neighboring words. We see n-grams used in NLP and speech recognition. In speech recognition, we see words modeled as a series of n-words. In the application of language identification, we see sequences of characters (or graphemes, “letters of the alphabet”) as n-grams when we’re trying to discover the language in which a text is composed. When we parse a document or body of text, we model words such that each n-gram is composed as n words in a sequence.

You should take care with n-grams because when n is high enough, it becomes computationally expensive to run the algorithm. The vocabulary growth is overwhelming even when $n = 5$, so you should use n-grams with caution and use a lower n value.

Kernel Hashing

Another method you can turn to in text vectorization is called kernel hashing. It turns out that feature creation in vectorization is not that easy, and we can always benefit from making things simpler. Making multiple passes through the data as in TF-IDF is difficult when we deal with a lot of data with respect to having to touch a lot of hardware while trying to attain good throughput. You use kernel hashing when you want to vectorize the data in a single pass, making it a “just in time” vectorizer. The advantage to using kernel hashing is that you don’t need the precursor pass like you do with TF-IDF, but you run the risk of having collisions between words. You can use it when you want to vectorize text right before we feed it to our learning algorithm.

Comparing Word2Vec and VSM Comparison

As we discussed in [Chapter 5](#), Word2Vec detects similarities between words mathematically by learning the context in surrounding words. Word2Vec creates vectors that are distributed numerical representations of word features, such as the context of individual words. Specifically, Word2Vec produces a list of word embeddings (vectors) in which a single vector represents each word.

With Word2Vec, we are able to capture semantics of word use by understanding the word's position across many sentences and relating that position to meaning. This allows Word2Vec vectors to capture latent information in the input data based on the neural network's internal representation of the word.



Word2Vec Vector Arithmetic

The word meaning and relationships in the produced word embeddings are encoded spatially and have useful characteristics such as vector arithmetic.

Vector space models (e.g., TF-IDF), in contrast, are built using only co-occurrence statistics from the input document list. This allows for interesting properties, such as being able to calculate the similarity between two documents (or words), yet is limited in representing semantic or syntactic relationships between words or documents (by using only co-occurrence information in the corpus).

Word2Vec's vector representations have allowed for novel approaches to raw text understanding,¹⁰ recommendation,¹¹ and graph modeling,¹² as previously mentioned in this book. For code samples on how to construct Word2Vec (and paragraph vectors), refer to [Chapter 5](#).

Working with Graphs

Vectorizing graph structures is a difficult problem in engineering data science pipelines. Although we won't exhaustively cover the topic of graph vectorization here, we'll discuss the Node2Vec variant (see also [Chapter 5](#)).

With Node2Vec¹³ we can generate a vector for each vertex (and, by extension, each edge) in a graph. We can use these vectors just like word vectors (mentioned in the previous section) for a variety of tasks, such as node classification and inferring whether an edge (connection) should exist.

Node2Vec is a generalization of another graph method called DeepWalk.¹⁴ DeepWalk works by performing truncated random walks on the graph to build up local information about the graph structure. This allows DeepWalk to learn latent representations in the graph by treating the trips through the graph as the equivalent of

¹⁰ Nay. 2016. "[Gov2Vec: Learning Distributed Representations of Institutions and Their Legal Text](#)."

¹¹ Barkan and Koenigstein. 2016. "[Item2Vec: Neural Item Embedding for Collaborative Filtering](#)"

¹² Grover and Leskovec. 2016. "[node2vec: Scalable Feature Learning for Networks](#)."

¹³ *Ibid.*

¹⁴ Perozzi, Al-Rfou, and Skiena. 2014. "[DeepWalk: Online Learning of Social Representations](#)."

sentences. Finally, we would like to note the work¹⁵ with **graph CNNs** as something you should be aware of, but these are not currently supported in the DL4J framework.

Adjacency Matrices

Vectorizing graphs as an adjacency matrix is a common method¹⁶ we see in machine learning for this task. In this method, we use a square matrix to represent a finite graph structure where the rows and columns of the matrix each have a list of all of the nodes in the graph. The adjacency matrix is a (0,1)-matrix with its diagonal filled with zeros (a node cannot have an edge connected to itself in this type of representation).

In this book, we direct you more toward methods that are able to operate directly on the graph structure as opposed to custom variants of building adjacency matrices.

¹⁵ Niepert, Ahmed, and Kutzkov. 2016. “Learning Convolutional Neural Networks for Graphs.”

¹⁶ We also see the “adjacency list” data structure used in a similar fashion for graph vectorization. “Section 22.1: Representations of graphs,” *Introduction to Algorithms* (Second ed.), MIT Press and McGraw-Hill: pp. 527–531.

Using Deep Learning and DL4J on Spark

*Ten years on the road, making one night stand
Speeding my young life away
Tell me one more time just so I'll understand
Are you sure Hank done it this way?
Did old Hank really do it this way?*

—Wylon Jennings, “Are You Sure Hank Done It This Way”

Introduction to Using DL4J with Spark and Hadoop

Two key datacenter technologies that have emerged in the past decade are Apache Hadoop and Apache Spark. Hadoop in particular has become the epicenter of data warehouse growth and evolution. Spark has succeeded MapReduce to become the mainline execution framework on Hadoop for running parallel iterative algorithms.

DL4J supports scale-out of network training on Spark. We can use Spark execution for DL4J to significantly reduce the time required to train our networks. This scenario also gives us the option to mitigate increased training time as input size grows.



To the Cloud!

Platforms such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure makes it possible allow people to set up a Spark cluster on demand, for just a few dollars. DL4J is able to run on most public cloud infrastructure,¹ giving practitioners flexibility in how and where they run their deep learning workflows.

¹ Currently, DataBrick's cloud Spark offering is not supported, but that might change in the future.

Spark is a general parallel-processing engine that can execute on its own, on an **Apache Mesos** cluster, or on a Hadoop cluster via the Hadoop YARN (Yet Another Resource Negotiator) framework. It can work with data in the Hadoop Distributed File System (HDFS) by using input formats included with Hadoop. Spark uses some techniques around caching frequently used data in memory with Resilient Distributed Datasets (RDDs, covered in a moment). Spark also allows the programmer to abstract away parallel processing and focus more on the algorithm at hand. In this book, we focus on the batch processing aspects of Spark as applied to parallel iterative algorithms such as Stochastic Gradient Descent (SGD) in DL4J.

In the following list, we present the key components of a Spark job:

Application

The Spark job JAR we compile represents our Spark application. It can be a single job, multiple jobs chained together, or an interactive Spark session.

Spark driver

This runs the Spark context and converts the application into a directed graph of tasks. These tasks are scheduled to run on the cluster. There is only a single driver per Spark application.

Spark application master

When we run Spark on Hadoop via YARN, the Spark application master negotiates with YARN for resources on the cluster. There is a single application master per Spark application.

Spark executor

An executor runs multiple tasks on a single Java Virtual Machine (JVM) instance on the local host on which the executor is running. The Spark driver instructs the long-lived executors about what tasks they need to run. A single host can have multiple Spark executors locally on it. A cluster can end up having 100s, or even 1000s, of executors spread across many machines running different Spark applications concurrently.

Spark task

Represents a single unit of work for an executor to perform on part of a distributed dataset (also called an RDD).

RDD

A fault-tolerant collection of elements that can be processed in parallel.

RDD

RDDs are a core concept in Apache Spark. These datasets are a fault-tolerant collection of elements that can be processed in parallel. The operations on RDDs are compiled from higher-level language code to make it easier for the programmer to focus on the algorithm and business problem and put less thought into managing the distributed-systems, aspect of the execution.

Following are two common ways in which we can create an RDD in a Spark program:

- Parallelize an existing collection in our Spark application
- Referencing a dataset in an external storage system

The external storage systems include HDFS, **HBase**, **Cassandra**, or any other system that is compatible with a Hadoop `InputFormat`. In practice, this is most often HDFS.

Apache Hadoop is a set of parallel processing tools (e.g., MapReduce) and a distributed file system (e.g., HDFS) written in Java. It is based on the published design of Google's MapReduce² and Google File System (GFS). Hadoop was originally constructed to parallelize inverted index construction for the **Apache Lucene** search engine as part of the **Apache Nutch** project. The Hadoop project was started by **Doug Cutting** and Mike Cafarella and ultimately kicked off the democratization of search engine infrastructure, later revolutionizing how data warehouses are built. In January 2008, Hadoop was split out from the Nutch project into its own top-level Apache project.

Yahoo! further incubated the Hadoop project by adopting the technology internally and putting much-needed engineering resources into the project. In April 2008, Hadoop claimed the world record for sorting a terabyte of data (on a 910 node Hadoop cluster) **in only 209 seconds**. By 2009 many companies, such as last.fm, Facebook, the *New York Times*, and even the Tennessee Valley Authority³ saw Hadoop as a practical way to use commodity hardware to **parallelize processing on large amounts of data**. Today, multiple Hadoop distribution vendors have taken Hadoop to the Fortune 500 and beyond. Apache Hadoop has become the modern data warehouse infrastructure and is a de facto standard as an enterprise execution environment.

² Dean and Ghemawat. 2008. "MapReduce: Simplified Data Processing on Large Clusters."

³ See <http://cnet.co/2uytyct> and <http://bit.ly/2uSutEa>.



DL4J, Spark, and Hadoop

DL4J was designed from day one to be a first-class citizen in the Hadoop environment. DL4J is built to operate on standalone Spark, Spark on Mesos, or with Spark running via YARN on a Hadoop cluster. The practitioners who experiment with DL4J might be using only DL4J on a Spark-only cluster, but large enterprises tend to operate Spark in the context of Spark on Mesos or a supported Hadoop cluster under YARN.

Operating Spark from the Command Line

You typically run Spark jobs from the command line and include a number of parameters. In the subsection that follows, we explain the basics of running a Spark job from the command line and how to deal with the different options.

spark-submit

The `spark-submit` bash script is how we submit jobs to the cluster. We can see the `spark-submit` command and its required command-line options to run the job on a Hadoop cluster ([CDH](#), [HDP](#)) with spark (via YARN) below.

```
spark-submit --class [class name] --master yarn [jar name]
              [job options]
```

Following are descriptions of each of the attributes in the preceding code snippet:

class name

This is the fully qualified class name.

jar name

This is the path and JAR filename. This is also known as an “uber jar”; it includes everything we need to run the job.

job options

These are the various options the Spark job takes.

Here's an example:

```
spark-submit --class io.skymind.spark.SparkJob --master yarn
              /tmp/Skymind-SNAPSHOT.jar /user/skymind/data/iris/iris.txt
```

In this job, we're running the class `io.skymind.spark.SparkJob` in the JAR `Skymind-SNAPSHOT.jar`. The job has one parameter detailing the location of the input data.

Sometimes, you might want to change certain job configuration properties from the command line at execution time. We can do this either by putting extra flags on the command line or by specifying them in a special configuration file, as demonstrated here:

```
spark.master      spark://mysparkmaster.skymind.com:7077
spark.eventLog.enabled   true
spark.eventLog.dir       hdfs:///user/spark/eventlog
# Set spark executor memory
spark.executor.memory    2g
spark.logConf           true
```

It's pragmatic to place configuration key values into a text file, such as in the preceding example, to make job execution faster and more manageable. We generally put this text file in the local directory with the job jar and reference it when we run the job from the command line.

Working with Hadoop security and Kerberos

Kerberos is a common enterprise-grade authentication system. Kerberos gives us protection from attacks such as intercepted authentication. It also helps stop the threat of user impersonation by not sending credentials across the network in clear text. (If you aren't interested in reading about security, go to ["Configuring and Tuning Spark Execution" on page 362](#).)

Major Hadoop distributions such as CDH and HDP support Kerberos authentication.



Certified to Run on Hadoop Distributions

Skymind (commercial support for DL4J) ensures that new DL4J releases are certified on new releases of both CDH and HDP.

Kerberos credentials are optionally stored in a Lightweight Directory Access Protocol (LDAP) or active directory.

To run Spark on YARN on a Kerberized cluster we need to do two things:

1. Manually upload the Spark assembly jar to HDFS
2. Initialize Kerberos on the command line

Uploading the Spark assembly. We want to first manually upload the Spark assembly jar to the HDFS directory:

```
/user/spark/share/lib
```

The Spark assembly jar is located on the local filesystem, typically at:

```
/usr/lib/spark/assembly/lib
```

or, on CDH:

```
/opt/cloudera/parcels/CDH/lib/spark/assembly/lib
```

When running a Spark job on HDP, the library is uploaded into HDFS, so the user running the job **needs permission to write to HDFS**.



Kerberos and Spark Note

On Kerberized clusters, if we try to run a Spark job without manually uploading the jar to HDFS, the job will fail. The upload jar command (running as part of the job) will fail silently because Spark has limited Kerberos support.

Initializing Kerberos. To initialize Kerberos, first type the following:

```
kinit [user-name]
```

You are prompted to enter a password. After we have Kerberos initialized we can confirm that we have our Kerberos ticket:

```
klist
```

This will show something similar to the following output and confirm that our Kerberos ticket is active and valid, as demonstrated here:

```
[skymind@sandbox ~]$ klist
Ticket cache: FILE:/tmp/krb5cc_1025
Default principal: skymind@HORTONWORKS.COM

Valid starting     Expires            Service principal
07/05/16 20:39:08  07/06/16 20:39:08  krbtgt/HORTONWORKS.COM@HORTONWORKS.COM
renew until 07/05/16 20:39:08
```

Configuring and Tuning Spark Execution

Spark can run on different types of distributed platforms or locally on a single machine. For the purposes of this chapter, we'll focus on Spark as executed on a YARN-based Hadoop cluster and on a Mesos-based cluster. These systems should be familiar to the enterprise user of either Cloudera's CDH or Hortonworks' HDP distributions as well as users of the Apache Mesos cluster management system.

Spark can execute in different ways based on whether the execution is distributed and where the Spark Driver process for the job is executing while the job runs. Under-

standing some basic concepts around how this works will help you to move from a simple job locally to executing a long-running job on a cluster that can be disconnected from and left to run overnight.

Each Spark application has a driver process that can run in the foreground (client mode) or the background (cluster mode). Closing a foreground client, for instance, stops the job because we're discontinuing the local process that is controlling the job on the cluster. However, we can quit a background client and allow the Spark job to continue to run because the job controller is running on another host. This active driver is used by Spark to manage the job flow and schedule tasks for Spark.

In client mode, the Spark Driver process is running on the local machine on which it was executed. In cluster mode, the Spark Driver process is run on a remote machine in the cluster itself.

Running Spark on Mesos

You might prefer to operate Spark in distributed mode without a Hadoop distribution [with the supported Apache Mesos](#) cluster manager. Mesos is a way for a user to abstract CPU, memory, storage, and other compute resources away from the physical machines themselves. It allows us to treat the cluster of machines as a single logical pool of resources, enabling fault-tolerant, multitenant systems to be run more efficiently.

The Mesos master replaces the Spark master as the cluster manager when we're using Mesos to manage our cluster. In this mode, Mesos will determine what machine handles which task when a driver creates a Spark job and creates tasks to be scheduled. There will be many short-lived tasks on the Mesos cluster and Mesos is able to direct these tasks in such a way that it allows for multitenancy with other frameworks and tasks on the cluster.

There are two main modes by which you can use Spark on Mesos:

- Client mode
- Cluster mode

In client mode, the Spark Mesos framework is launched directly on a client machine and waits for the user to enter commands or run a program. The user will see the driver output directly on the console of their screen.

To run a job using [Spark on Mesos in client mode](#), you need to do the following:

1. Set up certain Mesos-related environment variables in `spark-env.sh`
2. Pass in the correct Mesos cluster URL to the `SparkContext`

In [cluster mode](#), the Spark Driver is launched on a host in the cluster itself, and the user will be able to see the results of the job from the Mesos Web UI. In cluster mode,

we'll need to start the `MesosClusterDispatcher` in our cluster via the `sbin/start-mesos-dispatch.sh` script along with the correct URL for the Mesos master URL. We can use the same `spark-submit.sh` script as before, but we'll need to include the master URL for the `MesosClusterDispatcher`, as shown in the following example:

```
./bin/spark-submit \
--class io.skymind.spark.mesos.MyTestMesosJob \
--master mesos://210.181.122.139:7077 \
--deploy-mode cluster \
--supervise \
--executor-memory 20G \
--total-executor-cores 100 \
/tmp/mySparkJob.jar \
1000
```



To learn more about using Mesos with Spark, check out the [page](#) for “Running on Mesos” on the Apache website for Spark. This page has a complete listing of key configuration options to note for configuring Mesos jobs.

There is also the scenario in which you might want to run Spark and Mesos alongside an existing Hadoop cluster as separate services on the same machines. In this scenario, the Spark jobs on Mesos would be able to access the data in HDFS by using a fully qualified URL for addresses within the HDFS.

Running Spark on YARN

YARN is the Hadoop API exposed to general-purpose applications to allow programs to run as first-class citizens alongside MapReduce. The current Spark on YARN implementation supports two modes for executing Spark jobs:

- `yarn-client`
- `yarn-cluster`

YARN applications encompass the concepts of `ApplicationMasters` and `NodeManagers`.

YARN, ApplicationMasters, and NodeManagers

ApplicationMasters track the resources and execution of a specific job in a Hadoop cluster. NodeManagers allocate YARN containers where job tasks execute. The intent of this section is not to confuse you with more distributed system jargon but to provide some context about what all goes on inside a production-cluster Hadoop cluster.

In the case of Spark, we see Spark executors running inside YARN containers and the Spark job being coordinated by the Spark ApplicationMaster. We just want you to be aware of these processes to better understand how, where, and when they run their Spark jobs.

Each Spark executor runs as a YARN container.⁴ Spark hosts multiple tasks in the same container allowing for orders of magnitude quicker task startup time.

Let's look at each mode more closely:

yarn-client

In this mode, the Spark Driver is executing on the machine from which the job was submitted. Many times, this will be the developer's local laptop connected to the cluster through the network.

The Spark Driver communicates with the Spark ApplicationMaster in the Hadoop cluster and issues commands to be executed as tasks on the Spark Executors running within YARN containers.

yarn-cluster

The Spark Driver process runs remotely on an ApplicationMaster in a YARN-based Hadoop cluster (HDP,⁵ CDH). The YARN Application Master, from which YARN expects input to coordinate the job, also runs the Spark Driver process. A Spark job run in this manner allows the user to shut off his client or terminal session and still have the job complete.

Comparing Spark execution modes

Table 9-1 gives a quick summary of the effects of running Spark in different modes under YARN. This table was originally created by Sandy Ryza for [Cloudera's Engineering Blog](#).

⁴ Not to be confused with the concept of a Linux container; this is a different concept.

⁵ See <http://bit.ly/2tQmuuh> and then launching remote Spark jobs from NiFi.

Table 9-1. Understanding the different Spark execution modes under YARN

	YARN cluster	YARN client	Spark standalone
Driver runs in:	ApplicationMaster	Client	Client
Who requests resources?	ApplicationMaster	ApplicationMaster	Client
Who starts executor the processes?	YARN NodeManager	YARN NodeManager	Spark worker
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and workers
Supports Spark Shell?	No	Yes	Yes



When to Use Which YARN Spark Execution Mode

When we're interactively debugging our Spark jobs on YARN, we likely want to use the `yarn-client` mode. When we have a long running or scheduled production Spark job, we'll likely use the `yarn-cluster` mode.

Why Run Spark on YARN or Mesos?

The YARN framework in Hadoop allows applications beyond MapReduce to take advantage of the execution environment in Hadoop. Running Spark on YARN allows us to:

- Dynamically share and centrally configure the same pool of cluster resources between different concurrent applications and frameworks
- Take advantage of the yarn scheduler capabilities for better concurrency between applications
- Dynamically allocate and control how many executors are being used at any given time on the cluster for a single Spark application
- Kerberos support⁶

Being able to dynamically share the cluster between multiple applications is an important property of an enterprise production system. This means that we can run Impala queries, MapReduce jobs, and Spark applications all at the same time on the same cluster and have some reasonable way to govern how many resources each application consumes. Being able to use the YARN or Mesos scheduler allows us to concurrently run ad hoc queries on the cluster while also running a scheduled production job and ensuring that the production job is allocated enough resources to complete the required Service-Level Agreement (SLA). This is because YARN—as well as Mesos—understands how to manage multiple frameworks concurrently in a way that governs resources in a predictable and manageable fashion.

⁶ Mesos supports most of these, except for the Kerberos support.

In a multitenant environment with lots of types of workloads being run on the same cluster, YARN and Mesos are DevOps' best friend.

General Spark Tuning Guide

At a high level, we have two major tuning knobs to work with when adjusting a Spark job: CPU and memory. We can set the number of executors, the number of cores each executor uses, and then the amount of memory each executor has to work with for processing.



Mesos and GPUs

Under Mesos we also have the option to manage graphics processing units (GPUs), whereas currently with YARN, we do not.

Setting the number of executors

We can set the number of executors for our application with a command-line flag or configuration file property. The command-line flag is set by using the following:

```
--num-executor
```

The configuration file property is set by using this property key:

```
spark.executor.instances
```

We do this in the *spark-defaults.conf* file, an extra spark configuration file, or with a `SparkConf` object from the API.

Spark executors and CPU cores

Every executor in a Spark application has the same number of cores available to it. We set it by using the command line or in a configuration file property. The command-line flag is set as follows:

```
--executor-cores
```

The configuration file property is set with the property key:

```
spark.executor.cores
```

We do this in the *spark-defaults.conf* file, an extra spark configuration file, or with a `SparkConf` object from the API.



Executor Cores and Concurrent Executor Tasks

When we set a number of cores available to an executor in batch processing mode for Spark, we are effectively setting the number of tasks to be executed at the same time.

Spark executors and memory

In a similar fashion to controlling cores in executors, we can control the heap size for each executor by using the following:

```
--executor-memory
```

Along with this, you list the size in either megabytes (m) or gigabytes (g) of ram allocated per executor. We also can set this property from a configuration file with the property, as demonstrated here:

```
spark.executor.memory
```

Inside of `spark.executor.memory` we have two ways to further control memory usage in Spark:

- `spark.shuffle.memoryFraction`
- `spark.storage.memoryFraction`

These two settings control how the executor divides its memory between transforming data and persisting data. Let's take a closer look at each of them:

`spark.shuffle.memoryFraction`

Controls how much Java heap memory is used for aggregation and cgroups during the shuffle process in Spark. Defaults to 0.2 and we recommend staying with this setting at least early on.

`spark.storage.memoryFraction`

The parameter to control how Spark manages the total size of the cached RDD. The default value is 0.6, which controls the fraction of Java heap memory to use for Spark's memory cache. This ensures that the executor doesn't exceed RDD heap space volume multiplied by this value.

The Spark executor also can have overhead, which we manage with the setting, using the following:

`spark.yarn.executor.memoryOverhead`

Controls the amount of off-heap memory to be allocated for each executor. This memory is used for virtual machine (VM) overhead, interned strings, and other native overhead.

The default value is (`executorMemory * 0.10`) with a minimum value of 384 MB.

Spark and YARN container resource allocation

When working with Spark on YARN, you must consider the resources that YARN has available.⁷ The major YARN properties to consider are as follows:

- `yarn.nodemanager.resource.memory-mb`
- `yarn.nodemanager.resource.cpu-vcores`

`yarn.nodemanager.resource.memory-mb`

This setting controls the maximum amount of memory used by the containers on each host in the Spark cluster.

`yarn.nodemanager.resource.cpu-vcores`

This setting controls the maximum amount of cores used by the containers on each host in the Spark cluster. If we ask for 10 executor cores for our Spark job, this becomes a request in YARN for 10 virtual cores.

Understanding executor memory requests in YARN

The parameter `spark.executor.memory` controls the executor heap size, but the JVM can also use some memory off heap. To calculate the full YARN memory request, we need to add `spark.executor.memory` and `spark.yarn.executor.memoryOverhead` together to get the full YARN memory request. [Figure 9-1](#) presents this visually.

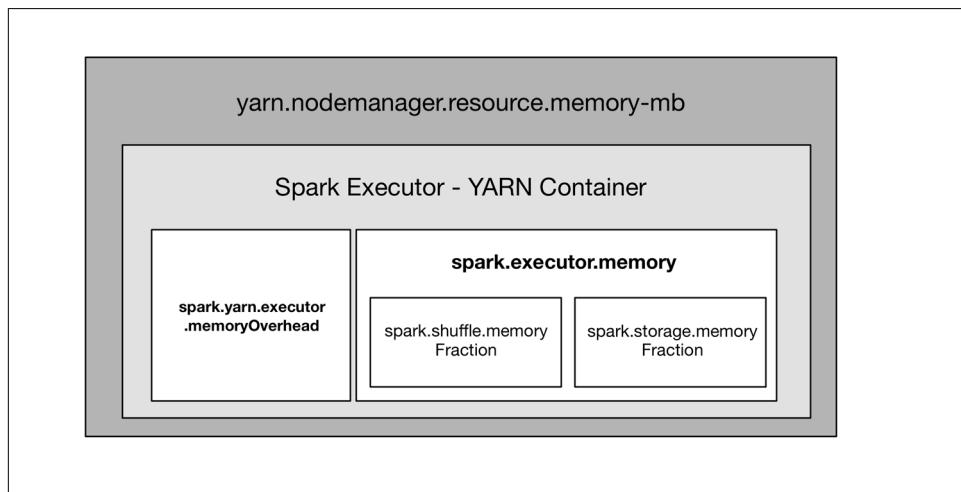


Figure 9-1. Hierarchy of memory configurations of Spark on YARN

⁷ It's also worth noting that YARN trumps Spark in allocation because the Spark process runs within a constrained YARN container.

To get the full memory request for a Spark executor, we take the value of `spark.yarn.executor.memoryOverhead` and `spark.executor.memory` and add them together.



Be Mindful of How Much Memory Is Requested

If the sum of `spark.yarn.executor.memoryOverhead` and `spark.executor.memory` is greater than the value of the `yarn.nodemanager.resource.memory-mb` setting in YARN, the job will not start, because it will not be allocated the required container resources from the YARN system.

The default value for `spark.shuffle.memoryFraction` is 0.2 and the default value for `spark.shuffle.safetyFraction` is 0.8. Here's how we calculate the amount of memory available to each executor task:

```
( spark.executor.memory * spark.shuffle.memoryFraction  
* spark.shuffle.safetyFraction ) / spark.executor.cores
```

Hopefully we've provided an adequate overview of dealing with Spark and YARN tuning so you can get started. You should be aware that there are many more settings to tune in Spark and YARN that are beyond the scope of this book.



Further Reading

Check out [Sandy Ryza's blog entries](#) on Cloudera's Engineering Blog for more [Spark tuning tricks](#).

Understanding Spark, the JVM, and garbage collection

The topic of JVM tuning under Spark is large and complex. For the purposes of this book, we'll summarize a few key topics and let you review the topic further as needed on your own.

Dealing with slowing garbage collection efficiency or pauses. The practitioner should check and make certain the Spark application is using the allocated memory space effectively. If the RDD is taking up a lot of memory, the executor will have less space to work with and you'll see performance degradation.

If you see garbage collection happening too often, Spark needs to use its memory more efficiently. We can alleviate some of the issues by explicitly cleaning up cached RDDs after they are no longer needed.

Selecting a garbage collector for the JVM and Spark. It's generally recommended that you use the [G1 Garbage Collector](#) for Spark executors.

Tuning DL4J Jobs on Spark

Now that we've covered the basic mechanics of tuning a Spark job, we can look at how tuning a DL4J job works on Spark. Just like with general Spark tuning, we can control three major factors in our DL4J jobs:

- Number of workers (executors)
- Amount of memory that's available for each worker to use
- Number of cores that are available for each worker to use

Let's see how each of these factors affect training a deep learning model.

Tuning the number of executors

As we add more executors (workers) to a Spark job, we can partition the data out more and allow each worker to process fewer records. This will decrease total training time.



Diminishing Returns and Executors

Adding more executors (beyond a certain point) will result in a diminishing return on investment, in terms of accuracy obtainable per unit time for training. Effectively, after you've subdivided the workers' data split down to the mini-batch size and you have the same number of workers as data splits, you can go no lower. Adding more workers would be useless because the workers would have no data to process.

Tuning the amount of memory for executors

A single mini-batch of training records is sent to ND4J as a single matrix or block of data to be processed in a vectorized fashion. This allows the hardware to be more efficiently used. Controlling mini-batch size is important because it affects not only learning, but also how much memory the executor will need.⁸

The higher the number of records we have for our executor mini-batch factor, the more memory we'll need allocated to the executor. If we have limited memory for the executor to use, we'll have to be mindful of not setting the mini-batch too high.

⁸ Breuel. 2015. "The Effects of Hyperparameters on SGD Training of Neural Networks."



Rules of Thumb for Memory Usage

For float data (most common): 4 bytes per value, plus some overhead. Here are a couple of examples:

MNIST data (small): $28 \times 28 = 784 \rightarrow \sim 3 \text{ kB}$ (plus labels – 10 values) per example

Time-series with 256 inputs, 1,000 time-steps $\rightarrow 1 \text{ MB}$ per example for features (plus labels)

Understanding Parallel Performance

Parallel training of iterative algorithms has some minor effects on convergence. Users of DL4J’s distributed deep learning can choose the mini-batch size, and the software will optimally distribute the workload across workers in Spark. This is an optimal input split similar to Hadoop’s. The system will maximize the number of processing units available to it, automatically scaling horizontally. Running DL4J on runtimes such as YARN directly, or Spark on Hadoop, will run a worker on a split of that dataset based around Hadoop’s InputSplit (or HDFS data block) system, which will also give us a good balance of disk I/O per worker.

In practice, we see parameter averaging act as a regularizer for the parameter vector values. Typically, we recommend beginning by increasing the number of training epochs by 10 percent to account for the extra regularization. Although the number of epochs goes up slightly, we see the time to perform each training epoch reduced by a factor of the number of Spark executors working on the training.

Some settings (like infrequent averaging periods) will require more epochs. Averaging every mini-batch (not recommended for performance reasons) should be very similar to local training (with a mini-batch size equivalent to the total number of examples for all executors, for one iteration).

Setting Up a Maven Project Object Model for Spark and DL4J

Setting up a Maven Project Object Model (POM) file is a key part of building a DL4J project, Hadoop job, or Spark job. Apache Maven allows you to package all of the needed files and dependencies into a single **JAR artifact**. In this section, we look at some best practices when building DL4J POM files. Following are the major dependencies for a DL4J Spark application:

- DL4J
- ND4J
- DataVec
- DL4J-Spark

In the following sections, we look at how to get these going in a Maven *pom.xml* file. Depending on the version or Spark you’re running, you’ll need to configure the *pom.xml* file differently. To support basic interactions with Hadoop, we add the following Maven dependency:

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-common</artifactId>
  <version>${hadoop.version}</version>
  <scope>${spark.scope}</scope>
</dependency>
```

The `hadoop.version` variable is going to be set in the properties area of the *pom.xml* file depending on the Hadoop distribution. To see more on the values this Maven variable can take on check out the Hadoop vendor’s websites for Maven artifacts ([CDH](#), [HDP](#)).⁹

To use DL4J on Spark, you’ll need to include the `deeplearning4j-spark` dependency:

```
<dependency>
  <groupId>org.deeplearning4j</groupId>
  <artifactId>dl4j-spark_${scala.binary.version}</artifactId>
  <version>${dl4j.version}</version>
</dependency>
```



scala.binary.version

In the preceding code snippet, note that `_${scala.binary.version}` is a Maven property. It must be `_2.10` or `_2.11` and should match the version of Spark you are using.

The Maven variable `spark.version` is going to be dependent on the Hadoop distribution or Spark distribution we’re running on, and the `scala.binary.version` variable is dependent on what version of Spark we’re using. [Table 9-2](#) provides a summary of these variables.

⁹ For more on how semantic versioning works with Hadoop, go to the [Apache Hadoop website](#).

Table 9-2. Key Maven pom.xml entries

Maven variable	Description
hadoop.version	Describes the distribution or version of Hadoop we're using (e.g., CDH, HDP, etc.)
scala.binary.version	Dependent on what version of Spark we're using
spark.version	Dependent on the Hadoop distribution or Spark distribution



Platform Dependency

If you're building on one platform but deploying on another, use `nd4j-native-platform` dependency instead. Setting this property includes native binaries for all platforms.

For example, if you are building your project in Maven on a Macbook Pro and you intend to run the job on a Spark cluster running RedHat on the nodes, you will need to set this property to target RedHat.

A pom.xml File Dependency Template

In this section, we show how to set up the dependency area of the `pom.xml` file. We'll first develop the template for the dependencies. We'll then show how the variables are configured depending on the specific Hadoop distribution. Example 9-1 provides the code for this setup.

Example 9-1. DL4J project pom.xml file

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.nd4j</groupId>
      <artifactId>nd4j-native-platform</artifactId>
      <version>${nd4j.version}</version>
    </dependency>

    <dependency>
      <groupId>org.nd4j</groupId>
      <artifactId>nd4j-api</artifactId>
      <version>${nd4j.version}</version>
    </dependency>

    <dependency>
      <groupId>org.scala-lang</groupId>
      <artifactId>scala-library</artifactId>
      <version>${scala.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

```

<dependencies>

    <!-- Spark and Scala Dependencies -->
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-mllib_${scala.binary.version}</artifactId>
        <version>${spark.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
        <version>${scala.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_${scala.binary.version}</artifactId>
        <version>${spark.version}</version>
        <scope>${spark.scope}</scope>
    </dependency>

    <!-- DeepLearning4j Dependencies -->
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>deeplearning4j-core</artifactId>
        <version>${dl4j.version}</version>
    </dependency>

    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>dl4j-spark_${scala.binary.version}</artifactId>
        <version>${dl4j.version}</version>
    </dependency>

    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-kryo_${scala.binary.version}</artifactId>
        <version>${nd4j.version}</version>
    </dependency>

    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-native-platform</artifactId>
        <version>${nd4j.version}</version>
    </dependency>

    <!-- DataVec Dependencies -->
    <dependency>
        <groupId>org.datavec</groupId>

```

```

<artifactId>datavec-api</artifactId>
<version>${datavec.version}</version>
</dependency>

<dependency>
    <groupId>org.datavec</groupId>
    <artifactId>datavec-spark_${scala.binary.version}</artifactId>
    <version>${datavec.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.7.1</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<!-- hadoop-mapreduce-client-app -->

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-app</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
</dependency>

<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.7</version>
</dependency>
```

```

<dependency>
    <groupId>org.apache.mrunit</groupId>
    <artifactId>mrunit</artifactId>
    <version>1.1.0</version>
    <classifier>hadoop2</classifier>
</dependency>

<!-- JCommander for parsing args -->
<dependency>
    <groupId>com.beust</groupId>
    <artifactId>jcommander</artifactId>
    <version>${jcommander.version}</version>
</dependency>

</dependencies>

```

Controlling JAR Size

One of the biggest culprits in creating overly large job JARs is including dependencies that are already available on the execution platform. By excluding these JARs, we can sometimes reduce jar size by 50 to 80 percent, making compiling, moving, and executing jobs easier. We can control which dependencies are included in our jar by using the `<scope>` tag, as shown in this example:

```
<scope>provided</scope>
```

Here's an example of a dependency with the scope set in this manner:

```

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>provided</scope>
</dependency>

```



Serialization

Spark still uses Java serialization by default, which is generally considered a bad idea. In [Example 9-1](#), we mitigate this by including the Kryo dependency above, which you can see here:

```

<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-kryo_${scala.binary.version}</artifactId>
    <version>${nd4j.version}</version>
</dependency>

```

We talk more about this dependency later in the chapter when we discuss common issues around ND4J and Spark.

Now, we'll look at how to set up the specific Hadoop distribution variants.

Setting Up a POM File for CDH 5.X

This section gives the key component version numbers and the needed components to build a Spark CDH 5.x job. [Example 9-2](#) presents the properties for the job.

Example 9-2. pom.xml for building CDH5.x job

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <slf4j.version>1.7.5</slf4j.version>
  <jackson.version>2.5.1</jackson.version>

  <hadoop.version>2.6.0-cdh5.5.2</hadoop.version>

  <!-- DL4J Versioning -->
  <nd4j.version>0.7.2</nd4j.version>
  <dl4j.version>0.7.2</dl4j.version>
  <datavec.version>0.7.2</datavec.version>

  <scala.binary.version>2.10</scala.binary.version>
  <scala.version>2.10.4</scala.version>
  <spark.version>1.3.1</spark.version>

</properties>
```



The version numbers for `nd4j.version`, `dl4j.version`, and `datavec.version` all need to match in the Maven `pom.xml` file for any DL4J-based project.

Now we'll switch gears and see what this looks like for Horton's distribution of Hadoop.

Setting Up a POM File for HDP 2.4

This section shows how to configure a `pom.xml` file for a HDP 2.4 Spark job. (We assume that you can take care of creating the `pom.xml` boilerplate.) These settings will fill out the key dependencies and versions of components for the job.

[Example 9-3](#) shows the properties section of the Maven `pom.xml` template for an HDP Spark job. These variables will match up to the dependencies in the following section.

Example 9-3. pom.xml file for HDP 2.4 project

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <slf4j.version>1.7.5</slf4j.version>
  <jackson.version>2.4.4</jackson.version>
  <jcommander.version>1.27</jcommander.version>

  <!-- HDP 2.4 Version modify for older/newer HDP releases -->
  <hdp.version>2.4.0.0-169</hdp.version>
  <hadoop.version>2.7.1</hadoop.version>

  <spark.version>1.6.0</spark.version>
  <spark.scala.version>2.10</spark.scala.version>

  <!-- DL4J Versioning -->
  <nd4j.version>0.7.2</nd4j.version>
  <dl4j.version>0.7.2</dl4j.version>
  <datavec.version>0.7.2</datavec.version>

  <scala.binary.version>2.10</scala.binary.version>
  <scala.version>2.10.4</scala.version>

</properties>
```

It's worth noting that the properties in [Example 9-3](#) are used by the build system to build the Maven artifact JAR. Beyond usage in Maven, these variables are not used in the DL4J project code itself.

Troubleshooting Spark and Hadoop

In this section, we list a few of the common issues that you'll encounter periodically when using Spark on Hadoop.

Sometimes, if the executor requests more RAM per container than is available, the Spark Driver will issue the following report:

```
WARN TaskSchedulerImpl: Initial job has not accepted any resources; check your
cluster ui to ensure that workers are registered and have sufficient memory
```

Many times, we can alleviate this by lowering the amount of RAM or cores requested by the job.

Where to Look to Debug Spark

Table 9-3 lists some of the key ports for viewing and debugging Spark activity.

Table 9-3. Key ports for Spark debugging

Web service	Port number
YARN Job History Server UI	19888
YARN Resource Manager UI	8088
Spark Job History Web UI	18080

Common Issues with ND4J

In the subsections that follow, we list a few common issues encountered when using ND4J on Spark, along with their resolutions.

ND4J and Kryo serialization

Kryo is a serialization library commonly used with Apache Spark. It proposes to increase performance by reducing the amount of time taken to serialize objects.



SerDe

The term “SerDe” is systems engineering shorthand for serialization and deserialization of data. Serialization on Spark involves data and functions. Spark cares only about setting up serialization and relies by default on Java serialization, which is convenient but inefficient.

Hadoop introduced its own SerDe mechanisms (Writables) and uses input and output formats to parse writables from/to file formats. Spark needs these in/out formats to operate on data in HDFS.

However, Kryo has difficulties working with the off-heap data structures in ND4J.¹⁰ To use Kryo serialization with ND4J on Apache Spark, we need to do some additional configuration for Spark. If Kryo is not correctly configured, it’s possible to get `NullPointerExceptions` on some of the `INDArray` fields, due to incorrect serialization.

To use Kryo, add the appropriate `nd4j-kryo` dependency and configure the Spark configuration to use the ND4J Kryo registrator, as follows:

¹⁰ This isn’t so much an ND4J issue as it is a Spark issue, but it’s worth noting here from a practical standpoint.

```
SparkConf conf = new SparkConf();
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.kryo.registrator", "org.nd4j.Nd4jRegistrator");
```



When using DL4J's `SparkDl4jMultiLayer` or `SparkComputationGraph` classes, a warning will be logged if the Kryo configuration is incorrect.

jnind4j and java.library.path

Here's a common error:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no jnind4j in
java.library.path
```

DL4J Parallel Execution on Spark

DL4J supports training neural networks on a Spark cluster as a way to accelerate network training.

Similar to DL4J's `MultiLayerNetwork` and `ComputationGraph` classes, DL4J defines two classes for training neural networks on Spark:

`SparkDl4jMultiLayer`

A wrapper around `MultiLayerNetwork`

`SparkComputationGraph`

A wrapper around `ComputationGraph`

Because these two classes are wrappers around the standard single-machine classes, the network configuration process (e.g., creating a `MultiLayerConfiguration` or `ComputationGraphConfiguration`) is identical in both standard and distributed training. Distributed on Spark differs from local training in two respects, however: how data is loaded and how training is set up (requiring some additional cluster-specific configuration).

The typical workflow for training a network on a Spark cluster (using Spark submit) is as follows:

1. Create your network training class. Typically, this will involve code for the following:
 - Specifying your network configuration (`MultiLayerConfiguration` or `ComputationGraphConfiguration`), as you would for single-machine training.
 - Creating a `TrainingMaster` instance: this specifies how distributed training will be conducted in practice (more on this later).

- Creating the `SparkDl4jMultiLayer` or `SparkComputationGraph` instance using the network configuration and `TrainingMaster` objects.
 - Load your training data. There are a number of different methods for loading data, with different trade-offs; further details will be provided in future documentation.
 - Calling the appropriate `fit` method on the `SparkDl4jMultiLayer` or `SparkComputationGraph` instance.
 - Saving or using the trained network (the trained `MultiLayerNetwork` or `ComputationGraph` instance).
2. Package your jar file ready for Spark submit.
 - If you are using maven, running `mvn package -DskipTests` is one approach.
 3. Call Spark submit with the appropriate launch configuration for your cluster.



Spark Training on a Single Machine

For single-machine training, Spark local *can* be used with DL4J, though this is not recommended (due to the synchronization and serialization overhead of Spark). Instead, consider the following:

- For single CPU/GPU systems, use standard `MultiLayerNetwork` or `ComputationGraph` training.
- For multiple CPU/GPU systems, use `ParallelWrapper`. This is functionally equivalent to running Spark in local mode, though it has lower overhead (and hence provides better training performance).

The current version of DL4J uses a process of parameter averaging in order to train a network. Future versions might additionally include other distributed network training approaches.

The process of training a network using parameter averaging is conceptually quite simple:

1. The master (Spark driver) begins with an initial network configuration and parameters.
2. Data is split into a number of subsets, based on the configuration of the `TrainingMaster`.
3. Iterate over the data splits. For each split of the training data:
 - Distribute the configuration parameters (and if applicable, network updater state for Momentum/RMSProp/AdaGrad) from the master to each worker.
 - Fit each worker on its portion of the split.

- Average the parameters (and if applicable, update state) and return the averaged results to the master.
4. Training is complete, with the master having a copy of the trained network.

Let's now see how these concepts would look in code.

A Minimal Spark Training Example

The following code shows the basic concepts we'll work with later in the chapter to build out our full Spark examples.

```
JavaSparkContext sc = ...;
JavaRDD<DataSet> trainingData = ...;
MultiLayerConfiguration networkConfig = ...;

//Create the TrainingMaster instance
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster =
    new ParameterAveragingTrainingMaster
        .Builder(examplesPerDataSetObject)
        .(other configuration options)
        .build();

//Create the SparkDl4jMultiLayer instance
SparkDl4jMultiLayer sparkNetwork =
    new SparkDl4jMultiLayer(sc, networkConfig, trainingMaster);

//Fit the network using the training data:
sparkNetwork.fit(trainingData);
```

A `TrainingMaster` in DL4J is an abstraction (interface) that allows for multiple different training implementations to be used with `SparkDl4jMultiLayer` and `SparkComputationGraph`. Currently, DL4J has one implementation: the `ParameterAveragingTrainingMaster`. This implements the parameter averaging process shown in the preceding code. To create one, use the builder pattern:

```
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(int dataSetObjectSize)
    ... (your configuration here)
    .build();
```

The `ParameterAveragingTrainingMaster` defines a number of configuration options that control how training is executed. Let's explore these options:

`dataSetObjectSize`

(Required option.) This is specified in the builder constructor. This value specifies how many examples are in each `DataSet` object. Here's a couple of things to keep in mind:

- If you are training with preprocessed `DataSet` objects, this will be the size of those preprocessed `DataSets`.
- If you are training directly from `Strings` (e.g., CSV data to a `RDD<DataSet>` through a number of steps), this will usually be 1.

`batchSizePerWorker`

This controls the mini-batch size for each worker. This is analogous to the mini-batch size used when training on a single machine. Put another way: it is the number of examples used for each parameter update in each worker.

`averageFrequency`

This controls how frequently the parameters are averaged and redistributed, in terms of the number of mini-batches of size `batchSizePerWorker`. Here are three things to keep in mind:

- Low averaging periods (e.g., `averagingFrequency=1`) can be inefficient (too much network communication and initialization overhead, relative to computation).
- Large averaging periods (e.g., `averagingFrequency=200`) can result in poor performance (parameters in each worker instance can diverge significantly).
- Averaging periods in the range of 5 to 10 mini-batches is usually a safe default choice.

`workerPrefetchNumBatches`

Spark workers are capable of asynchronously prefetching a number of mini-batches (`DataSet` objects) to avoid waiting for the data to be loaded. Setting this value to 0 disables prefetching. A value of 2 is often a sensible default. Much larger values are unlikely to help in many circumstances (but will use more memory).

`saveUpdater`

In DL4J, training methods such as Momentum, RMSProp, and AdaGrad are known as “updaters.” Most of these updaters have internal history or state. If `saveUpdater` is set to true, the updater state (at each worker) will be averaged and returned to the master along with the parameters; the current updater state also will be distributed from the master to the workers. This adds extra time and network traffic, but can improve training results. If `saveUpdater` is set to false, the updater state (at each worker) is discarded, and the updater is reset/reinitialized in each worker.

`repartition`

This setting specifies when data should be repartitioned. The `ParameterAveragingTrainingMaster` does a `mapPartitions` operation; consequently, the number of partitions (and the values in each partition) matters a lot for proper cluster

utilization. However, repartitioning is not a free operation, because some data must necessarily be copied across the network. The following options are available:

- **Always:** (Default option.) That is, repartition data to ensure the correct number of partitions.
- **Never:** Never repartition the data, no matter how imbalanced the partitions might be.
- **NumPartitionsWorkersDiffers:** Repartition only if the number of partitions and the number of workers (total number of cores) differs. However, note that even if the number of partitions is equal to the total number of cores, this does not guarantee that the correct number of `DataSet` objects is present in each partition: some partitions may be much larger or smaller than others.

repartitionStrategy

This is the startegy by which repartitioning should be done. Here are your options:

- **SparkDefault:** This is the stardard repartitioning strategy used by Spark. Essentially, each object in the initial RDD is mapped to one of N RDDs independently at random. Consequently, the partitions might not be optimally balanced; this can be especially problematic with smaller RDDs, such as those used for preprocessed `DataSet` objects and frequent averaging periods (simply due to random sampling variation).
- **Balanced:** This is a custom repartitioning strategy defined by DL4J. It attempts to ensure that each partition is more balanced (in terms of number of objects) compared to the `SparkDefault` option. However, in practice this requires an additional count operation to execute; in some cases (most notably in small networks, or those with a small amount of computation per mini-batch), the benefit might not outweigh additional overhead of executing the better repartitioning.

DL4J API Best Practices for Spark

Following are some things you should do to get the most out of your Spark+Hadoop cluster:

- Build a slimmed down jar
- Use a well-tuned cluster
- Optimize the ETL/vectorization pipeline
- Ensure that the JVM is tuned

Ideally, we want to build the smallest job possible. A core tenant in scale-out processing is to “move the compute to the data,” and we want to move the smallest jar possible around the cluster.

Use Spark on a well-tuned cluster with a good distributed filesystem.

What Is a “Good Distributed Filesystem”?

There are several options in the land of distributed filesystems. You should consider looking for a distributed filesystem that is well known to work with the suite of distributed components they are likely to use.

We contend that a data scientist’s job is neither to set up nor maintain distributed systems’ infrastructure. A machine learning project that becomes distracted with infrastructure complexities is a *deadringer*¹¹ for a project that will likely fail. From that perspective, we recommend that you go for a distributed filesystem that includes the following properties:

- Known to be reliable
- Supports Kerberos integration
- Well-tested
- Scalable
- Works with the bulk of the other components in your data science stack

Many times, especially in the context of Spark and Hadoop, this choice will end up being HDFS.

Serious enterprise users will want to have a well-maintained Hadoop cluster with Spark most likely running on a modern Hadoop distribution. We recommend CDH5 or HDP 2.4.

Take ETL and vectorization out of the main training loops. Large jobs on clusters can be expensive, so we’d like to make them as efficient as possible. Ideally, we want to save and load serialized `DataSet` objects, instead of transforming RDDs over and over again.

Finally, a well-tuned JVM goes a long way. Make everyone’s life easier and make sure the JVM is tuned such that we’re not having long garbage collection pauses.

¹¹ It’s been a long road to get to this point in the book. We suggest that you take a break and listen to the classic RJD2 album *Deadringer*. Just imagine it as the soundtrack for the last half of this chapter.

Multilayer Perceptron Spark Example

In this example, we'll revisit the MNIST dataset to classify digits, but this time we'll use a multilayer perceptron and we'll train the network on Spark.

The major thing that has changed in this chapter's examples compared those in [Chapter 5](#) is how we're approaching parallel training on Spark; most everything else for the examples in this chapter is similar.

Another aspect to consider is that we're now working with data in HDFS, and our ETL and vectorization (generally) pipelines need to be built in a fashion such that they can scale horizontally. We can either use Spark functions to do this (as you'll see momentarily) or we can use ETL libraries such as DataVec. In this example, however, we're using some custom built-in code to handle the MNIST format specifically, so it's less of a concern. We note this as something you should consider for other projects involving CSV and text-based data that might need extensive preprocessing.



Be Careful with Single-Process ETL Code

As we deal with larger datasets, we need to be mindful of how we build ETL pipelines. Spark allows us to mix single-process Java code in with Spark functions, and, if we aren't careful, we'll leave code in that will not scale horizontally with our input dataset size. DataVec provides ETL design patterns in [the examples](#) of the GitHub repository to show how we can build effective parallel ETL pipelines in Spark for DL4J workflows.

[Example 9-4](#) lists the code for the Saturn example (adjusted to execute on Spark).

Example 9-4. Saturn dataset for Spark multilayer perceptron network

```
public class MnistMLPExample {
    private static final Logger log = LoggerFactory.getLogger(MnistMLPExample.class);

    @Parameter(names = "-useSparkLocal", description =
        "Use spark local (helper for testing/running without spark submit)",
        arity = 1)
    private boolean useSparkLocal = true;

    @Parameter(names = "-batchSizePerWorker", description =
        "Number of examples to fit each worker with")
    private int batchSizePerWorker = 16;

    @Parameter(names = "-numEpochs", description = "Number of epochs for training")
    private int numEpochs = 15;

    public static void main(String[] args) throws Exception {
```

```

        new MnistMLPExample().entryPoint(args);
    }

protected void entryPoint(String[] args) throws Exception {
    //Handle command line arguments
    JCommander jcldr = new JCommander(this);
    try {
        jcldr.parse(args);
    } catch (ParameterException e) {
        //User provides invalid input -> print the usage info
        jcldr.usage();
        try { Thread.sleep(500); } catch (Exception e2) { }
        throw e;
    }

    SparkConf sparkConf = new SparkConf();
    if (useSparkLocal) {
        sparkConf.setMaster("local[*]");
    }
    sparkConf.setAppName("DL4J Spark MLP Example");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);

    //Load the data into memory then parallelize
    //This isn't a good approach in general - but is simple to use for this
    //example
    DataSetIterator iterTrain =
        new MnistDataSetIterator(batchSizePerWorker, true, 12345);
    DataSetIterator iterTest =
        new MnistDataSetIterator(batchSizePerWorker, true, 12345);
    List<DataSet> trainDataList = new ArrayList<>();
    List<DataSet> testDataList = new ArrayList<>();
    while (iterTrain.hasNext()) {
        trainDataList.add(iterTrain.next());
    }
    while (iterTest.hasNext()) {
        testDataList.add(iterTest.next());
    }

    JavaRDD<DataSet> trainData = sc.parallelize(trainDataList);
    JavaRDD<DataSet> testData = sc.parallelize(testDataList);

    -----
    //Create network configuration and conduct network training
    MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(12345)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .activation(Activation.LEAKYRELU)
        .weightInit(WeightInit.XAVIER)
        .learningRate(0.02)
        .updater(Updater.NESTEROVS).momentum(0.9)

```

```

    .regularization(true).l2(1e-4)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(28 * 28).nOut(500).build())
    .layer(1, new DenseLayer.Builder().nIn(500).nOut(100).build())
    .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
        .activation(Activation.SOFTMAX).nIn(100).nOut(10).build())
    .pretrain(false).backprop(true)
    .build();

//Configuration for Spark training: see http://deeplearning4j.org/spark for
//explanation of these configuration options
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(batchSizePerWorker)      //Each DataSet object: contains
                                    // (by default) 32 examples
    .averagingFrequency(5)
    .workerPrefetchNumBatches(2)          //Async prefetching: 2 examples
                                    //per worker
    .batchSizePerWorker(batchSizePerWorker)
    .build();

//Create the Spark network
SparkDl4jMultiLayer sparkNet = new SparkDl4jMultiLayer(sc, conf, tm);

//Execute training:
for (int i = 0; i < numEpochs; i++) {
    sparkNet.fit(trainData);
    log.info("Completed Epoch {}", i);
}

//Perform evaluation (distributed)
Evaluation evaluation = sparkNet.evaluate(testData);
log.info("***** Evaluation *****");
log.info(evaluation.stats());

//Delete the temp training files, now that we are done with them
tm.deleteTempFiles(sc);

log.info("***** Example Complete *****");
}
}

```

In the following sections, we break out key sections of the code to highlight core differences in the Spark version of the examples.

To build our Spark job project, we need to change into the main directory for the project and run the following Maven command:

```
mvn package
```

This builds the needed Spark job jar in the ./target subdirectory. We'll copy this jar over to a gateway host for our Spark cluster to run our job and use the Spark execution notes from the command line outlined earlier in this chapter.

Setting Up MLP Network Architecture for Spark

In [Example 9-4](#) we can see our network architecture is similar to some of the networks in [Chapter 5](#). In the following example, we highlight network architecture configuration code.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(12345)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .activation(Activation.LEAKYRELU)
    .weightInit(WeightInit.XAVIER)
    .learningRate(0.02)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .regularization(true).l2(1e-4)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(28 * 28).nOut(500).build())
    .layer(1, new DenseLayer.Builder().nIn(500).nOut(100).build())
    .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
        .activation(Activation.SOFTMAX).nIn(100).nOut(10).build())
    .pretrain(false).backprop(true)
    .build();
```

For this example, we're still using SGD to optimize our parameters for the network. The multilayer perceptron example in [Chapter 5](#) had only a single hidden layer, however, and this example has an additional hidden layer. We also see a switch to the leaky Rectified Linear Unit (ReLU) variant of the ReLU activation function, which was used in the multilayer perceptron example in [Chapter 5](#). Beyond that, much of the rest of the network and output layer are the same as the other example. This is a great illustration of how seemingly simple adjustments to a network can enable us to model completely different data. The trick is to figure out what these simple adjustments are through hyperparameter search.



Same Network Architecture for Local and Spark

It's a nice feature of DL4J to be able to develop a network on a subset of the data on a local machine and then port the same architecture to Spark and model the full dataset.

Distributed Training and Model Evaluation

Following are two key differences we want to highlight in this code (compared to the single-process examples in [Chapter 5](#)):

- Introduction of the `ParameterAveragingTrainingMaster` method into this example
- The different training wrapper: `SparkDl4jMultiLayer`

Those two differences are really the only major changes from the example in [Chapter 5](#). Those changes do not cover many lines of code so we can see how making the jump from a local machine learning workflow to a machine learning workflow on Spark is not that difficult. In the example that follows, we see the code snippet from [Example 9-4](#) that introduces the `TrainingMaster`, as explained earlier in this chapter. In this next snippet, we use this to control how we perform parameter averaging in Spark:

```
//Configuration for Spark training: see http://deeplearning4j.org/spark
//for explanation of these configuration options
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(batchSizePerWorker)           //Each DataSet
                                            //object: contains
                                            //(by default)
                                            //32 examples
    .averagingFrequency(5)
    .workerPrefetchNumBatches(2)           //Async prefetching: 2
                                            //examples per worker
    .batchSizePerWorker(batchSizePerWorker)
    .build();
```

The other major change in distributed training code on Spark involves using a different `MultiLayerNetwork` wrapper called `SparkDl4jMultiLayer`, as shown here:

```
//Create the Spark network
SparkDl4jMultiLayer sparkNet = new SparkDl4jMultiLayer(sc, conf, tm);

//Execute training:
for (int i = 0; i < numEpochs; i++) {
    sparkNet.fit(trainData);
    log.info("Completed Epoch {}", i);
}
```

This wrapper works in concert with the `TrainingMaster` to perform all the required distributed training our model needs and largely abstracts us from the details. The `SparkDl4jMultiLayer` class is the way we can control executors in nearly the same fashion as we performed model training on a local machine.

As we can see in the previous snippet, the `SparkDl4jMultiLayer` class takes three parameters:

1. Spark context
2. DL4J network configuration
3. `ParameterAveragingTrainingMaster` object

In the local machine version of this example, we see the `MultiLayerNetwork` class being used for the purpose of modeling the data. The nice thing about this wrapper class is that it looks almost just like the `MultiLayerNetwork` class, and we can use it in a `for` loop to control the number of epochs in the same way.

Finally, we see the network's F1 score being calculated and printed to the logging system:

```
//Perform evaluation (distributed)
Evaluation evaluation = sparkNet.evaluate(testData);
log.info("***** Evaluation *****");
log.info(evaluation.stats());
```

This example is a great model for how the basics of Spark training work with DL4J in a clean, easy fashion. There's not a lot for you to do beyond setting up the few lines of extra Spark code and the `TrainingMaster` to get their deep learning model training on a production, secure Spark cluster.

Building and Executing a DL4J Spark Job

To begin, change the current directory to the base directory of the example. Next, we'll use Maven to build the job jar by invoking the following command:

```
maven package
```

This will produce a full job jar in the `./target/` subdirectory.

After we've copied the job jar to our intended machine to execute on Spark, we type this command:

```
spark-submit --class org.deeplearning4j.examples.feedforward.MnistMLPExample
--num-executors 3 --properties-file ./spark_extra.props
./dl4j-examples-1.0-SNAPSHOT.jar
```

This will output a lot of training information to the console as the model reports back how the training is progressing. The command-line parameters of note here are those that are giving Spark runtime arguments via the `--properties-file` flag. This allows us to take common command-line arguments and save them to a file so that we don't need to type them out every time. The other property of note, `--num-executors`, is setting the number of workers (executors) in Spark; here we are setting it to 3.

Generating Shakespeare Text with Spark and Long Short-Term Memory

Let's revisit the Long Short-Term Memory (LSTM) example from [Chapter 5](#) and see how we can modify the code and [construct the same model on Spark](#). In the explanations that follow we don't go through the code for data loading, because we want to

focus on how the network architecture as well as the training process is slightly different under Spark. [Example 9-5](#) shows the core training method.

Example 9-5. Core training method for LSTM Shakespeare example in Java

```
protected void entryPoint(String[] args) throws Exception {
    //Handle command line arguments
    JCommander jcldr = new JCommander(this);
    try {
        jcldr.parse(args);
    } catch (ParameterException e) {
        //User provides invalid input -> print the usage info
        jcldr.usage();
        try {
            Thread.sleep(500);
        } catch (Exception e2) {
        }
        throw e;
    }

    Random rng = new Random(12345);
    int lstmLayerSize = 200;                                //Number of units in each GravesLSTM
    //layer
    int tbpttLength = 50;                                  //Length for truncated
    //backpropagation through time. i.e.,
    //do parameter updates every 50
    //characters
    int nSamplesToGenerate = 4;                            //Number of samples to generate
    //after each training epoch
    int nCharactersToSample = 300;                         //Length of each sample to generate
    String generationInitialization = null;                //Optional character
    //initialization; a random
    //character is used if null
    // Above is Used to 'prime' the LSTM with a character sequence to
    // continue/complete. Initialization characters must all be in
    // CharacterIterator.getMinimalCharacterSet() by default

    //Set up network configuration:
    MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
        .learningRate(0.1)
        .rmsDecay(0.95)
        .seed(12345)
        .regularization(true)
        .l2(0.001)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.RMSPROP)
        .list()
        .layer(0, new GravesLSTM.Builder().nIn(CHAR_TO_INT.size())
            .nOut(lstmLayerSize).activation(Activation.TANH).build())
```

```

    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)           //MCXENT + softmax for classification
        .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
    .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();

//-----
//Set up the Spark-specific configuration
/* How frequently should we average parameters (in number of minibatches)?
Averaging too frequently can be slow (synchronization + serialization costs)
whereas too infrequently can result learning difficulties (i.e., network may
not converge) */
int averagingFrequency = 3;

//Set up Spark configuration and context
SparkConf sparkConf = new SparkConf();
if (useSparkLocal) {
    sparkConf.setMaster("local[*]");
}
sparkConf.setAppName("LSTM Character Example");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

JavaRDD<DataSet> trainingData = getTrainingData(sc);

//Set up the TrainingMaster. The TrainingMaster controls how learning is actually
//executed on Spark
//Here, we are using standard parameter averaging
//For details on these configuration options, see:
//https://deeplearning4j.org/spark#configuring
int examplesPerDataSetObject = 1;
ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(examplesPerDataSetObject)
    .workerPrefetchNumBatches(2)      //Asynchronously prefetch up to 2 batches
    .averagingFrequency(averagingFrequency)
    .batchSizePerWorker(batchSizePerWorker)
    .build();
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, conf, tm);
sparkNetwork.setListeners(Collections.<IterationListener>singletonList(new
    ScoreIterationListener(1)));

//Do training, and then generate and print samples from network
for (int i = 0; i < numEpochs; i++) {
    //Perform one epoch of training. At the end of each epoch, we are returned a
    //copy of the trained network
    MultiLayerNetwork net = sparkNetwork.fit(trainingData);
}

```

```

//Sample some characters from the network (done locally)
log.info("Sampling characters from network given initialization \'" +
(generationInitialization == null ? "" : generationInitialization) +
"\'");
String[] samples = sampleCharactersFromNetwork(generationInitialization,
net, rng, INT_TO_CHAR,
nCharactersToSample, nSamplesToGenerate);
for (int j = 0; j < samples.length; j++) {
    log.info("----- Sample " + j + " -----");
    log.info(samples[j]);
}
}

//Delete the temp training files, now that we are done with them
tm.deleteTempFiles(sc);

log.info("\n\nExample complete");
}

```

As we did earlier, we'll let you explore the rest of the code on your own, and we are not going to list the data-loading code here for the sake of brevity.

Here are the two key differences we want to highlight in this code:

- Introduction of the `ParameterAveragingTrainingMaster` into this example
- The different training wrapper: `SparkDl4jMultiLayer`

Those two differences are really the only major changes from the example in [Chapter 5](#), and we saw the same changes earlier in the multilayer perceptron Spark example. Those changes do not cover many lines of code, so we can see how making the jump from a local machine learning workflow to a machine learning workflow on Spark is not that difficult. To reinforce that idea, let's look specifically at how we define the LSTM network here in this example.

Setting Up the LSTM Network Architecture

The interesting thing about the highlighted code snippet that follows (from the preceding section) is that it is identical to the single-process code example version in [Chapter 5](#):

```

//Set up network configuration:
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)

```

```

    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(CHAR_TO_INT.size())
        .nOut(lstmLayerSize).activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)           //MCXENT + softmax for
                                                //classification
        .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();

```

The reason why we feel it was important to note this is because from an operational standpoint this is a huge advantage. We are able to take network configurations that we've developed locally and run them easily on a Hadoop or Mesos cluster with Spark.

Again, in this example, we have two hidden layers (GravesLSTM), both with the TanH activation function. We're also using the same custom LSTM output layer (RnnOutput Layer) with the MCXENT loss function coupled with the SOFTMAX activation function. Recall from [Chapter 6](#) that we use the softmax activation function any time we want to predict one of many classes, and here we're predicting the next character (one of many). Let's now take a look at how this code differs slightly from the [Chapter 5](#) version.

Training, Tracking Progress, and Understanding Results

Earlier in this chapter, we outlined the basic concepts involved in moving a Spark example over to Spark when we highlighted the (generalized) code snippet:

```

//Create the TrainingMaster instance
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster = new ParameterAveragingTrainingMaster
    .Builder(examplesPerDataSetObject)
    .(other configuration options)
    .build();

//Create the SparkDl4jMultiLayer instance
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, networkConfig,
    trainingMaster);

```

In the next code snippet, we can see how this actually looks in a new modified LSTM example using a specific `TrainingMaster` subclass called `ParameterAverageTrainingMaster` and also wrapping our network configuration in the `SparkDl4jMultiLayer` object instance to handle the nuances of operating on a Spark cluster.

In a previous section in this chapter, we covered most of the available options for using the `TrainingMaster` variants and the Spark wrapper, so we'll not rehash those details here.

```
//Set up the TrainingMaster. The TrainingMaster controls how learning is
//actually executed on Spark
//Here, we are using standard parameter averaging
//For details on these configuration options, see:
//https://deeplearning4j.org/spark#configuring
int examplesPerDataSetObject = 1;
ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(examplesPerDataSetObject)
    .workerPrefetchNumBatches(2) //Asynchronously prefetch up to 2 batches
    .averagingFrequency(averagingFrequency)
    .batchSizePerWorker(batchSizePerWorker)
    .build();
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, conf, tm);
sparkNetwork.setListeners(Collections.<IterationListener>singletonList(new
    ScoreIterationListener(1)));

//Do training, and then generate and print samples from network
for (int i = 0; i < numEpochs; i++) {
    //Perform one epoch of training. At the end of each epoch, we are
    //returned a copy of the trained network
    MultiLayerNetwork net = sparkNetwork.fit(trainingData);

    //Sample some characters from the network (done locally)
    log.info("Sampling characters from network given initialization \""
        + (generationInitialization == null ? "" : generationInitialization) +
        "\"");
    String[] samples = sampleCharactersFromNetwork(generationInitialization,
        net, rng, INT_TO_CHAR,
        nCharactersToSample, nSamplesToGenerate);
    for (int j = 0; j < samples.length; j++) {
        log.info("----- Sample " + j + " -----");
        log.info(samples[j]);
    }
}
```

Beyond the small number of lines of code to get the example moved over to Spark, we have a similar loop to train the model that uses the same method (`.fit()`) to train the model. The interop with filesystems (e.g., local or HDFS), Spark model communication, and other related functions are all handled under the hood.

Modeling MNIST with a Convolutional Neural Network on Spark

For this example we'll return to the Convolutional Neural Network (CNN) example from [Chapter 5](#) and update it to run on Spark. [Example 9-6](#) shows the code for build-

ing a CNN model with Spark. This example is similar to the MNIST modeling example earlier in this chapter, but this time we're changing the network architecture over to a network more appropriate for image data: the CNN architecture.

Example 9-6. Spark MNIST with a CNN

```
public class MnistExample {
    private static final Logger log = LoggerFactory.getLogger(MnistExample.class);

    public static void main(String[] args) throws Exception {

        //Create spark context, and load data into memory
        SparkConf sparkConf = new SparkConf();
        sparkConf.setMaster("local[*]");
        sparkConf.setAppName("MNIST");
        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        int examplesPerDataSetObject = 32;
        DataSetIterator mnistTrain = new MnistDataSetIterator(32, true, 12345);
        DataSetIterator mnistTest = new MnistDataSetIterator(32, false, 12345);
        List<DataSet> trainData = new ArrayList<>();
        List<DataSet> testData = new ArrayList<>();
        while(mnistTrain.hasNext()) trainData.add(mnistTrain.next());
        Collections.shuffle(trainData,new Random(12345));
        while(mnistTest.hasNext()) testData.add(mnistTest.next());

        //Get training data. Note that using parallelize isn't recommended for real
        //problems
        JavaRDD<DataSet> train = sc.parallelize(trainData);
        JavaRDD<DataSet> test = sc.parallelize(testData);

        //Set up network configuration (as per standard DL4J networks)
        int nChannels = 1;
        int outputNum = 10;
        int iterations = 1;
        int seed = 123;

        log.info("Build model....");
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(seed)
            .iterations(iterations) // Training iterations as above
            .regularization(true).l2(0.0005)
            .learningRate(.01)//.biasLearningRate(0.02)
            .weightInit(WeightInit.XAVIER)
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .updater(Updater.NESTEROVS).momentum(0.9)
            .list()
            .layer(0, new ConvolutionLayer.Builder(5, 5)
                //nIn and nOut specify depth. nIn here is the nChannels and nOut is
                //the number of filters to be applied
```

```

    .nIn(nChannels)
    .stride(1, 1)
    .nOut(20)
    .activation(Activation.IDENTITY)
    .build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
    //Note that nIn need not be specified in later layers
    .stride(1, 1)
    .nOut(50)
    .activation(Activation.IDENTITY)
    .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
.layer(4, new DenseLayer.Builder().activation(Activation.RELU)
    .nOut(500).build())
.layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
    .NEGATIVELOGLIKELIHOOD)
    .nOut(outputNum)
    .activation(Activation.SOFTMAX)
    .build())
.setInputType(InputType.convolutionalFlat(28,28,1)) //See note below
.backprop(true).pretrain(false).build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();

//Create Spark multi layer network from configuration
ParameterAveragingTrainingMaster tm =
    new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
        .workerPrefetchNumBatches(0)
        .saveUpdater(true)
        .averagingFrequency(5)                                //Do 5 minibatch
                                                       //fit operations
                                                       //per worker,
                                                       //then average
                                                       //and
                                                       //redistribute
                                                       //parameters
                                                       //Number of
                                                       //examples that
                                                       //each worker
                                                       //uses per fit
                                                       //operation
        .batchSizePerWorker(examplesPerDataSetObject)
        .build();

```

```

SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, net, tm);

//Train network
log.info("--- Starting network training ---");
int nEpochs = 5;
for( int i=0; i<nEpochs; i++ ){
    sparkNetwork.fit(train);
    System.out.println("----- Epoch " + i + " complete -----");

    //Evaluate using Spark:
    Evaluation evaluation = sparkNetwork.evaluate(test);
    System.out.println(evaluation.stats());
}

log.info("*****Example finished*****");
}
}

```

In the sections that follow, we discuss specific parts of the code in more detail.

Configuring the Spark Job and Loading MNIST Data

To use the Spark execution framework in Java, we need to create a `SparkConf` object and a `JavaSparkContext` object instance. This is where we set up the job to run in local mode or on a Spark cluster.¹²

When we work with Spark, our data generally needs to be stored in an RDD structure. The code snippet that follows demonstrates how to take the raw MNIST data from the iterator we previously used in [Chapter 5](#) and convert the data to `JavaRDD` instances.

```

DataSetIterator mnistTrain = new MnistDataSetIterator(32, true, 12345);
DataSetIterator mnistTest = new MnistDataSetIterator(32, false, 12345);
List<DataSet> trainData = new ArrayList<>();
List<DataSet> testData = new ArrayList<>();
while(mnistTrain.hasNext()) trainData.add(mnistTrain.next());
Collections.shuffle(trainData,new Random(12345));
while(mnistTest.hasNext()) testData.add(mnistTest.next());

//Get training data. Note that using parallelize isn't recommended for
//real problems

```

¹² To run the example locally, run the example as is. The example is set up to use Spark local by default. Note: Spark local should be used only for development/testing. For data parallel training on a single machine (for example, multi-GPU systems), use `ParallelWrapper` (which is faster than using Spark for training on a single machine).

```
JavaRDD<DataSet> train = sc.parallelize(trainData);
JavaRDD<DataSet> test = sc.parallelize(testData);
```

We're collecting all of the MNIST data into memory and then using the Spark context object to create JavaRDDs.

Setting Up the LeNet CNN Architecture and Training

Similar to the LeNet CNN in [Chapter 5](#), we can see the same architecture being configured in the preceding code. For simplicity, we've highlighted this section of code here.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations) // Training iterations as above
.regularization(true).l2(0.0005)
.learningRate(.01)//.biasLearningRate(0.02)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new ConvolutionLayer.Builder(5, 5)
    //nIn and nOut specify depth. nIn here is the nChannels and nOut
    //is the number of filters to be applied
    .nIn(nChannels)
    .stride(1, 1)
    .nOut(20)
    .activation(Activation.IDENTITY)
    .build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
    //Note that nIn need not be specified in later layers
    .stride(1, 1)
    .nOut(50)
    .activation(Activation.IDENTITY)
    .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
    .kernelSize(2,2)
    .stride(2,2)
    .build())
.layer(4, new DenseLayer.Builder().activation(Activation.RELU)
    .nOut(500).build())
.layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
    .NEGATIVELOGLIKELIHOOD)
    .nOut(outputNum)
    .activation(Activation.SOFTMAX)
    .build())
.setInputType(InputType.convolutionalFlat(28,28,1)) //See note below
.backprop(true).pretrain(false).build();
```

We're using the same `MultiLayerConfiguration` we did with the non-Spark example from [Chapter 5](#), and the layers are the same, as well. The major difference is how we control the parallelization of the training loop on Spark, as we'll see in the next section.

Training a DL4J model on Spark is similar to how we did it on a single machine, yet there are some differences. In the code that follows, let's take a look at how the training section of our example code is different from the example in [Chapter 5](#):

```
//Create Spark multi layer network from configuration
ParameterAveragingTrainingMaster tm =
    new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
        .workerPrefetchNumBatches(0)
        .saveUpdater(true)
        .averagingFrequency(5)                                //Do 5
                                                       //minibatch fit
                                                       //operations per
                                                       //worker, then
                                                       //average and
                                                       //redistribute
                                                       //parameters
                                                       //Number of
                                                       //examples that
                                                       //each worker
                                                       //uses per fit
                                                       //operation
        .batchSizePerWorker(examplesPerDataSetObject)
        .build();

SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, net, tm);

//Train network
log.info("--- Starting network training ---");
int nEpochs = 5;
for( int i=0; i<nEpochs; i++ ){
    sparkNetwork.fit(train);
    System.out.println("----- Epoch " + i + " complete -----");

    //Evaluate using Spark:
    Evaluation evaluation = sparkNetwork.evaluate(test);
    System.out.println(evaluation.stats());
}
}
```

The major differences here are (as we've seen in the previous two examples):

- Use of the `ParameterAveragingTrainingMaster` class
- Use of the `SparkDl4jMultiLayer` class

Other than those two changes, the code is not terribly different.



Spark Code Versus Local Code

A useful aspect to DL4J is how creating a model on your local machine and creating a model on Spark is not significantly different. The major differences, as demonstrated throughout this chapter, don't amount to many lines of code.

APPENDIX A

What Is Artificial Intelligence?

Cooper: Hey TARS, what's your honesty parameter?

TARS: 90 percent.

Cooper: 90 percent?

TARS: Absolute honesty isn't always the most diplomatic nor the safest form of communication with emotional beings.

Cooper: Okay, 90 percent it is.

—Scene from the movie *Interstellar*

Artificial Intelligence (AI) is a discipline as old as the study of philosophy itself. It has evolved over time, yet we still struggle to find its place in society, let alone the existential implications for the human race itself. One of the most salient written lines about the beginnings of AI was by author Pamela McCorduck when she wrote that AI began with “an ancient wish to forge the gods.”¹

While McCorduck had high-minded prose for the topic, much of marketing today plays on such aspirational themes yet is actually promoting far simpler functions in terms of business results. Deep learning comes up in the context of discussion of AI regularly, and it becomes difficult to have a directed conversation on the topic.

We added this appendix because the practitioner commonly needs to have a grounded conversation with customers, executives, and managers about what deep learning can do for them and how it fits into the AI landscape. The themes in this appendix are a mixture of a history of the discipline of AI (for context) and discussions we have had with customers and industry peers. We seek to provide you, the practitioner, with the tools to reset the narrative around deep learning, communicate realistic expecta-

¹ McCorduck. 2004. *Machines Who Think* (2nd ed.).

tions with stakeholders in projects, and to put them on ground that will better support their deep learning efforts going forward. Simply put, the narrative around AI has become overhyped and eventually the market will correct.

This appendix is also meant to be fun and thought provoking in a way that stokes the imagination of the researcher or practitioner such that we can still dream but with our feet on the ground. We'll work through some basic definitions, provides a short history of the topic of AI, and then look forward to where this all could go. Hopefully we'll help folks avoid the pitfalls of previous cycles of AI interest and better support their deep learning projects to be more successful through responsibly setting goals and expectations.

The Story So Far

The main topic of this book, deep learning, is constantly attached to the term “Artificial Intelligence” in media and marketing. Definitions are fluid at best and make it difficult to have a discussion on the topic with other practitioners or stakeholders. Marketing departments run with the theme of the current hype cycle. Here are some themes from the not-too-distant past:

- Smart grid
- The cloud
- Big data

When we work in domains like these or in deep learning, as practitioners we need to delineate what is real from what is marketing hype. That begins with understanding the history of a subject and having solid definitions from which to build. Let's begin by reviewing what we consider to be deep learning, and then we'll dive into the topic of defining AI.

Defining Deep Learning

In Chapters 1 and 3, we laid out a working definition for deep learning that is described as neural networks with the following properties:

- More neurons than previous neural networks
- More complex ways of connecting layers
- “Cambrian-like” explosion of computing power to train
- Automatic feature learning

These networks perform the same modeling functions as other machine learning models (regression, classification) but have also been shown to be good at tasks such as the following:

- Generative modeling (e.g., generating art and text)
- Speech-recognition technology
- Image-recognition technology

Another key driving feature of deep learning is that it is able to automatically learn features (as opposed to hand-engineering features) from data in a domain-agnostic fashion. These capabilities of deep learning are driving many of the new technology applications and have stimulated the imagination of many folks beyond technology circles. By itself, however, deep learning does not have higher-level functions such as “automatically understanding the most interesting question to ask a dataset,” let alone any type of sentient operation.

Defining Artificial Intelligence

The history of AI is fraught with myths, stories, and overzealous marketing departments trying to tap into the latest technology narrative. To define AI, we need some context around the history of the study of intelligence, modern arguments, and how the discipline has evolved over time.

Using that line as our embarkation point, let’s explore how the discipline got its start and then evolved as an industry over the past 60 years.

The study of intelligence

The study of intelligence was formally initiated in 1956 at Dartmouth yet is at least 2,000 years old. The field is based on understanding intelligent entities and studying topics such as these:

- Seeing
- Learning
- Remembering
- Reasoning

These topics are components of what we’d consider intelligent function to the capacity we have to understand intelligence (and this is a relative perspective). We can find the study of intelligence throughout history if we look for it. The list that follows presents a few of the building blocks of intelligent study over time:

Philosophy (400 BC)

Philosophers began to suggest the mind as a mechanical machine that encodes knowledge in some form inside the brain.

Mathematics

Mathematicians developed the core ideas of working with statements of logic along with the groundwork for reasoning about algorithms.

Psychology

This field of study is built on the ideas that animals and humans have a brain that can process information.

Computer science

Practitioners came up with hardware, data structures, and algorithms to support reverse engineering basic components of the brain.

The study and application of AI techniques we see today are based on these fundamentals. We typically see the study of AI broken into a focus on either behaving or thinking in simulated intelligent systems. We see these applied in areas such as machine learning applications, basic knowledge systems, and game playing (e.g., Chess and Go).

However, there are limitations to our implementations of intelligent study. There are still no good models of higher-order brain functions such as consciousness. Science has also yet to determine where in the brain consciousness resides. This leads some to question if consciousness is even a real function of the brain, but we'll leave that to the philosophers and computer scientists to debate.



For Further Study

One of the best books written on the subject of AI (if not the best) is Stuart Russell and Peter Norvig's *Artificial Intelligence: A Modern Approach*. We can't recommend this book enough for you to get a more complete idea of the depth and history of AI.

Cognitive dissonance and modern definitions

When we're dealing with a topic such as AI that ties into so many core definitions society depends on, we naturally find dissonance around setting ground truth to work from. [Beau Cronin writes:](#)

Like the Internet of Things, Web 2.0, and big data, AI is discussed and debated in many different contexts by people with all sorts of motives and backgrounds: academics, business types, journalists, and technologists. As with these other nebulous technologies, it's no wonder the meaning of AI can be hard to pin down; everyone sees what they want to see.

Part of the problem with viewpoints and defining intelligence is that we weave liberally into defining consciousness and then by extension more philosophical (e.g., "what is consciousness") and religious topics (e.g., "what is a soul?"). We're touching on some complex territory at this point and any discussion about the definition of the soul is fraught with complications. Our best maps today for the definition of intelligence are covered in regions of "Here Be Dragons." Given that we don't understand natural intelligence, it is even more difficult to define the artificial variant.

Dr. Jason Baldridge wrote on the topic of AI and machine learning and talked about how there are conflicting meanings in play around the topic:

Regardless of any nuanced technical definition of AI, I'm pretty sure that when the public hears "artificial intelligence," they think of conscious nonbiological entities that interact with humans much as we interact with each other.

They don't think of an expert system that can analyze a complex domain-specific problem and provide interesting courses of action, or machine learning algorithms that find fascinating patterns in heaps of data.

Despite this, the general public seems to find it all too easy to mentally close the gap between these two very different levels of technological and scientific accomplishment on the spectrum of AI-related work.

Dr. Baldridge goes on to define the difference between deep learning and a full artificial model of the biological brain:

Despite all this progress, and for better or for worse, these are still far from sentient machines. Deep learning is inspired by the functioning of human neurons, but as far as I'm aware, artificial neural networks as yet have nothing like the architecture of meat-based intelligence

So, we struggle with these definitions because they are complicated and touch on many topics from many viewpoints. Let's take a step toward a better definition by segmenting the topic and breaking these segments down into simpler topics.

Francois Chollet recently made the following [salient comment on Twitter](#):

[A]rtificial intelligence is a poorly defined thing, to which many people attribute wildly unrealistic abilities. It's a recipe for trouble.

And then a further [tweet](#):

Part of the problem is that some companies and journalists are hyping it up, blurring the line between sci-fi and reality. Because it sells.

Chollet goes on to say that [we should "define" what we're talking](#) about:

When you talk about "AI", *define* what you are talking about. Make explicit what it can do, and what it can't do. Avoid brain analogies.

This is sound advice and the industry needs to do a far better job of locking down these definitions.

What AI is not. Folks who claim machine learning to be AI do the entire computer science industry a disservice. Machine learning is classification and regression and in no way matches up to ephemeral aspirations of an all-knowing, self-aware system that can help the reader with their marketing problem. As Francois Chollet mentioned earlier, it's just best (for now) to avoid brain analogies.

Many times AI is marketed as an application with all of the answers. It will not, at least not anytime soon.

Moving the goal posts. Psychologists habitually have had disdain for the metaphor of the human brain as a computer. In a 2016 article, Robert Epstein states:

No matter how hard they try, brain scientists and cognitive psychologists will never find a copy of Beethoven's 5th Symphony in the brain—or copies of words, pictures, grammatical rules or any other kinds of environmental stimuli.

Unfortunately Dr. Epstein has not seen the renders in Chapter 4 of Convolutional Neural Network (CNN) filter renders. His central argument in the article is stated as:

Your brain does not process information, retrieve knowledge, or store memories. In short: your brain is not a computer.

This isn't a new sentiment and has been echoed, by and large, by the disciplines outside computer science for the past 60 years. As stated in Russell and Norvig's book on AI:

The intellectual establishment, by and large, preferred to believe that a "machine can never do X."

They go on to demonstrate examples in which AI researchers have systematically responded by demonstrating one X after another. Thus, the study of AI and definitions for the discipline have long suffered from the industry "moving the goal posts" with respect to what they really mean by "Artificial Intelligence."

Segmenting the definitions of AI. It's useful to break down the different viewpoints of how folks talk about AI today and list them. A good writeup by Beau Cronin uses the following four major segmented definitions of AI:

- AI as interlocutor
 - HAL, Siri, Cortana, Watson
 - Conversational intelligence
 - Limited reasoning
- AI as android
 - Machine in the form of a humanoid
 - AI as mechanically embodied
 - Examples would be The Terminator or C3PO
 - Similar to the interlocutor but in a humanoid body
- AI as the reasoner
 - Early AI pioneers were drawn to more refined and high-minded tasks—playing chess, solving logical proofs, and planning complex tasks
 - Still struggle with tasks simple for children
- AI as the big data learner
 - More recent definition
 - Seeing many people talk about building "AI models"

Let's now look at these segmented definitions with a critical eye.

Critical commentary on segments. AI as the interlocutor and AI as the big data learner are both recent definitions due to the incorporation of many machine learning techniques into engineered commercial products. AI as the interlocutor can perform basic functions based on voice recognition. It's the combination of voice-to-text machine learning (or deep learning) and Natural Language Processing (NLP) technology to determine what the user wants to accomplish. AI as the interlocutor has limited reasoning capabilities because it typically relies on a separate system to send the voice-to-text and NLP-processed results as input. This separate system is many times as basic as a classic rule-base system or "expert system."

Even though users might be initially entertained with conversation by the system and even fooled by its "intelligence," they quickly realize the limitations of the interactions. Ultimately, AI as the interlocutor is a well-engineered combination of machine learning techniques that over time became just good enough to progressively be wired further into useful consumer products.

AI as the android is an interesting embodiment of the concept yet is ultimately dependent on a network of machine learning subsystems just as the interlocutor is. AI as the reasoner is a classical implementation of AI yet has plateaued in recent years with respect to levels of industry product integration interest. It continues to be a core component in intelligent systems that wire together multiple components to produce value, as in the interlocutor example.

"AI as the big data learner" is a troublesome use of the term that has gained popularity in the past few years (2010–2015). Many times, a marketing department will rebrand a product's use of a basic machine learning technique over customer data as "Artificial Intelligence." Worse, other times the product is doing basic business intelligence functions and is also lumped into this category. The practice of machine learning (or deep learning) alone should not be considered a type of AI. It is, however, a useful subsystem of an intelligent system.



Showing Restraint When Marketing Deep Learning

You should show restraint in labeling a machine learning model—deep learning model or not—as "Artificial Intelligence." Overselling capabilities early might attract funding but will hamper your project in the longer term.

A fifth aspirational definition of AI. Another way to frame the question of "What is Artificial Intelligence?" is to ask another question. If we look at this question from the viewpoint of "what would definitively end the debate of 'What is AI'?"

If we were presented with a transcendent, conscious, self-aware intelligence that understood our world (and data) far better than any human, we'd probably call it "true Artificial Intelligence." Or, an alien.

Unfortunately chasing the mirage of AI is what brings on the unrealistic expectations that always crash down on the industry no matter how much tangible progress was made.

The AI winters

The AI industry has experienced multiple periods of up and down interest and funding. These down periods of interest have been the result of the sector being unrealistically overhyped followed by a cycle of predictably underwhelming results. This down period is referred to as an "AI winter" and involves cuts in academic research funding, reduced venture capital interest, and stigma in the marketing realm around anything connected to the term "artificial intelligence."

The result of these cycles is that the good technical advances (e.g., voice recognition or optical character recognition) is rebranded and is integrated into other products.

AI Winter I: (1974–1980). The lead-up to the first **AI winter** saw machine translation fail to live up to the hype. Connectionism (neural networks) interest waned in the 1970s, and speech understanding research overpromised and underdelivered.

In 1973, DARPA cut academic research in the AI domain. The Lighthill report in the United Kingdom harshly criticized the field and research funding was further curtailed.

AI Winter II: Late 1980s. In the late 1980s and early 1990s, there was overpromotion of technologies such as expert systems and LISP machines, both of which failed to live up to expectations. The Strategic Computing Initiative canceled new spending at the end of this cycle. The fifth-generation computer also failed to meet its goals.

The common patterns of AI Winters

The common patterns of these winters see the industry being overhyped by a series of promising successes. After conditions percolate enough we see the hype reach a nadir and the money for academic and industrial research pour into the AI domain. A few real projects based on solid technology reach at least part of their goals and solve real problems. Most of the marketed promises go unfulfilled, however, and the Trough of Disillusionment looms.

Winter kills the weak.

A few interesting applications slowly emerge from the trough, are rebranded (“voice recognition”), and are integrated into other projects as features, generally under the “latent intelligence” class of improvements. We’ve seen this with the following:

- Informatics
- Machine learning
- Knowledge-based systems
- Business rules management
- Cognitive systems
- Intelligent systems
- Computational intelligence

The name change might be partly because they consider their field to be fundamentally different from AI. It is also true that the new names help to procure funding by avoiding the stigma of the false promises attached to the name “Artificial Intelligence.”

Here’s an interesting note from an AI meeting in the 1980s:

At the meeting, Roger Schank and Marvin Minsky—two leading AI researchers who had survived the “winter” of the 1970s—warned the business community that enthusiasm for AI had spiraled out of control in the ’80s and that disappointment would certainly follow. Three years later, the billion-dollar AI industry began to collapse.

What Is Driving Interest Today in AI Today?

Three major contributors are driving interest in AI today:

1. The big jump in computer-vision technology in the late 2000s
2. The big data wave of the early 2010s
3. Advancements in applications of deep learning by top technology firms

In 2006, Geoff Hinton and his team at the University of Toronto published a key paper on Deep Belief Networks (DBNs).² This provided the industry with a spark of creativity on what could possibly improve the state of the art. We’ve seen a tsunami of deep learning publications at top journals over the succeeding decade. These publications began improving top scores for accuracy in many areas beyond just computer vision, and deep learning took over the applied machine learning landscape in short order.

Large web firms such as Google, Facebook, and Amazon all watch top journals for the best ideas. These firms saw the developments by Yann LeCun, Hinton, and others and began implementing these ideas in their own pipelines. These new applications (e.g.,

² Hinton, Osindero, and Teh. 2006. “A fast learning algorithm for deep belief nets.”

better face detection, or Alexa at Amazon) were widely recognized in the technology media.

In the mid-2000s much of the storage and ETL technology developed on the West Coast by the large web firms began to be open source as projects such as Hadoop and MongoDB.

Just as these web firms (Google, Yahoo!, etc.) had ramped up their storage and ETL systems, they were then building out new machine learning and deep learning techniques to better take advantage these new and larger datasets.

Traditional *Fortune 500* enterprise companies were bringing online large distributed systems in the early 2010s to hold their growing transactional datasets. These enterprise companies tend to follow what the West Coast web firms do by about 5 to 10 years. This has given rise to an interest in deep learning in the *Fortune 500* and in systems that allow the traditional enterprise to better exploit their investments in big data.

If we combine the aforementioned three factors and then mix in the very public successes of projects such as Watson (winning *Jeopardy*), AlphaGo (winning Go), and Google's self-driving cars, we create an environment in which enthusiasm outstrips the reality of the road ahead.

It's high tide in coverage and enthusiasm around AI. Unfortunately, in these types of cycles, we also see the tide go back out eventually. There are real applications using complex datasets for deep learning. Here are just some of these applications:

- Healthcare (e.g., predicting patient length of stay)
- Retail (e.g., analyzing the shopping experience)
- Telecom/Financial Services (e.g., analyzing transactions for fraudulent patterns)

We've touched on some of the aforementioned use cases (and more) in this book. When you, as practitioner, promote deep learning and AI we recommend finding real use cases such as these and standing on "solid ground." We mention solid ground here metaphorically because eventually the tide will go out and we hope our fellow practitioners will have something to stand on when it does.

Winter Is Coming

Deep learning in and of itself has been grounded in reality over the course of this book. It is a framework for performing industry-leading neural network modeling on complex data types. Deep learning, by itself, would not fulfill our aforementioned fifth aspirational definition of AI, so we don't have much to worry about on that front.

We are seeing systems marketed in 2016 as “Artificial Intelligence for X,” which are plainly using basic machine learning. AlphaGo was a tremendous advancement in game playing, but as we saw with DeepBlue and Chess, game-playing advances do not always translate easily to business use cases.³

Marketing departments are setting up similar scenarios as occurred in the previous two AI winters, unfortunately. The coals of the real advances in the domain, as in previous winters, will keep the true enthusiasts and hardcore researchers warm during the cold of the coming third AI winter.

³ *Jeopardy* does seem to be a “solved problem,” however.

RL4J and Reinforcement Learning

Ruben Fiszel

Preliminaries

We begin this appendix with an introduction to *reinforcement learning*, followed by a detailed explanation of Deep Q-Networks (DQNs) for pixel inputs, and then we conclude by showing you an RL4J example. Let's begin with a look at the core concepts of reinforcement learning.

Reinforcement learning is an exciting area of machine learning. It is, basically, the learning of an efficient strategy in a given environment. Informally, this is very similar to *Pavlovian conditioning*: you assign a reward for a given behavior, and, over time, the agents learn to reproduce that behavior in order to receive more rewards.

Markov Decision Process

Formally, an environment is defined as a *Markov Decision Process* (MDP). Behind this scary name is nothing other than the combination of (5-tuple):

- A set of states S (e.g., in chess, a state is the board configuration)
- A set of possible actions A (in chess, every possible move in every configuration possible; e.g., e4–e5).
- The conditional distribution $P(s'|s,a)P(s',a)$ of the next state, given a current state and an action. (In a deterministic environment like chess, there is only one state s' with probability 1, and all the others with probability 0. Nevertheless, in a stochastic (involving randomness, like a coin toss) environment, the distribution is not as simple.)
- The reward function of transitioning from state s to s' : $R(s,s')$ (e.g., in chess, +1 for a final move that leads to a victory, -1 for a final move that leads to a defeat, 0 otherwise).

- The discount factor: $\gamma\gamma$. This is the preference for present rewards compared to future rewards (**a concept very common in finance**).



Using the Set of Action A_s

It is usually more convenient to use the set of Action A_s , which is the set of available moves from a given state, than the complete set A .

A_s : elements a in A such that $P(s'|s,a) > 0$.

The Markov property (see [Figure B-1](#)) is memoryless. After you reach a state, the past history (the states visited before) should not affect the next transitions and rewards; only the present state matters.

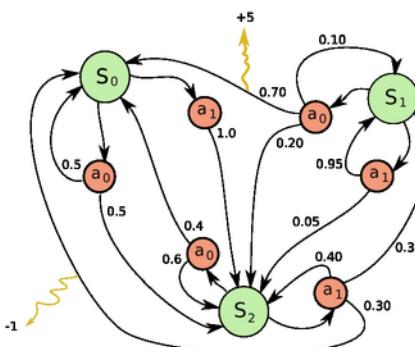


Figure B-1. Schema of an MDP

Terminology

Before we proceed any further, let's take a moment define some common words and terms:

Final/terminal states

The states that have no available actions are final/terminal states.

Episode

An episode is a complete play from one of the initial states to a final state:

$$s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_n$$

Cumulative reward

The cumulative reward is the discounted sum of reward accumulated throughout an episode:

$$R = \sum_{t=0}^n \gamma^t r_{t+1}$$

Policy

A policy is the agent's strategy to choose an action at each state. It is denoted by π .

Optimal policy

The optimal policy is the theoretical policy that maximizes the expectation of cumulative reward. From the definition of expectation and the law of large numbers, this policy has the highest average cumulative rewards given sufficient episodes. This policy might be intractable.

The objective of reinforcement learning is to train an agent such that the agent learns a policy as close as possible to the optimal policy.

Different Settings

« Qui peut le plus peut le moins » (He who can do the greater things, can do the lesser things)

Model-Free

The conditional distribution and the reward function constitute the model of the environment. In a game of backgammon, we know the model (each possible transition is decided by the known dice distribution, and we can predict each reward from transition without realizing them [because we can calculate the new value of the board]). The algorithm **TD-gammon** uses that fact to learn the V function (see “**Q-Learning**” on page 421).

Some reinforcement learning algorithms can work without being given the model. Nevertheless, to learn the best strategy, they additionally need to learn the model during the training. This is called *model-free reinforcement learning*. Model-free algorithms are very important because a large majority of real-world complex problems fall in that category. Furthermore, model-free is simply an additional constraint. It is simply more powerful because it is a superset of model-based reinforcement learning.

Observation Setting

Instead of being given access to the state, you might be given access to only a partial observation of the state. It is the same idea behind a *Hidden Markov Chain*. This is the difference between the partial and fully observed setting. For instance, our field of vision is a very partial observation of the full state of the universe (the position and

energy of every particle in the universe). Fortunately, the partial observation setting can be reduced to a fully observed setting with the use of a history (the state becomes an accumulation of previous states).

Nevertheless, it is most common to not accumulate the entire history. Either only the last h observations are stacked (in a windowed fashion) or you can use a Recurrent Neural Network to learn what to keep in memory and what to forget (that is essentially how a Long Short-Term Memory [LSTM] works).

Abusing the language slightly for consistency purposes with the existing notation, history (even truncated ones) will also be called “state” and also symbolized as S_t .

Single-Player and Adversarial Games

A single-player game has a natural translation into an MDP. The states represent the moment during which the player is in control. The observations from those states are all the information accumulated between states (e.g., as many pixel frames as there are in between frames of control). An action is all the available commands at the disposal of the player (in the game *Doom*, go up, right, left, shoot, etc.).

Reinforcement learning also can be applied to adversarial games by self-play: the agent plays against itself. Often in this setting, there exists a **Nash equilibrium** such that it is always in your interest to play as if your opponent were a perfect player. This makes sense in chess, for example. Given a specific board configuration, a good move against a chess master would still be a good move against a beginner. Whatever is the current level of the agent, by playing against himself, the agent stills gains information about the quality of his previous moves (seen as good moves if he won, bad moves if he lost).

Of course, the information, which is a gradient in the context of a neural network, is of higher quality if he played directly against a very good agent from the beginning. But it is really mind-blowing that an agent can learn to increase his level of play by playing against himself, an agent of the same level. That is actually the method of training employed by **AlphaGo** (the Go agent from DeepMind that beat the World Champion). The policy was bootstrapped (initially trained) on a dataset of master moves. It then used reinforcement learning and self-play to further increase the level (quantified with Elo). In the end, the agent became better than the policy it was learning from the original dataset. After all, it beat the master above all the masters. To compute the final policy, the AlphaGo Team used their policy gradient in combination with a Monte-Carlo Search Tree on a massive amount of computation power.

This setting is a bit different from learning from pixels. First, because the input is not as high-dimensional, **the manifold** is a lot closer to its embedding space. Nevertheless, a convolutional layer was still used in this case to make efficient use of the local-

ity of some subgrid board patterns. Second, because AlphaGo is not model-free (it is deterministic).

Q-Learning

I am no friend of probability theory, I have hated it from the first moment when our dear friend Max Born gave it birth. For it could be seen how easy and simple it made everything, in principle, everything ironed and the true problems concealed.

—Erwin Schrödinger

From Policy to Neural Networks the following

Our goal is to learn the optimal policy π^* that maximizes:

$$E[R_0] = E\left[\sum_{t=0}^n \gamma^t r_{t+1}\right]$$

Let's introduce an auxilliary function:

$$V_\pi(s) = E\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_n \mid s_t = s, \text{policy followed at each state is } \pi\}$$

This is the expected cumulative reward from a state s following the policy π . Let's consider an oracle, such as this:

$$V_{\pi^*}(s)$$

This gives us the V function of the optimal policy. From it, we could retrieve the optimal policy by defining the policy that, from among all available actions at the current state, chooses the action that maximizes the expectation of $V_{\pi^*}(s)$. This is a greedy behavior. The optimal policy is the greedy policy with respect to V_{π^*} :

$$\pi^*(s) \text{ chooses a s.t } a = \arg \max_a [E_\pi(r_t + \gamma V(s_{t+1}) \mid s_t = s, a_t = a)]$$

If you were very attentive, something probably sounds wrong here. In the model-free setting, we cannot predict the after-state s_{t+1} from s_t because we ignore the transition model. Even with that oracle, our model is still not computable!

To solve this very annoying issue, we are going to use another auxiliary function, the Q -function:

$$Q_{\pi^*}(s, a) = E_\pi[r_t + \gamma V_{\pi^*}(s_{t+1}) \mid s_t, a_t = a]$$

In a greedy setting, we have the following relationship:

$$V_\pi(s_t) = \max_a Q_\pi(s_t, a)$$

Now, suppose that instead of the V oracle, we have the Q oracle. We can now redefine π^* , as follows:

$$\pi^*(s) \text{ chooses } a \text{ s.t } a = \max_a [Q_{\pi^*}(s, a)]$$

No more uncomputable expectations, neat.

Nevertheless, we have only moved the expectation from outside to inside the oracle. And, unfortunately, oracles do not exist in the real world.

The trick here is that we have reduced an abstract notion that is a policy into a numerical function that might be relatively “smooth” (continuous) thanks to the expectation. Fortunately for us, there is one weapon at our disposal to approximate such complex functions: neural networks.

Neural networks are universal function approximators. They can approximate any continuous differentiable function. However, they can become stuck in local extrema, and many proofs of convergence from reinforcement learning are not valid anymore when throwing neural networks in the equation. This is because their learning is not as deterministic or boundable as their tabular counterparts. Nonetheless, in most cases, with the right hyperparameters, **they are unreasonably powerful**. Using deep learning with reinforcement learning is called *deep reinforcement learning*.

Policy Iteration

At this point, your knowledge of machine learning and plain old common sense is likely telling you that there is still something missing about our approach. Neural networks can approximate functions that already have labels. Unfortunately for us, oracles are not summonable, so we will need to get our labels another way.

This is where the magic of Monte Carlo comes in. Monte Carlo methods rely on repeated random sampling to calculate an estimator. (A famous example is the [pi calculation](#).)

If we play randomly from a given state, the better states should get better rewards on average (thank you [law of large numbers](#)). So, without knowing anything about the environment, you can gather some information about the expected value of a state. For instance, at poker, better hands will win more often on average than lesser hands even when every decision is taken randomly. The Monte Carlo Search Tree is

based on this property, as well (shocking, isn't it?). This is a phase of exploration that leads to unsupervised learning and enables us to extract meaningful labels.

More formally, given a policy π , a state s , and an action a , to get an approximation of $Q(s,a)$, we sample it according to its definition:

$$Q_\pi(s, a) = E[r_t + \gamma r_{t+1} + \dots + \gamma^n r_n \mid s_t = s, a_t = a]$$

In plain English, we can get a label for $Q_\pi(s, a)$ by playing a sufficient number of times from s according to policy π .

From an aggregate of signals (Figure B-2):

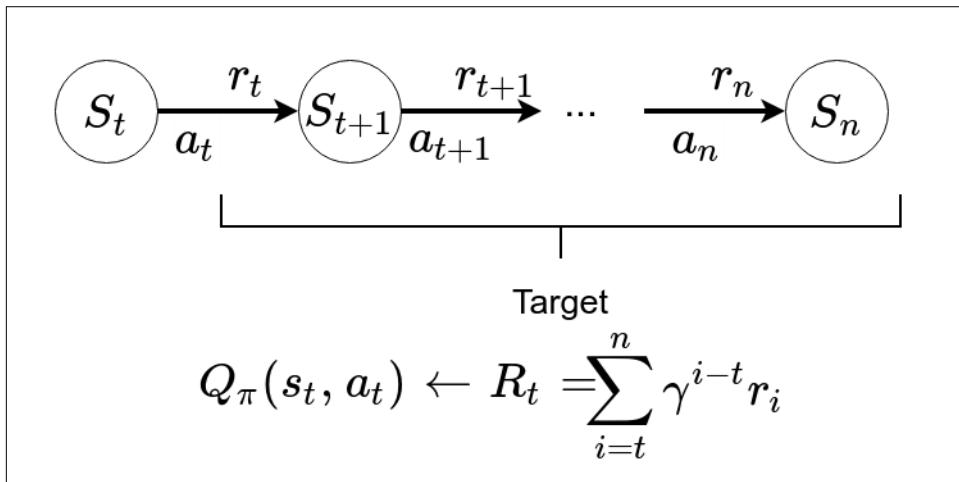


Figure B-2. One signal

The actual learning is done by Stochastic Gradient Descent (SGD), using the labels in batches. The gradient is the standard mean squared error (MSE) one such that the td-error gets minimized at each iteration.

We use the MSE loss function (L2 loss) with a learning rate of α and apply SGD (on a batch of size 1) of:

$$[Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)]]$$

(s_t, a_t) is the input, $Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)]$ is the label (e.g., target).



Even if we use MSE, there is no square in the formula because the loss is applied afterward on the difference of the expected output $Q_\pi(s_t, a_t)$ and the label $\alpha[R_t - Q_\pi(s_t, a_t)]$.

Repeat many times, sampling from π :

$$Q_\pi(s_t, a_t) \leftarrow E_\pi[R_t] = E_{s_t, a_t, \dots, s_n \sim \pi} \left[\sum_{i=t}^n \gamma^{i-t} r_i \right]$$

Figure B-3 can converge to the right expectation.

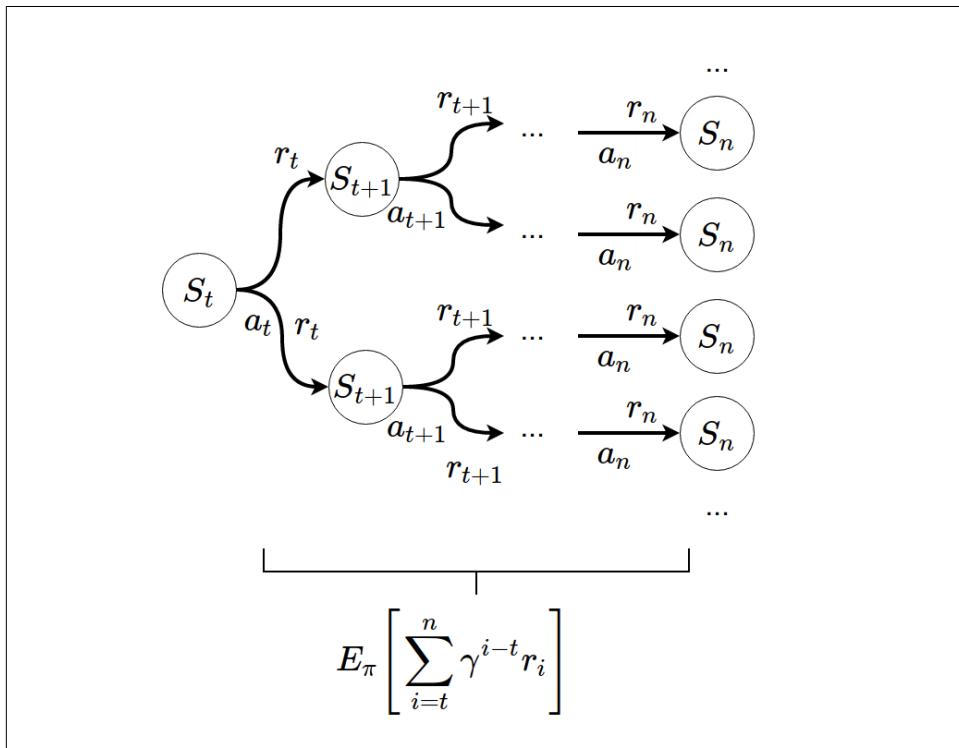


Figure B-3. Many signals

So, we now can design a naïve prototype of our learning algorithm (in Scala, but it is intelligible without any Scala knowledge), which you can see in [Example B-1](#).

Example B-1. RL4J Prototype in Scala

```
//A randomly uninitialized neural network
val neuralNet: NeuralNet
```

```

//Iterate until you reach max epoch
for (t <- (1 to MaxEpoch))
    epoch()

def epoch() = {

    //pick a random state and action
    val state = randomState
    val action = randomAction(state)

    //transition to a new state, initialize the reward
    var (new_state, accuReward) = transition(state, action)

    //play until terminal state and accumulate the reward
    accuReward += playRandomly(state)

    //Do SGD over input and label!
    fit((state, action), accuReward)
}

// MDP specific, return the new state and the reward
def transition(state: State, action: Action): (State, Double)

//return a randomly sampled state among all the state space
def randomState: State

//play until terminal state
def playRandomly(state): Double = {
    var s = state
    var accuReward = 0
    var k = 0
    while (!s.isTerminal) {
        val action = randomAction(s)
        val (state, reward) = transition(s, action)
        accuReward += Math.pow(gamma, k) * reward
        k += 1
        s = state
    }
    accuReward
}

//choose a random action among all the available actions at this state
def randomAction(state: State): Action =
    oneOf(state.available_action)

//helper function, pick one among

```

```

def oneOf(seq: Seq[Action]): Action =
  seq.get(Random.nextInt(seq.size))

//How it would be roughly done with DL4J
def fit(input: (State, Action), label: Double) =
  neuralNet.fit(toTensor(input), toTensor(label))

//return an INDArray from ND4J
def toTensor(array: Array[_]): Tensor =
  Nd4j.create(array)

```

There are multiple issues: this should work, but it's terribly inefficient. We are playing a full game with n state and n actions for a single label and that label might not be very meaningful (if the interesting trajectories are difficult to reach at random).

Exploration Versus Exploitation

Exploring at random, the environment will converge to the optimal policy, but this is guaranteed only after an almost infinite time: you will need to visit every possible trajectory at least once. (A trajectory is the ordered list of all of the states visited and actions chosen during an episode.) Considering how many states and branches there are, it is impossible. The branching issue is the reason why Go is so much more difficult than chess. In the real world, we do not have infinite time (and time is money).

Thus, we should exploit the past information and our learning of it to focus our exploration on the most promising possible trajectories. We can achieve this through different ways, and one of them is ϵ -greedy exploration. ϵ -greedy exploration is fairly simple. It is a policy that chooses an action at random with odd ϵ or the best action as deemed by our current policy with odd $(1-\epsilon)$. Usually ϵ is annealed over time to privilege exploitation over exploration after enough exploration. This is a trade-off between exploration and exploitation.

With each new piece of information, our actual Q functions become more accurate about the present policy, and the exploration is focused on better paths. The policy based on our new Q function becomes better (because Q is more accurate), and the ϵ -greedy exploration reaches better paths. Focused on those better paths, our Q function can explore yet better parts and must update its returns according to the new policy. As [Figure B-4](#) illustrates, this is an iterative cycle that enables convergence to the optimal policy. Unsurprisingly, it's called *policy iteration*. Unfortunately, the convergence can take infinite time and is not even guaranteed when Q is approximated by neural networks. Nevertheless, impressive results can make up for the lack of formal convergence proofs.

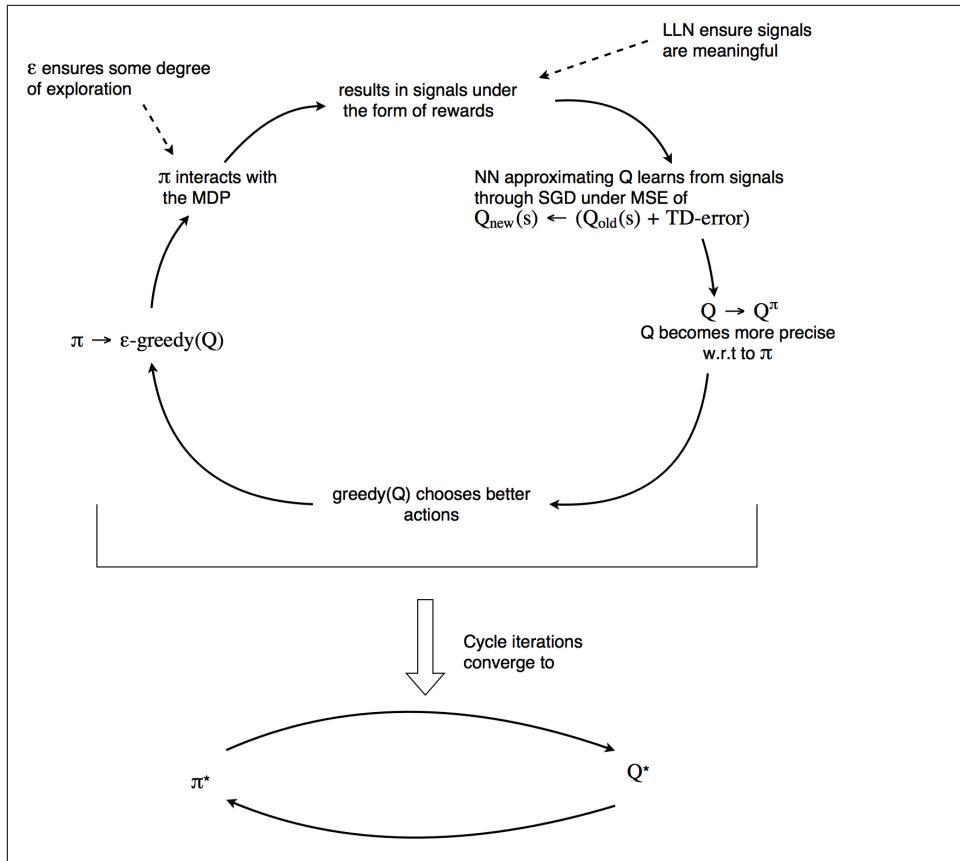


Figure B-4. Policy iteration

This algorithm also requires you to be able to sample the states in a “good” manner: It should be proportionally representative of the states that are usually present in a game (or at least the kind of game at the targeted agent’s level).

Bellman Equation

We can transform the Q equation into a Bellman equation, as demonstrated here:

$$\begin{aligned}
 Q_\pi(s, a) &= E[r_t + \gamma r_{t+1} + \dots + \gamma^n r_n \mid s_t = s, a_t = a] \\
 &= E[r_t + \gamma r_{t+1} + V(s_{t+1}) \mid s_t = s, a_t = a] = E[r_t + \gamma r_{t+1} + \dots + \gamma \max_{a'} Q(s_{t+1}, a')] \\
 &\quad \mid s_t = s, a_t = a
 \end{aligned}$$

As in the Monte Carlo method, we can do many updates of Q .

MSE:

$$Q_{\pi}(s_t, a_t) \leftarrow Q_{\pi}(s_t, a_t) + \alpha \left[\underbrace{r_t + \max_a Q_{\pi}(s_{t+1}, a)}_{\text{target}} - Q_{\pi}(s_t, a_t) \right] \underbrace{\left[\dots \right]}_{\text{TD-error}}$$

TD-error is the temporal difference error. Indeed, we are actually calculating the difference between what the Q approximation expects in the future plus the realized reward and its present value as evaluated by the neural network.

The Bellman equation only make sense with some boundary conditions. If s is terminal:

$$V(s) = 0$$

and for any a :

$$Q(s_{t-1}, a) = r_t$$

The states near the terminal states are the first to converge because they are closer in the chain to the “true” label, the known boundary conditions. In Go or chess, reinforcement learning is applied by assigning $+1$ to the transitions that lead to a final winning board (respectively -1 for a losing board) and 0 otherwise. It diffuses the Q -values by finding a point between the two extremes $[-1; 1]$. A transition with a Q -value close to 0 represents a transition leading to a balanced board. A transition with a Q -value close to 1 represents a near certain victory.

It could be surprising that the moves do not have only -1 and 1 values (because deviating from the optimal path should be fatal). One interesting aspect of calculating Q -values is the realization that in many games/MDP, no mistake in itself is ever really fatal. It is the accumulation of them that really kills you. AI is full of life lessons. Moreover, the expected accumulated reward space is a lot smoother than often thought. One possible explanation is that expectations always have an average effect: an expectation is nothing more than a weighted average with probabilities as weights. Furthermore, gamma being $<1<1$, the very long-term effects are not too preponderant. Isn’t it exciting to be able to calculate directly the odds of winning a game for every transition?

As long as we sample sufficient enough transitions near the terminal states, Q -learning is able to converge. The incredible power of deep reinforcement learning is that it will be able to generalize its learning from visited states to unvisited states. It should be able to understand what is a balanced or winning board even if it has never seen it before. This is because the network should be able to abstract patterns and

understand the strength of an action based on previously seen pattern (e.g., shoot an enemy when recognizing its form).



Offline and Online Reinforcement Learning

To learn more about the differences between online and offline reinforcement learning, see the [excellent post from Kofzor](#).

Initial State Sampling

In a one-player setting (like an Atari game), we do not actually need to learn to play well in every situation (Although, if we did, that would show that we would have reached a very good level of generalization). We need only to learn to play efficiently from the states that our policy encounters. Thus, we can sample from states that are simply reachable by playing with our current policy from an initial state. This enables us to sample directly from a played episode by our agent.

Q-Learning Implementation

At this stage, we can design a naïve prototype of our Q-learning. [Example B-2](#) presents the code to do this.

Example B-2. Naïve prototype of our Q-learning in Scala

```
def epoch() = {  
    //sample among the initial state space  
    //(often unique state)  
    var state = initState  
  
    //while the state is not terminal,  
    //play an episode and do a Q-update at each transition  
    while(!state.isTerminal) {  
  
        //sample action from eps-greedy policy  
        val action = epsilonGreedyAction(state)  
  
        //interaction with the environment  
        val (nextState, reward) = transition(state, action)  
  
        //Q-update  
        update(state, action, reward, nextState)  
  
        state = nextState  
    }  
}
```

```

//Our Q-update as explained above
def update(state: State, action: Action, reward: Double, nextState: State) = {
    val target = reward + maxQ(nextState)
    fit((state, action), target)
}

//the eps-greedy policy implementation
def epsilonGreedyAction(state: State) = {
    if (Random.float() < epsilon)
        randomAction(state)
    else
        maxQAction(state)
}

//Retrive max Q value
def maxQ(state: State) =
    actionsWithQ(state).maxBy(_.value)

//Retrive action of the max Q value
def maxQAction(state: State) =
    actionsWithQ(state).maxBy(_.value).action

//return a list of actions and the q-value of their transition from the state
def actionsWithQ(state: State) = {
    val stateActionList = available_actions.map(action => (state, action))
    available_actions.zip(neural_net.output(toTensor(state_action_list)))
}

def initState: State

```

Modeling Q(s,a)

Instead of having $a|a$ as an additional input of the neural net combined with the state, the state is the only input and the output contains the Q-value of every action possible. This makes sense only when the available actions are consistent across the full episode (otherwise, the neural output layer would need to be different at each state). It is many times solvable by having the full set AA of actions as output and ignoring the impossible actions (some papers put the target of impossible actions at 0).

Experience Replay

There is one issue with using a neural network as the Q approximator. The transitions are very correlated. This reduces the overall variance of the transition. After all, they are all extracted from the same episode. Imagine if you had to learn a task without any memory (not even short-term); you would always optimize your learning based on the last episode.

The Google DeepMind research team used experience replay, which is a windowed buffer of the last N transitions (N being a million in the original paper) with DQN,

and greatly improved their performance on Atari. Instead of updating from the last transition, you store it inside the experience replay and update from a batch of randomly sampled transitions from the same experience replay.

`epoch()` becomes:

```
def epoch() = {  
  
    //sample among the initial state space  
    // (often unique state)  
    var state = initState  
  
    //while the state is not terminal,  
    //play an episode and do a Q-update at each transition  
    while(!state.isTerminal) {  
  
        //sample action from eps-greedy policy  
        val action = epsilonGreedyAction(state)  
  
        //interaction with the environment  
        val (nextState, reward) = transition(state, action)  
  
        //store transition (Exp Replay is just a Ring buffer)  
        expReplay.store(state, action, reward, nextState)  
  
        //Q update in batch  
        updateFromBatch(expReplay.getBatch())  
  
        state = nextState  
    }  
}
```

Compression

ND4J, the tensor library of DL4J, does not support as first-class type `uint8`. However, pixels in grayscaling are encoded with that precision. To avoid wasting too much space on memory, `INDArray` were compressed to `uint8`.

Convolutional Layers and Image Preprocessing

Convolutional layers are layers that are excellent at detecting local patterns in images. For pixels, it is used as a processor that is required to reduce the dimension of the input into its real manifold. Given the proper manifold of observations, the decision becomes much easier.

Image processing

You could feed the neural network the RGB directly, but then the network would have to also learn that additional pattern. It seems like the brain is hardwired to com-

bine colors (fortunately!). Thus, it would seem reasonable to tolerate that preprocessing.

Figure B-5 depicts what you see.



Figure B-5. A screenshot from the game Doom

Figure B-6 shows what the neural network sees.

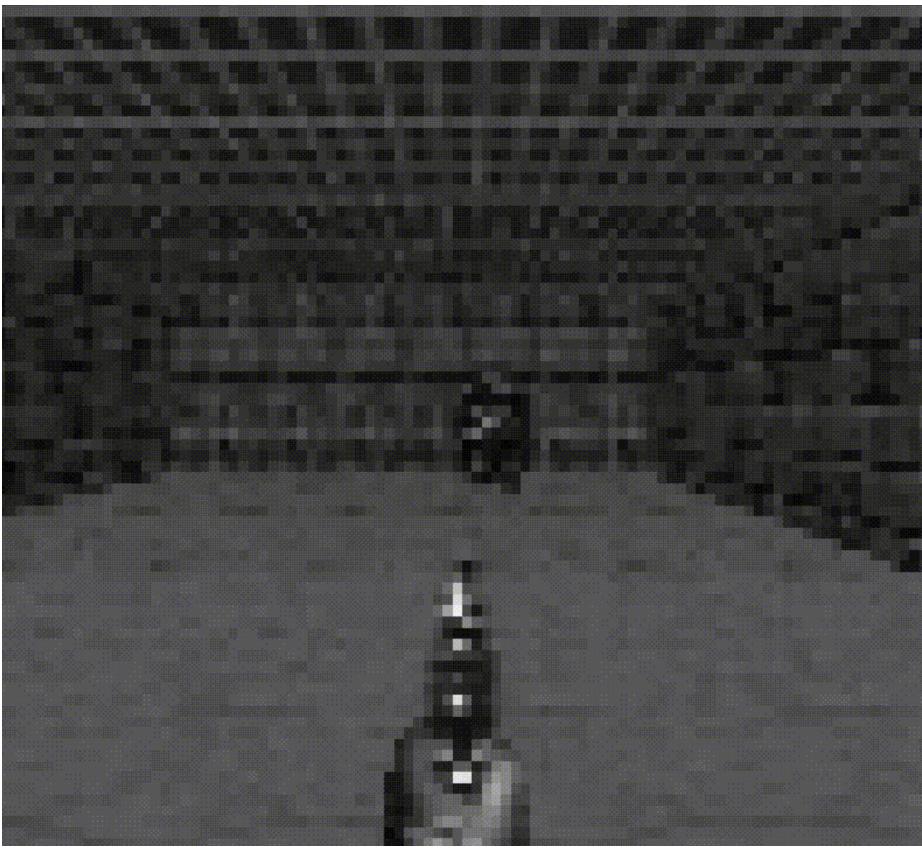


Figure B-6. Neural network input from the same screenshot

The image is resized into 84×84 . As the size of the input grows, so, too, does the need for memory and compute by the convolutional layers. The fine details of the image are not required to play the game correctly. Indeed, many are purely aesthetic. Resizing to a more reasonable size speeds up the training.



Skip Frame

In the original Atari paper, only one in four frames is actually processed. For the following three images, the last action is repeated. It speeds up roughly by four times the training without losing much information. Indeed, Atari games are not supposed to be played frame-perfect, and for most actions, it makes more sense to keep them for at least four frames.

History Processing

To give information to the neural network about the current momentum, the last four frames (with skip frame, you pick one of every four) are stacked into four channels, as demonstrated in [Figure B-7](#). Those four frames represent a history, as previously discussed in “Observation Setting” on page 419.

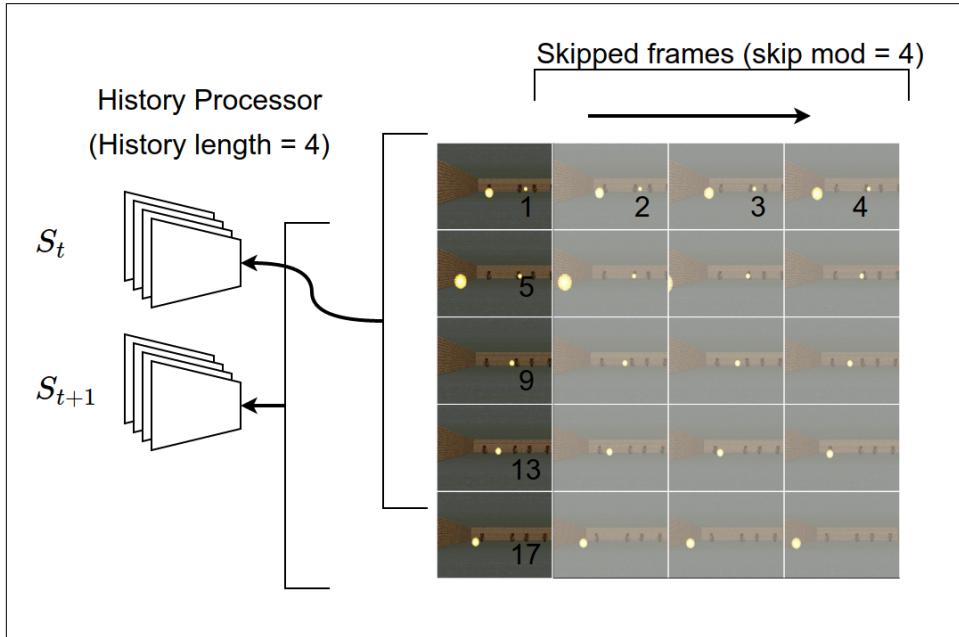


Figure B-7. History processing and stacking

To fill the first frames of the history, a random policy or a noop replay is used.¹

Double Q-Learning

The idea behind double-DQN is that the network is frozen every M updates (hard update) or smoothly averaged (`target = target * (smooth) + current * (1 - smooth)`) every update (soft update). Indeed, it adds stability to the learning by using a Q evaluation to use in the TD-error formula that is less prone to “jigging.” The Q update becomes the following:

$$Y_{\text{target}} = r_t + \gamma * \left(Q_{\text{target}}(s_t + 1, \arg \max_a Q(s_t + 1, a)) \right)$$

¹ You can use random starts for fair evaluation.

Clipping

You can clip the TD-error (bound between two limit values) such that no outlier update can have too much impact on the learning.

Scaling Rewards

Scaling the rewards such that Q -values are lower (in a range of $[-1; 1]$) is similar to normalization) can dramatically alter the efficiency of the learning. This is an important hyperparameter to not neglect.

Prioritized Replay

The idea behind [prioritized replay](#) is that not all transitions are born equal. Some are more important than others. One way to sort them is through their TD-error. Indeed, a high TD-error is correlated to a high level of information (in the sense of surprise). Those transitions should be sampled more often than the others.

Graph, Visualization, and Mean-Q

To visualize and debug the training or a method of reinforcement learning, it is useful to have a visual monitoring of the agent's progress. This is why I built the dashboard [webapp-rl4j](#) ([Figure B-8](#)).

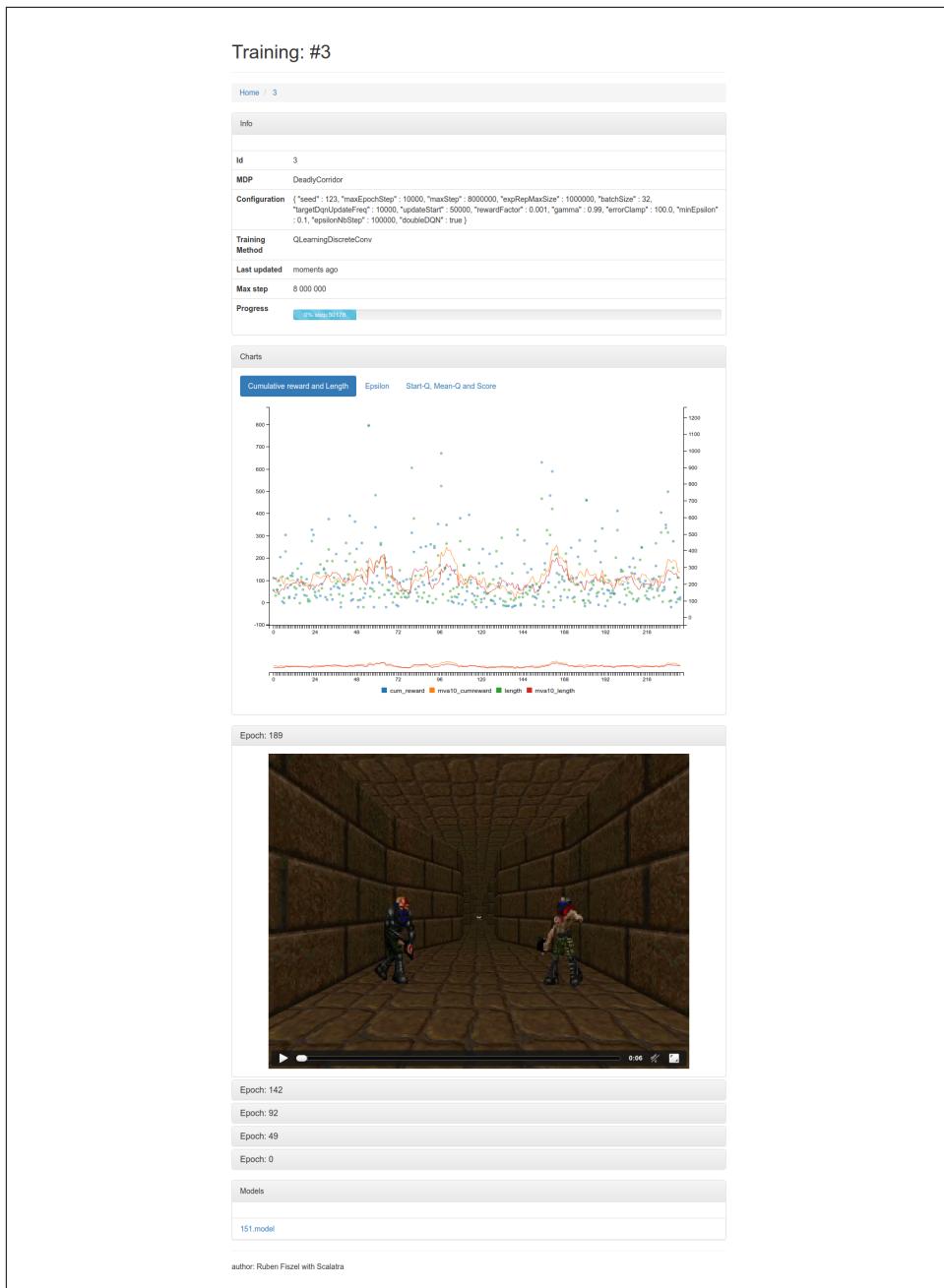


Figure B-8. Screen capture of webapp-rl4j

The most important thing is to keep track of cumulative rewards, as illustrated in [Figure B-9](#). This is a way to check that the agents effectively become better. It is important to notice that it represents the epsilon greedy strategy and not the directly derived policy from the Q approximation.

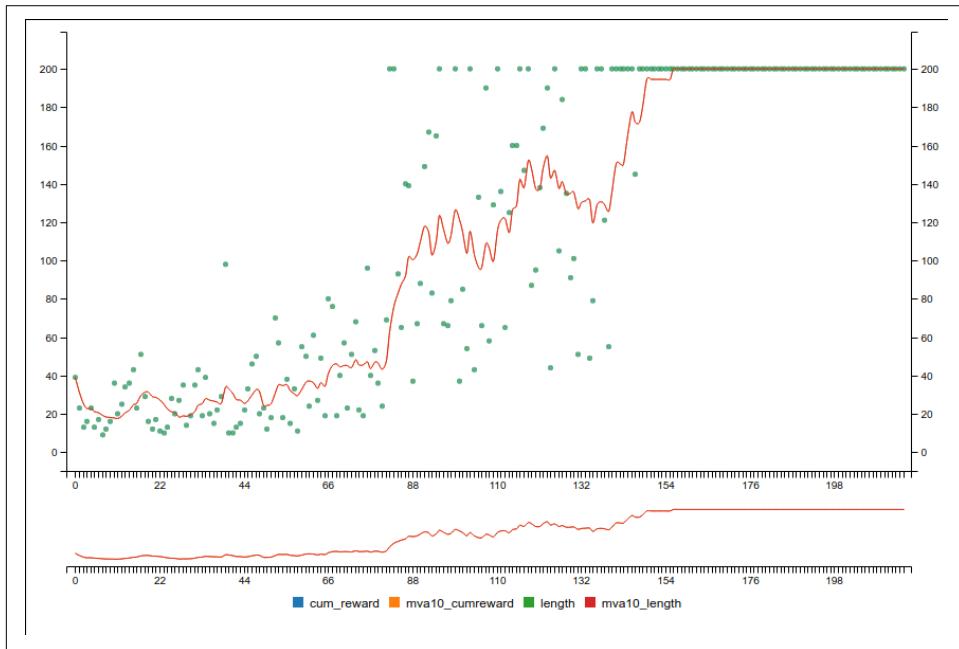


Figure B-9. Cumulative reward graph

You also might want to also track the loss (score of the neural network) and mean Q -values, as shown in [Figure B-10](#).

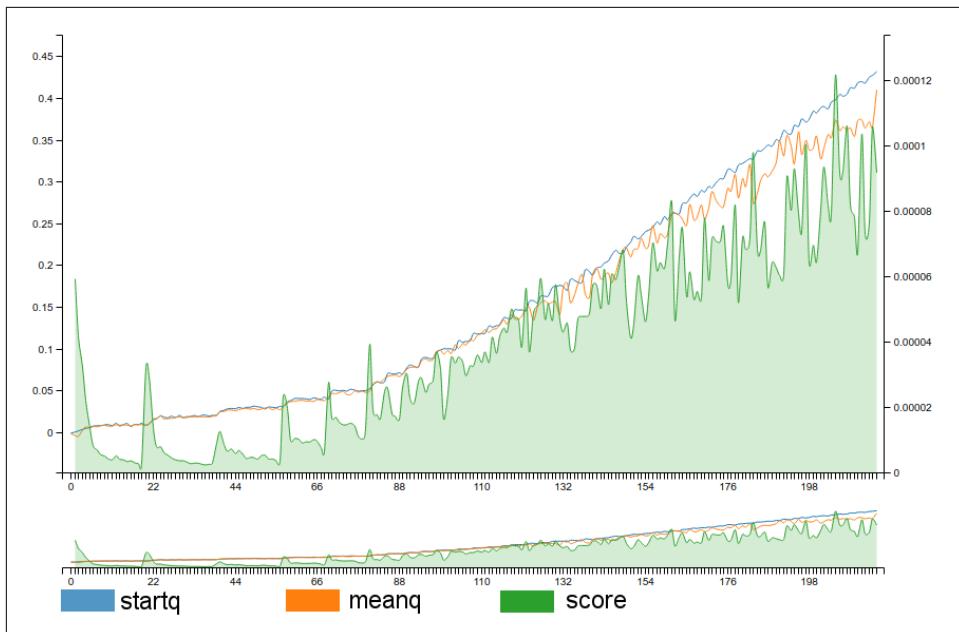


Figure B-10. Score and mean-Q graph

Unlike with classic supervised learning, the loss does not necessarily always decrease because the learning affects the labels!

If used with a target network, you should see some discontinuities from the noncontinuous evaluation of different target networks. Loss should decrease with respect to a single target network. The mean Q-values should smoothly converge toward a value proportional to the mean expected reward.

RL4J

RL4J is available on [GitHub](#). Currently, DQN with Experience Replay, Double Q-learning, and clipping is implemented.² Asynchronous Reinforcement Learning with A3C and Async N-step Q-Learning is included, too. It is possible to play both from pixels or low-dimensional problems (like Cartpole). Async Reinforcement Learning is experimental. Hopefully, contributions will enrich the library.

[Example B-3](#) presents a working example with RL4J to play Cartpole with a simple DQN. You can play Doom, too. Check out [rl4j-examples](#) for more examples. It also is

² For more details, check out the original blog post at <https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html>.

possible to provide your own constructed neural network model as an argument to any training method.

Example B-3. Basic RL4J in Scala

```
public static QLearning.QLConfiguration CARTPOLE_QL =
    new QLearning.QLConfiguration(
        123,           //Random seed
        200,           //Max step By epoch
        150000,         //Max step
        150000,         //Max size of experience replay
        32,            //size of batches
        500,           //target update (hard)
        10,             //num step noop warmup
        0.01,           //reward scaling
        0.99,           //gamma
        1.0,            //td-error clipping
        0.1f,           //min epsilon
        1000,           //num step for eps greedy anneal
        true            //double DQN
    );

public static DQNFFactoryStdDense.Configuration CARTPOLE_NET =
    new DQNFFactoryStdDense.Configuration(
        3,               //number of layers
        16,              //number of hidden nodes
        0.001,           //learning rate
        0.00             //l2 regularization
    );

public static void main( String[] args )
{
    //record the training data in rl4j-data in a new folder (save)
    DataManager manager = new DataManager(true);

    //define the mdp from gym (name, render)
    GymEnv<Box, Integer, DiscreteSpace> mdp = new GymEnv("CartPole-v0", false,
        false);

    //define the training
    QLearningDiscreteDense<Box> dql = new QLearningDiscreteDense(mdp,
        CARTPOLE_NET, CARTPOLE_QL, manager);

    //train
    dql.train();

    //get the final policy
    DQNPolicy<Box> pol = dql.getPolicy();

    //serialize and save (serialization showcase, but not required)
```

```
pol.save("/tmp/pol1");

//close the mdp (close http)
mdp.close();

}
```

Conclusion

This was an exciting journey through deep reinforcement learning. From equations to code, Q-learning is a powerful yet somewhat simple algorithm. The field of reinforcement learning is very active and promising. In fact, supervised learning could be considered a subset of reinforcement learning (by setting the labels as rewards). Maybe one day, reinforcement learning will be the panacea of Artificial Intelligence. Until then, we can expect to be awed by its diverse application in more and more mind-blowing problems. As a word of acknowledgement, I would like to thank Skymind and its amazing team for this very enriching internship.

APPENDIX C

Numbers Everyone Should Know

Table C-1 lists the numbers everyone should know, also known as “Jeff Dean’s 12 Numbers.”

Table C-1. Jeff Dean’s 12 numbers

Computer architecture operation	Duration (in nanoseconds)
L1 cache reference	0.5
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	100
Main memory reference	100
Compress 1 KB with Zippy	10,000
Send 2 KB over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Roundtrip within same datacenter	500,000
Disk seek	10,000,000
Read 1 MB sequentially from network	10,000,000
Read 1 MB sequentially from disk	30,000,000
Send packet CA->Netherlands->CA	150,000,000

Neural Networks and Backpropagation: A Mathematical Approach

Alex Black

Introduction

In this appendix, we'll look at the mathematics that underlie neural network training: the *backpropagation algorithm*. But before we dive in, let's take a step back, and ask the question: What are we trying to do when training a neural network?

At its core (issues such as overfitting aside), we want the training process to adjust the parameters of the neural network (based on the training data) such that the network will generate accurate predictions. The two components to this are *accurate predictions* and *adjust the parameters*.

Let's for a moment consider the task of classification, for which the neural network is asked to predict a class of the example, given some input data. There are many different ways to quantify how good predictions are for classification: accuracy, F1 score, negative log likelihood, and so on. All of these are valid measures, but some of them are much harder to optimize than others. For example, the accuracy of our network might not change if we change any given parameter in our network by a small amount; thus, directly optimizing accuracy using gradient-based methods is not possible (i.e., "accuracy" is not differentiable).

Conversely, other measures, such as negative log likelihood, will increase or decrease as we change the parameter values, even by a small amount. If we restrict ourselves to the class differentiable loss functions (such as negative log likelihood) to quantify the quality of our network's predictions and apply some calculus, we arrive at a simple and elegant algorithm for training neural networks.

The key ideas behind the backpropagation algorithm are quite simple:

- Quantify the goodness/badness of the network's current predictions based on a differentiable loss function
- Calculate gradients for each parameter in the network by applying the rules of multivariable calculus to the loss function and network structure
- Use the calculated gradients to iteratively adjust the network parameters in the direction that minimizes this loss function

To specify this in algorithmic form, the Stochastic Gradient Descent (SGD) algorithm is as follows:

```
Input: network parameters  $\mathbf{w}$ , loss function  $L$ , training data  $D$ , learning rate  $\alpha > 0$ 
while termination conditions are not met do
     $(\text{features}, \text{labels}) \leftarrow D.\text{getRandomMiniBatch}()$ 
     $\text{out} \leftarrow \text{getNetworkOutput}(\mathbf{w}, \text{features})$ 
     $\frac{\partial L}{\partial \mathbf{w}} \leftarrow \text{calculateParameterGradients}(\mathbf{w}, L, \text{out}, \text{labels})$ 
     $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial L}{\partial \mathbf{w}}$ 
end
```

The key part here is the calculation of the partial derivatives $\frac{\partial L}{\partial \mathbf{w}}$. If you're not familiar with partial derivatives, you can think of them as describing how the quantity of interest (the value of the loss function L) changes as a function of some other quantity (the value of each parameter w_i in the weight vector \mathbf{w}), if we hold all other values constant.



Step Size

In most cases, $\frac{\partial L}{\partial w_i}$ is also a function of w_i . This is why a) we can take only small steps (i.e., use small learning rate α), and b) have to recalculate the gradients after changing the parameter values at each iteration.

If $\frac{\partial L}{\partial w_i}$ is positive, increasing w_i by some small amount will increase the loss L for the current example; decreasing w_i will decrease the loss function.

At its heart, that's all there is to the backpropagation algorithm: define a loss function, calculate the derivatives of all parameters with respect to the loss function, and take a small step in the direction that would minimize the loss function. It is a simple but very powerful idea that underlies virtually all of deep learning.

Backpropagation in a Multilayer Perceptron

Enough text, let's get to the mathematics. Here, we'll be considering a simple multi-layer perceptron; that is, a standard fully connected feed-forward neural network layer, or “DenseLayer” in DL4J along with the sum of squared error loss function (L2 loss function in DL4J) for a single example:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

where N is the number of outputs, y_i is the i^{th} label, and $\hat{y}_i = \text{output}(\mathbf{w}, \mathbf{f})$ is the network's prediction of y_p given the feature vector \mathbf{f} and current parameters \mathbf{w} .

Figure D-1 shows a single-layer neural network. You will want to refer to this figure as you progress through the mathematics for the remainder of this appendix.

$$L(\mathbf{y}, \hat{\mathbf{y}}) = (y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + (y_3 - \hat{y}_3)^2$$

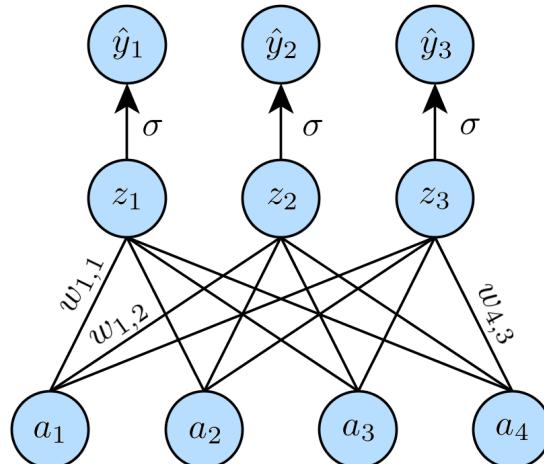


Figure D-1. Single-layer neural network

Let the input vector to the current layer be the vector \mathbf{a} (of length 4), the element-wise nonlinearity (activation function, such as tanh or Rectified Linear Unit [ReLU]) by σ , then the forward-pass equations for this network are:

$$z_i = b_i + \sum_{j=1}^4 w_{j,i} a_i$$

$$\hat{y}_i = \sigma(z_i)$$

where b_i is the bias and $w_{j,i}$ is the weight connecting input j to neuron i .

Given the loss function, the first partial derivative we want to calculate is with respect to the network output, \hat{y}_i :

$$\frac{\partial L}{\partial \hat{y}_j} = \frac{\partial}{\partial \hat{y}_j} \left(\sum_{i=1}^N (y_i - \hat{y}_i)^2 \right)$$

$$= \frac{\partial}{\partial \hat{y}_j} (y_j - \hat{y}_j)^2$$

$$= -2(y_j - \hat{y}_j)$$

Following the network structure backward, we next want to calculate the $\frac{\partial L}{\partial z_i}$ as a function of $\frac{\partial L}{\partial \hat{y}_i}$. This will depend on the mathematical form of the activation function $\sigma(z)$. For simplicity, we'll assume a simple activation function such as sigmoid, tanh, or ReLU (as the derivatives are simple element-wise functions; some functions such as softmax do not have this property, however).

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i}$$

$$= \sigma'(z_i) \frac{\partial L}{\partial \hat{y}_i}$$

For example, in the case of the sigmoid activation function, where $\sigma(z) = \frac{1}{1 + e^{-z}}$, we have $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Next, we apply the chain rule to calculate the partial derivatives of the weights $w_{j,i}$ given the previously calculated derivatives, $\frac{\partial L}{\partial z_i}$:

$$\frac{\partial L}{\partial w_{j,i}} = \sum_{k=1}^3 \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial w_{j,k}}$$

$$= \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial w_{j,i}}$$

$$= \frac{\partial L}{\partial z_i} \frac{\partial}{\partial w_{j,i}} \left(b_j + \sum_{k=1}^4 w_{k,i} a_k \right)$$

$$= a_i \frac{\partial L}{\partial z_i}$$

We can use the same approach to determine that the derivatives for the biases are given by $\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial z_i}$.

Finally, we want to calculate the derivatives of the loss function with respect to the input activations a_i , again given $\frac{\partial L}{\partial z_i}$:

$$\frac{\partial L}{\partial a_i} = \sum_{j=1}^3 \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial a_i}$$

$$= \sum_{j=1}^3 \frac{\partial L}{\partial z_j} \frac{\partial}{\partial a_i} \left(b_j + \sum_{k=1}^4 w_{k,j} a_k \right)$$

$$= \sum_{j=1}^3 \frac{\partial L}{\partial z_j} w_{i,j}$$

And that's it. If we had another layer below, we could apply the exact same approach, but with a_i taking the place of \hat{y}_i in the previous equations. We just follow the network connection structure backward, applying the chain rule as we go.

Extending this analysis to the mini-batch (i.e., multiple examples) case is straightforward: we can simply average the gradients for each parameter over the examples in each mini-batch. In practice, these equations are implemented using matrix multiplication and vector operations over mini-batches of data. Additionally, we can use this same approach to calculate the derivatives for other (more complex) models, such as Recurrent Neural Networks and Convolutional Neural Networks.

APPENDIX E

Using the ND4J API

ND4J is a scientific computing library for the Java Virtual Machine (JVM). It is designed to run fast in production environments. Here are its main features:

- Versatile n -dimensional array object
- Multiplatform functionality including graphics processing units (GPUs)
- Linear algebra and signal processing functions



ND4S is the Scala version of ND4J.

A usability gap has separated Java, Scala, and Clojure programmers from the most powerful tools in data analysis, like NumPy or Matlab. Libraries like Breeze don't support n -dimensional arrays, or tensors, which are necessary for deep learning and other tasks. ND4J and ND4S are used by national laboratories for tasks such as climatic modeling, which require computationally intensive simulations.

ND4J brings the intuitive scientific computing tools of the Python community to the JVM in an open source, distributed, and GPU-enabled library. In structure, it is similar to SLF4J. ND4J gives engineers in production environments an easy way to port their algorithms and interface with other libraries in the Java and Scala ecosystems.



The Complete ND4J User Guide Online

ND4J supports many more operations than are listed in this appendix. For the complete ND4J User Guide, go to <http://nd4j.org/user-guide>.



The Complete ND4J API Javadoc

For a complete resource on the ND4J API, check out Javadoc at <http://nd4j.org/doc/>.

Design and Basic Usage

ND4J is designed to run in many places and integrates with today's modern data systems. Following are some of its features:

- GPU support via CUDA
- Integration with Hadoop and Spark
- API is designed to mimic the semantics of Numpy

Most of the operations in ND4J are focused on manipulating arrays of numbers. The core data structure in ND4J is called an *NDArray*.

Understanding NDArrays

An NDArray is in essence an *n*-dimensional array. It is a rectangular array of numbers with some number of dimensions. We quantify an NDArray via the following properties:

- Rank
- Shape
- Length
- Stride
- Data type

In the list that follows, we explain what each of these properties do with respect to an NDArray.



NDArray and INDArray

We'll use the term NDArray to refer to the general concept of an *n*-dimensional array. The term INDArray refers specifically to the Java interface that ND4J defines. In practice, you can use these two terms interchangeably.

Rank

The rank of a NDArray is the number of dimensions. Two-dimensional NDArrays have a rank of 2; three-dimensional arrays have a rank of 3, and so on. You can create NDArrays with any arbitrary rank.

Shape

The shape of an NDArray defines the size of each of the dimensions. Suppose that we have a 2D array with 3 rows and 5 columns: this NDArray would have shape [3,5].

Length

The length of an NDArray defines the total number of elements in the array. The length is always equal to the product of the values that make up the shape.

Stride

The stride of an NDArray is defined as the separation (in the underlying data buffer) of contiguous elements in each dimension. Stride is defined per dimension, so a rank N NDArray has N stride values, one for each dimension. Note that most of the time, you don't need to know (or concern yourself with) the stride—just be aware that this is how ND4J operates internally. The next section has an example of strides.

Data type

The data type of an NDArray refers to the type of data it contains (e.g., float or double precision). Note that this is set globally in ND4J, so all NDArrays should have the same data type. Setting the data type is discussed later in this document.



Things to Know About NDArray-Indexing and Dimensions

Rows are dimension 0, and columns are dimension 1; thus, `INDArray.size(0)` is the number of rows, and `INDArray.size(1)` is the number of columns. Like normal arrays in most programming languages, indexing is zero-based. As such, rows have indexes 0 to `INDArray.size(0)-1`, and so on for the other dimensions.

NDArrays and Physical Memory Storage

NDArrays are stored in memory as a single flat array of numbers (or more generally, as a single contiguous block of memory) and hence differ a lot from typical Java multidimensional arrays such as a `float[][]` or `double[][][]`.

Physically, the data that backs an INDArray is stored off-heap; that is, it is stored outside of the JVM. This has numerous benefits, including performance, interoperability with high-performance BLAS libraries, and the ability to avoid some shortcomings of the JVM in high-performance computing (such as issues with Java arrays being limited to $2^{31}-1$ [2.14 billion] elements due to integer indexing).

ND4J General Syntax

There are three types of operations used in ND4J:

- Scalars
- Transforms
- Accumulations

We'll use the word "op" synonymously with operation. Most of the ops just take enums, or a list of discrete values that you can autocomplete. Scalars, transforms, and accumulations each have their own patterns. Let's take a look at them here.

Scalars

Scalars take just two arguments: the input and the scalar to be applied to that input. For example, `ScalarAdd()` takes two arguments: the input `INDArray x` and the scalar `Number num`; i.e., `ScalarAdd(INDArray x, Number num)`. The same format applies to every Scalar op.

Transforms

Transforms are the simplest because they take a single argument and perform an operation on it. Absolute value is a transform that takes the argument `x`, such as `abs(IComplexNDArray ndarray)`, and produces the result, which is the absolute value of `x`. Similarly, you would apply to the sigmoid transform `sigmoid()` to produce the "sigmoid of `x`."

Accumulations

Finally, we have accumulations, which are also known as reductions in the context of GPUs. Accumulations add arrays and vectors to one another and can *reduce* the dimensions of those arrays in the result by adding their elements in a row-wise op. For example, we might run an accumulation on the array

```
[1 2  
 3 4]
```

which would give us the vector:

```
[3  
 7]
```

This reduces the columns (i.e., dimensions) from two to one.

Accumulations can be either pair-wise or scalar. In a pair-wise reduction, we might be dealing with two arrays, `x` and `y`, which have the same shape. In that case, we could calculate the cosine similarity of `x` and `y` by taking their elements two by two:

```
cosineSim(x[i], y[i])
```

Or, take `EuclideanDistance(arr, arr2)`, a reduction between one array `arr` and another `arr2`.

The Basics of Working with NDArrays

The ND4J class

This class has a variety of helper static methods to assist in the creation of NDArrays. In the following subsections, we discuss a few of the more common methods you will encounter when working with NDArrays.

`Nd4j.zeros(int...)`. The shape of the arrays are specified as integers. For example, to create a zero-filled array with 3 rows and 5 columns, use `Nd4j.zeros(3,5)`.

`Nd4j.ones(int...)`. This method works similar to `.zeros(int...)` but fills the returned NDarray with 1s instead of 0s.

Initializing with other values. You can combine the ND4J class methods with other operations to create arrays with other values. For example, to create an array filled with 10s, do this:

```
INDArray tens = Nd4j.zeros(3,5).addi(10)
```

This initialization works in two steps: first by allocating a 3×5 array filled with zeros, and then by adding 10 to each value.

Initializing with random numbers. ND4J provides a few methods to generate INDArrays, in which the contents are pseudo-random numbers.

To generate uniform random numbers in the range 0 to 1, use:

```
Nd4j.rand(int nRows, int nCols)
```

(for 2d arrays), or:

```
Nd4j.rand(int[])
```

(for 3 or more dimensions).

Similarly, to generate Gaussian random numbers with mean zero and standard deviation 1, use:

```
Nd4j.randn(int nRows, int nCols)
```

or:

```
Nd4j.randn(int[])
```

For repeatability (i.e., to set ND4J's random number generator seed) you can use the following:

```
Nd4j.getRandom().setSeed(long)
```

Controlling the shape of NDArrays

In the next subsections, we discuss how to control the shape of NDArrays in the process of performing basic operations.

Creating basic arrays

In the following example, we create a one-dimensional array with two columns. The columns have the values { 1, 2 }.

```
INDArray nd = Nd4j.create(new float[]{1,2},new int[]{2}); //vector as row
```

Example: create a 2 x 2 NDArray. In the example that follows, we create a 2D NDArray that has two rows and two columns. The first row has the values {1, 2}, and the second row has the values {3, 4}.

```
INDArray arr1 = Nd4j.create(new float[]{1,2,3,4},new int[]{2,2});  
System.out.println(arr1);
```

Here's the output:

```
[[1.0 ,3.0]  
[2.0 ,4.0]  
]
```



Be Mindful of Row and Column Arrangements in ND4J

You can encode an NDArray in either C (row-major) or Fortran (column-major) order. For more details on row versus column major order, go to https://en.wikipedia.org/wiki/Row-major_order.

ND4J can use a combination of C and F order arrays together, at the same time. Most users can just use the default array ordering, but note that it's possible to use a specific ordering for a given array, should the need arise.

Example: add two 2 x 2 NDArrays together. Create a second array (`arr2`) and add it to the first (`arr1`):

```
INDArray arr2 = Nd4j.create(new float[]{5,6,7,8},new int[]{2,2});  
arr1.addi(arr2);  
System.out.println(arr1);
```

Here's the output:

```
[[7.0 ,11.0]  
[9.0 ,13.0]]
```

Creating NDArrays from Java arrays

ND4J provides convenient methods for the creation of arrays from Java float and double arrays.

To create a 1D NDArray from a 1D Java array, use the following:

- Row vector: `Nd4j.create(float[])` or `Nd4j.create(double[])`
- Column vector: `Nd4j.create(float[], new int[]{length,1})` or `Nd4j.create(double[], new int[]{length,1})`

For 2D arrays, use `Nd4j.create(float[][])` or `Nd4j.create(double[][])`.

For creating NDArrays from Java primitive arrays with three or more dimensions (`double[][][]`, etc.), one approach is to use the following:

```
double[] flat = ArrayUtil.flattenDoubleArray(myDoubleArray);
int[] shape = ...;           //Array shape here
INDArray myArr = Nd4j.create(flat,shape,'c');
```

Getting and setting individual NDArray values

For an INDArray, you can get or set values by using the indexes of the element you want to get or set. For a rank N array (i.e., an array with N dimensions), you need N indices.



Optimizing the Performance of NDArray Operations

Getting or setting values individually (e.g., one at a time in a for loop) is generally a bad idea in terms of performance. When possible, try to use other INDArray methods that operate on a large number of elements at a time.

To get values from a 2D array, you can use this:

```
INDArray.getDouble(int row, int column)
```

For arrays of any dimensionality, use this:

```
INDArray.getDouble(int...)
```

For example, to get the value at index i, j, k use the following:

```
INDArray.getDouble(i,j,k)
```

To set values, use one of the `putScalar` methods:

- `INDArray.putScalar(int[],double)`
- `INDArray.putScalar(int[],float)`
- `INDArray.putScalar(int[],int)`

Here, the `int[]` is the index, and the `double/float/int` is the value to be placed at that index.

Working with NDArray rows

There are multiple helper methods to work with sections of an NDArray.

Get a single row. To get a single row from an INDArray, you can use the following:

```
INDArray.getRow(int)
```

This obviously will return a row vector. Of note here is that this row is a view: changes to the returned row will affect the original array. This can be quite useful at times (e.g., `myArr.getRow(3).addi(1.0)` to add 1.0 to the third row of a larger array). To copy of a row, use this:

```
getRow(int).dup()
```

Get multiple rows. Similarly, to get multiple rows, use the following:

```
INDArray.getRows(int...)
```

This returns an array with the rows stacked; note, however, that this will be a copy (not a view) of the original rows, a view is not possible here due to the way NDArrays are stored in memory.

Setting a single row. For setting a single row, use this:

```
myArray.putRow(int rowIdx, INDArray row)
```

This will set the `rowIdx`th row of `myArray` to the values contained in the INDArray `row`.

Quick reference for determining the size/dimensions of NDArrays

The following methods are defined by the INDArray interface:

- Get the number of dimensions: `rank()`
- For 2D NDArrays only: `rows()`, `columns()`
- Size of the i th dimension: `size(i)`
- Get the size of all dimensions, as an `int[]`: `shape()`
- Determine the total number of elements in array: `arr.length()`
- See also: `isMatrix()`, `isVector()`, `isRowVector()`, and `isColumnVector()`

Dataset

The `org.nd4j.linalg.dataset.DataSet` class represents a **data transform with input and output data**.

In the context of neural networks a dataset represents a mapping of input features to an output vector (e.g., outcomes). The outcomes are specifically for neural network encoding such that any labels that are considered true are 1s. The rest are zeros.

Relationship to NDArray

NDArrays represent specific data vectors for input and output. `DataSet` objects represent the pairing of the input NDArray and the output NDArray. This is how we represent the function input and output for the learning process.

Common uses

In the code example that follows, we see how you can use the ND4J API to get the features and labels from a `DataSet` class instance. We also see how these structures are passed in to a trained model and produce the outcome NDArray.

```
DataSet t = ...  
INDArray features = t.getFeatureMatrix();  
INDArray labels = t.getLabels();  
INDArray predicted = model.output(features, false);
```

This simple snippet illustrates the core relationship of datasets, NDArrays, and DL4J models.

Creating Input Vectors

We want to highlight the topic of vector creation because it is a core technique used in every modeling job. You need to understand a vectorization strategy and then the practical application of the ND4J API to implement the vectorization strategy. As we mentioned previously, our input data for modeling is a set of input features associated with an output vector. In this section, we illustrate how to create both and tie them together with a `DataSet` object.



Simple Examples of Vectorization

These code snippets are meant to be simple to show off the basic concepts of setting features and label values. In practice, there are other ND4J methods that might be more efficient. Many times the DL4J record readers will perform most of this work for the practitioner manually. We're just showing how to manipulate the data in a practical manner in this section.

Basics of Vector Creation

Let's begin by creating a simple feature vector with two columns, as shown here:

```
INDArray myFeatures = ND4j.create(new float[]{0.5, 0.5}, new int[]{1,2});
```

Sizing the vector

We are controlling the size of the vector with the second parameter to the `ND4J.create()` method:

```
new int[]{1,2}
```

This parameter instructs ND4J to create an NDArray that is a single row with two feature columns.

Setting feature values

There are many ways to create feature values. In the following snippet, we show the simplest way to manually set the feature values in a vector:

```
for ( int row = 0; row < myFeatures.rows(); row++ ) {  
    for ( int col = 0; col < myFeatures.getRow( row ).columns(); col++ ) {  
        myFeatures.getRow(row).putScalar(col, 0.9);  
    }  
}
```

In this example we set every column in every row to the value 0.9.

Setting the label

Setting a label is as simple as setting the feature values because both the input feature data and the output data use the same data structure: the NDArray. To set up our label data, we need to encode the data based on what type of model we're building and what kind of output strategy we're after.

Single-label output. In this situation, our output NDArray for the label would have a single column and we'd set the value to 0.0 for one label and 1.0 for the other label:

```
myFeatures.getRow(0).putScalar(0, 1.0);
```

Multiple-label output. In this situation, our output NDArray for the label would have a column per label. Each class (label) in our training dataset will match up to a specific column in the output NDArray. For training purposes, we'd set the value of the column that has the same index as the label to be 1.0 as we see in the following example:

```
myFeatures.getRow(0).putScalar(0, 0.0);  
myFeatures.getRow(0).putScalar(1, 1.0);  
myFeatures.getRow(0).putScalar(2, 0.0);
```

Regression output. In this situation, our output NDArray for the output value of the regression model would have a single column. We set this value to the floating-point value we associate with the input vector.

```
myFeatures.getRow(0).putScalar(0, 55.25);
```

Using MLLibUtil

The DL4J platform contains tools to support interoperability with other machine learning libraries, as well. In this section, we illustrate a few methods for the `MLLibUtil` class to help work with Spark vector objects.

Converting from INDArray to MLLib Vector

In the event that you need to convert an `INDArray` to a MLLib vector, you can use the `MLLibUtil.toVector(INDArray)` method, as seen here:

```
Vector prediction = MLLibUtil.toVector( myNDArray );
```

Converting from MLLib Vector to INDArray

For the situation in which you need to convert a MLLib vector to an `INDArray` vector, you can use the `MLLibUtil.toVector(Vector)` method, as demonstrated in the following:

```
INDArray ndArray = MLLibUtil.toVector( labeledPoint.features() );
```

Making Model Predictions with DL4J

In this section, we look at the interplay of classes in both DL4J and ND4J for making model predictions at the API level.

Using the DL4J and ND4J Together

Typically, in the context of making a prediction from a model in DL4J, we use a `MultiLayerNetwork` method to represent the model and pass in an `INDArray` object as input to the `output()` method. The output of this method is a list of probabilities representing the probability per label, where we'd consider the largest probability as the “predicted label” for the input:

```
MultiLayerNetwork.output(INDArray input, boolean train)
```

We'll see more examples later of ways to use this method with a trained DL4J model to produce output represented by ND4J `NDArrays`.



Where Do the Output Values Come From?

These values are the activations of the final layer (“output layer”) of the network after a feed-forward pass based on the input vector.

The output values contained in the NDArray returned by the `output()` method typically take this form:

```
[0.5, 0.5]
```

In this example, we see a single row of an NDArray that has two columns. Each of these columns represents the score for the associated label. The meaning of the values in the columns of each row depends on which type of output layer generated the output. In the subsections that follow, we show how to deal with the different types of output generated by different output layers.

Differences between output vector depending on output layer type

Remember that the softmax activation function bounds the sum of the outputs to be 1.0 (e.g., probabilities) versus the sigmoid activation function, which bounds the output values individually. For example, the probabilities of the output of a softmax layer will all add up to 1.0, yet the output of a sigmoid layer might have every unit with 0.9 as the output. Sigmoid layers do not have this “lateral” constraint.

Logistic output layer for binary classification. The output for this type of layer represents the probability of a binary label being true.

```
INDArray predictions = trainedNetwork.output( ndArrayFeatures );
```

Each entry in the output vector represents a specific type of label and is independent from the other labels in terms of its probability value:

Softmax output layer for multilabel classification. The output for the softmax layer will be a list of probabilities all summing to 1.0:

```
INDArray predictions = trainedNetwork.output( ndArrayFeatures );
for ( int row = 0; row < output.rows(); row++ ) {
    System.out.println( "Input Row: " + row );
    for ( int col = 0; col < output.getRow( row ).columns(); col++ ) {
        System.out.println( "\tColumn: " + col + ":" + output.getRow( row )
            .getDouble( col ) );
    }
}
```

This gives us output similar to the following:

```
Input Row: 0
    Column: 0:0.996791422367096
    Column: 1:0.0032307980582118034
Input Row: 1
    Column: 0:0.0016306628240272403
    Column: 1:0.9983481764793396
Input Row: 2
    Column: 0:0.0016311598010361195
    Column: 1:0.9983481168746948
```

```
Input Row: 3
Column: 0:0.9988488554954529
Column: 1:0.0011729325633496046
```

Each entry in the output vector represents the probability of the associated label.
Each column in a row is associated with a specific label.

Linear output layer for regression output. A regression model uses a single output that represents the value we're looking to model as output from an "identity" activation function (e.g., linear).

Getting the predicted label from the returned INDArray

We take the result probabilities from the `output()` method call on the trained network and find the maximum probability, as demonstrated here:

```
INDArray predictions = trainedNetwork.output( ndArray );
int maxLabelIndex = Nd4j.getBlasWrapper().iamax( predictions );
```



Advantages of Using the `.output()` Method Over `.predict()`

You can use the `output()` method with regression and classification, so it's a bit more flexible in how you can use it.

APPENDIX F

Using DataVec

Alex Black

DataVec is a library for handling machine learning data. DataVec handles the Extract, Transform, and Load (ETL) or *vectorization* component of a machine learning pipeline. The goal of DataVec is to simplify the preparation and loading of raw data into a format ready for use for machine learning. DataVec includes functionality for loading tabular (comma-separated values [CSV] files, etc.), image, and time-series datasets, both for single machine and distributed (Apache Spark) applications.



ND4J Vector Creation and DataVec

DataVec is meant to handle many of the feature and label creation chores mentioned previously in this book. Using DataVec is considered a best practice for DL4J workflows on a single machine and on Spark.

DataVec provides two main categories of functionality:

- Functionality for loading data, from a variety of formats
- Functionality for performing common data transformation operations (often called *data wrangling* or *data munging*)

These two categories of functionality are discussed separately in the sections that follow.

Loading Data for Machine Learning

Machine learning data comes in a wide variety of formats, with different requirements and libraries for loading each. Too often, machine learning practitioners end

up writing one-off code to load their data; this can be both time consuming and error prone. DataVec attempts to alleviate these issues in two ways: first, by providing data loading functionality for common use cases (e.g., reading images and CSV data), and, second, by providing a simple set of abstractions for adding in new data formats or data sources.



ETL, Preprocessing, and Vectorization

It is a recommended best practice to use a tool like DataVec to preprocess the data into a format that DL4J can easily read and automatically vectorize. Hand-vectorizing raw data is a labor-intensive activity and you should avoid it whenever possible.

The set of abstractions that DataVec provides is relatively simple.

Writable is an interface that represents a piece of data. For example, you can use a DoubleWritable to represent double-precision numerical values, and you can use Text instances for text data.¹

The RecordReader interface provides a mechanism for going from raw data formats to a common example format. Specifically, a RecordReader takes raw data and converts it to a List<Writable> representation, which then can be read by DL4J's RecordReaderDataSetIterator class, which also handles combining examples into mini-batch DataSet objects. Figure F-1 shows this process.

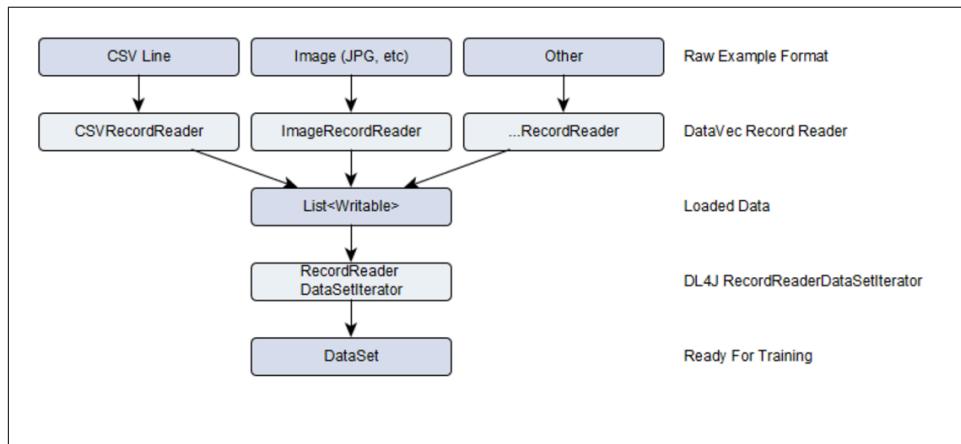


Figure F-1. DataVec processing pipelines

¹ Additional types include IntWritable, LongWritable, FloatWritable, and NullWritable. DataVec also provides an NDArrayWritable class for efficient handling of numerical array data using ND4J, such as images.

Similarly, the `SequenceRecordReader` interface provides a mechanism for loading sequence (time-series) data. Whereas a single (nonsequence) example is represented as a `List<Writable>` in `DataVec`, a sequence example is represented as a `List<List<Writable>>`: we can think of this as a list of (or sequence of) time-steps. That is, the outer list is a list of time-steps, and the inner list is a list of values at each time-step. Put another way, the code `mySequence.get(i).get(j)` would return the j th value at the i th time-step. Using a sequence record reader is virtually the same as a record reader, as illustrated in [Figure F-2](#).

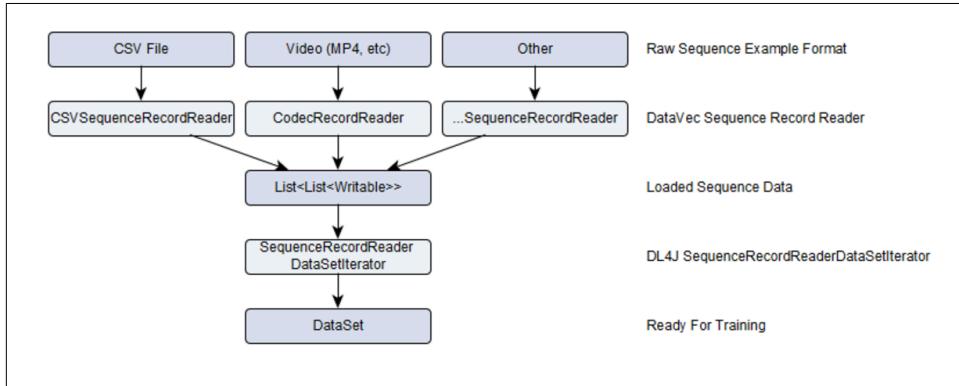


Figure F-2. DataVec sequence pipeline

One important aspect of the way `DataVec` and `DL4J` handle training data is that data is loaded only when required, through the use of iterator patterns. This means that we can load our training data only when we need it (prefetching it asynchronously, if necessary) instead of loading all of our data into memory at once (which usually isn't feasible with large datasets). The `RecordReader`, `SequenceRecordReader`, and `DataSetIterator` interfaces all have `next()`, `hasNext()`, and `reset()` methods for this purpose.

Loading CSV Data for Multilayer Perceptrons

You can load CSV data by using just a few lines of code. The process here also can be used for other delimited data such as tab-delimited formats.

[Example F-1](#) shows how to load a CSV data file and create a `DataSetIterator` ready for training a network in `DL4J`.

Example F-1. Loading CSV data

```
//First, specify the file location, and some properties:
File file = new File( "/path/to/my/file.csv" );
```

```

int numLinesToSkip = 0;           //Optional, allows us to skip header lines

String delimiter = ",";          //Comma-delimited

//Create the record reader, and initialize it:

RecordReader reader = new CSVRecordReader( numLinesToSkip, delimiter );
InputSplit inputSplit = new FileInputSplit( file );
reader.initialize( inputSplit );

//Create the DataSetIterator. We are assuming classification here:

int minibatchSize = 10;          //Number of examples in each minibatch

int labelIndex = 7;              //Index of column that contains the label

int numClasses = 5;              //Number of classes (label categories)

DataSetIterator iterator =
    new RecordReaderDataSetIterator( reader, minibatchSize, labelIndex, numClasses );

//Train the network with the DataSetIterator:

myNetwork.fit( iterator );

```

Here are some notable points:

- The no-arg constructor (`new CSVRecordReader()`) defaults to comma-delimited format and 0 lines skipped.
- In `RecordReaderDataSetIterator`, the mini-batch size is freely configurable by the user.
- A number of other constructors exists on `RecordReaderDataSetIterator` for cases such as multilabel regression, and situations without labels, such as unsupervised learning.
- For classification, `RecordReaderDataSetIterator` assumes that one of the columns contains integer values for the class index. That is, one column contains integer values in the range 0 to `numClasses` – 1 inclusive.
- You can use alternative delimiters. For example, `\t` for tab-delimited, or use the static field `CSVRecordReader.QUOTE_HANDLING_DELIMITER` to handle records that contain quotes with commas in them.

- CSVRecordReader outputs examples in the order in which they appear in the file. If the data is in a nonrandom order (such as all class 0 examples followed by all class 1 examples), it should first be shuffled as an extra step.

Loading Image Data for Convolutional Neural Networks

DataVec's `ImageRecordReader` is the record reader to use with image data and to perform common image operations such as cropping. `ImageRecordReader` supports a wide variety of image formats (including JPG, PNG, TIFF, and BMP, among others) with efficient loading and image operations provided by JavaCV and OpenCV.

The `ImageRecordReader` implementation contains a great degree of flexibility as to how data files are loaded. [Example F-2](#) presents a simple demonstration for loading images. In this example, assume that our image files are present in directories, where the name of the directory is the label (image class/category). For example, files are in directories `../root_directory/label_0`, `../root_directory/label_1`. Note that label directory names themselves can be anything.

Example F-2. Loading image data

```
//First, specify input image depth (number of channels)

int inputNumChannels = 3;      //3 for color/RGB, 1 for grayscale
int outputHeight = 32;         //Scale to 32 pixels height output
int outputWidth = 32;          //Scale to 32 pixels width output

//Second, specify the root directory, and allowable formats

// We'll use a Random instance to randomize the order

File rootDir = new File( "/path/to/my/root_directory" );
String[] allowedExtensions = BaseImageLoader.ALLOWED_FORMATS;
Random rng = new Random();

FileSplit inputSplit = new FileSplit( rootDir, allowedExtensions, rng );

//Each image has to be associated with a label. We'll use the path of each
// file to do this, using ParentPathLabelGenerator

ParentPathLabelGenerator labelMaker = new ParentPathLabelGenerator();
```

```

//Create and initialize our ImageRecordReader

ImageRecordReader reader =
    new ImageRecordReader( outputHeight, outputWidth, inputNumChannels, labelMaker );

reader.initialize( inputSplit );

//Finally, create the DataSetIterator:

int minibatchSize = 10;           //Number of examples in each minibatch

int labelIndex = 1;              //Always value 1 for ImageRecordReader

int numClasses = 3;              //Number of classes (label categories)

DataSetIterator iterator =
    new RecordReaderDataSetIterator( reader, minibatchSize, labelIndex, numClasses );

//Train the network with the DataSetIterator:

myNetwork.fit( iterator );

```

`ImageRecordReader` supports a large number of other features.

For example, to implement a train/test split on our images, we can add the code in [Example F-3](#) to that in [Example F-2](#).

Example F-3. Creating test/train splits

```

InputSplit[] trainTest = inputSplit.sample( null, 80, 20 );

InputSplit trainData = trainTest[0];           //80% train

InputSplit testData = trainTest[1];           //20% test

(other code omitted)

reader.initialize( trainData );

```

Similarly, to add some additional transform steps to the process, we can pass in an `ImageTransform` class to the `ImageRecordReader` initialization, as illustrated in [Example F-4](#); for example, to randomly flip and randomly crop images.

Example F-4. Transform steps

```
int maxCropPixels = 20;

ImageTransform transform =
    new MultiImageTransform( new Random(),
    new FlipImageTransform(), new CropImageTransform( maxCropPixels ) );

reader.initialize( trainData, transform );
```

Loading Sequence Data for Recurrent Neural Networks

There are many possible representations and formats for sequence (time-series) data. In this section, we focus on a simple but useful format:

- CSV data, with one time-series per file (time-series lengths can differ between files).
- In the CSV files, each row represents one time-step.
- The features and labels are present for all time-steps and are included in the same file for each example (i.e., some columns are features, one column is the label index for classification).

As per the (nonsequence) CSV example earlier, in [Example F-5](#) we will assume a classification problem, where the class label is an integer value between 0 and numClasses – 1 (inclusive).

Example F-5. Loading sequence data with DataVec

```
//First, specify the base directory that contains the CSV files

File baseDir = new File("/path/to/base_directory");

//Second, create and initialize the sequence record reader

// We are using a random number generator to randomize the order

InputSplit inputSplit = new FileSplit(baseDir, new Random());

int numLinesToSkip = 0;           //Optional, allows us to skip header lines

String delimiter = ",";          //Comma-delimited files

SequenceRecordReader reader = new CSVSequenceRecordReader(numLinesToSkip, delimiter);

reader.initialize(inputSplit);

//Create the DataSetIterator for training:
```

```

int minibatchSize = 10;           //Number of examples in each minibatch

int labelIndex = 7;              //Index of column that contains the label

int numClasses = 5;              //Number of classes (label categories)

boolean regression = false;

DataSetIterator iterator =
    new SequenceRecordReaderDataSetIterator( reader, minibatchSize, numClasses,
                                              labelIndex, regression );

DataSetIterator iterator =
    new SequenceRecordReaderDataSetIterator( reader, minibatchSize, labelIndex,
                                              numClasses );

//Train the network with the DataSetIterator

myNetwork.fit(iterator);

```

`SequenceRecordReader` also supports other use cases, such as regression and loading features and labels from separate `SequenceRecordReaders` (e.g., separate files).

Transforming Data: Data Wrangling with DataVec

Frequently, machine learning data is in a format that requires some level of preprocessing before we can use it. This preprocessing can be as simple as removing some unnecessary columns from the dataset, or as complex as joining and cleaning data from multiple independent data sources. DataVec also provides functionality for this stage of the pipeline, with extensive transformation functionality for both standard and sequence data.

The typical workflow when using DataVec for data wrangling is as follows:

1. Define a Schema for the raw input data (more on this later).
2. Define a set of operations to perform on the data (removing columns, handling missing values, etc.).
3. Load the data.
4. Execute the operations.
5. Save the processed data.

When you're handling data that requires preprocessing, we recommend separating out the data preprocessing and network training into separate steps. That is, we recommend executing the preprocessing, storing the data to disk, and then loading it from disk when required for network training.

For executing a set of operations on a dataset, Apache Spark is used; this enables execution on both a cluster and a single machine (via Spark in local mode). Furthermore, by executing operations on Spark, it is possible to scale to large datasets (hundreds of millions of records, or more). Though Spark execution is currently the only option, keep in mind that the DataVec API itself is independent of the execution platform; other cluster computing frameworks can be added in the future.

DataVec Transforms: Key Concepts

For transformation functionality, DataVec has a few key design ideas and classes of which you should be aware.

First, examples in DataVec are represented in the same manner as in the previous section: each example is a `List<Writable>` for standard data, or a `List<List<Writable>>` for sequences. After each operation, the new data is represented in one of these formats. This also means that you can use the `RecordReader` and `SequenceRecordReader` classes to load the raw data that you want to transform.

A Schema in DataVec is a class that defines three things for our data:

- The name of each column
- The type of each column (numerical, categorical, string, etc.)
- The restrictions (if any) on allowable values for each column (e.g., whether a column must contain only positive values)

For sequence data, we instead have a `SequenceSchema`, which contains the same per-column information as a normal Schema. DataVec tracks the Schema as it changes after each operation. We can obtain the Schema after the transformation operations have been executed or at any point during the process.

A `TransformProcess` defines a sequence of operations to perform on our data. A `TransformProcess` is constructed by specifying two things:

- The Schema of the initial input data
- The set of operations we wish to execute

These are specified by use of a builder pattern (as in the examples that follow), or optionally using a JSON or YAML file.

DataVec provides a number of types of operations that you can execute on data. [Table F-1](#) summarizes these operations.

Table F-1. DataVec operation types

Name	Description	Example applications
Transform	A general operation per example or per sequence	Removing columns, mathematical operations, parsing data/time values
Filter	Remove examples that match a condition	Filtering examples with missing or invalid values
Reduce	Group examples by a key and reduce	Minimum, maximum, or sum values for each customer ID
Convert to Sequence	Group individual examples into a sequence, by one or more key columns	Log data: group separate records into a sequence for each IP or customer
Convert from Sequence	Split each time-step in sequence data into separate (nonsequence) format	Split sequences of custom transactions into separate records

DataVec Transform Functionality: An Example

This section shows a very simple example of how to perform some common operations on a small dataset.

For this example, assume that we have some transaction data from which we want to predict a label (fraud/legitimate). Let's further assume that we have the following simple dataset, with the following columns: [customerID, dateTime, amount, label]:

```
3420348,2016-01-01 06:55:07,150.00,legitimate
9087434,2016-01-01 15:16:18,78.10,legitimate
4530843,2016-01-02 11:39:24,780.83,fraud
```

Our first step is to specify a Schema for the data. We can do this as follows:

```
Schema schema = new Schema.Builder()
    .addColumnLong("customerID")
    .addColumnString("dateTime")
    .addColumnDouble("amount")
    .addColumnCategorical("label", Arrays.asList("legitimate", "fraud"))
    .build();
```

Note that the columns are specified in the order in which they appear in the file.

However, this dataset cannot be used as is. To train a neural network, we require numerical inputs for our data. To prepare the data for training, we will do the following:

- Remove the customer ID column
- Convert the categorical (string) label to an integer (0 or 1)
- Parse the dateTime column, and extract out the hour of the day as a new feature column

We can specify all of these operations by using `TransformProcess`, as follows:

```
TransformProcess process = new TransformProcess.Builder(schema)
    .removeColumns("customerID")
    .categoricalToInteger("label")
    .stringToTimeTransform("dateTime", "YYYY-MM-dd HH:mm:ss", DateTimeZone.UTC)
    .transform(new DeriveColumnsFromTimeTransform.Builder("dateTime")
        .addIntegerDerivedColumn("hourOfDay", DateTimeFieldType.hourOfDay()).build())
    .removeColumns("dateTime")
    .build();
```

Each operation will be executed in the order in which it is specified. Most of these operations are straightforward to understand from the names. For those with more detailed arguments (such as the `stringToTimeTransform` operation) refer to the DataVec Javadoc.

Finally, we must load our data, execute the operations just defined, and then save our data. We can do this as follows:

```
//Some initial setup for Apache Spark:
SparkConf conf = new SparkConf();
conf.setMaster("local[*]");
conf.setAppName("DataVec Example");
JavaSparkContext sc = new JavaSparkContext(conf);

//Load our data using Spark
String path = "/path/to/my/file.csv";
JavaRDD<String> lines = sc.textFile(path);
JavaRDD<List<Writable>> examples =
    lines.map(new StringToWritablesFunction(new CSVRecordReader()));


//Execute the operations
SparkTransformExecutor executor = new SparkTransformExecutor();
JavaRDD<List<Writable>> processed = executor.execute(examples, process);

//Save the processed data:
JavaRDD<String> toSave = processed.map(new WritablesToStringFunction(","));
toSave.saveAsTextFile("/path/to/save/to/");
```

After executing these operations, our data is as follows:

```
6,150.00,0
15,78.10,0
11,780.83,1
```

The column of the output data above are “hourOfDay,” “amount,” and “label.” We can obtain the column names and types from the Schema for the output data using `process.getFinalSchema()`.

That's it. In just a few lines, we have loaded our data, executed a number of preprocessing operations (in a way that is scalable to very large datasets), and exported our data ready for training. Though the example is simple, hopefully it demonstrates the utility and flexibility of DataVec for preparing machine learning data.

Working with DL4J from Source

Some developers might choose to build custom extensions, modify the core of DL4J, or work from the latest codebase. For those practitioners, we provide the instructions here on how to get DL4J set up directly from source.

GitHub is a web-based revision control system, and the de facto host for most open source projects. If you're planning to contribute to the ND4J or DL4J projects by fixing bugs and committing code, you will need Git and GitHub.



Do You Really Need to Work from Source?

If you plan to simply use the code base, you do not need to install GitHub, and thus you should avoid downloading the source code.

Verifying Git Is Installed

To verify that Git is installed and working, type the following into the command line:

```
git --version
```

If the command returns an error, we suggest installing Git. If you do not already have a GitHub account we suggest signing up for one; it's free and simple to do.

Cloning Key DL4J GitHub Projects

To work with source, clone ND4J, or DL4J, type the following commands into the console:

```
git clone https://github.com/deeplearning4j/nd4j  
git clone https://github.com/deeplearning4j/datavec  
git clone https://github.com/deeplearning4j/deeplearning4j
```

You might also want to clone the DL4J examples so that they can work with ND4J or DL4J's prebuilt samples (the version will vary):

```
git clone https://github.com/deeplearning4j/dl4j-0.4-examples
```



More Examples Help

For a walkthrough of installing our examples with Git, IntelliJ, and Maven, see our [Quickstart page](#).

Downloading Source via Zip File

Another way to get the source code is by clicking the Download ZIP button from the [ND4J GitHub page](#).

Using Maven to Build Source Code

You can use Maven in conjunction with Git to ensure that ND4J, DataVec, and DL4J build correctly. To ensure that you have the most recent, working version of these libraries locally, change into their root directories and type the following command into your prompt:

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

Running a clean install on ND4J, DataVec, and DL4J, in that order, is a good way to get the most recent bug fixes and features.

APPENDIX H

Setting Up DL4J Projects

DL4J is a suite of tools that together provide a full platform for deep learning. There are multiple dependencies that you can wire together to perform different functions in support of deep learning models. DL4J uses Maven to control how dependencies are wired together in a project. In this section, we show you a few of the relevant dependencies that you can use to build your own deep learning models, tools, and integrations.

Creating a New DL4J Project

DL4J is an open source project targeting professional Java developers familiar with production deployments, an IDE such as IntelliJ, and an automated build tool such as Maven. Our tool will serve you best if you have those tools under your belt already. ND4J and DataVec, our vectorization library, will be automatically installed by following the quickstart instructions below.

Here are the system configuration requirements:

1. Java 7 or above
2. Maven 3.2.5 or above (dependency management and automated build tool)
3. Git

There are also some optional steps that you might want to perform including installing the following:

- Cuda 7 for GPUs
- Scala 2.10.x
- Windows
- GitHub

Let's begin by setting up your environment, starting with Java.

Java

Java is the main interface and networking language of ND4J because it's used for everything from distributed cloud-based systems with thousands of nodes, to low-memory Internet of Things (IoT) devices. It's a "write once, run anywhere" language.

To test which version of Java you have (or whether you have it at all), type the following into your command line:

```
java -version
```

If you don't have Java 7 installed on your machine, download the [Java Development Kit \(JDK\) here](#). For newer Macs, you'll want the file on the first line to mention Mac OS X (the number after jdk-7u increments with each update). It will look something like this:

Mac OS X x64 185.94 MB - jdk-7u79-macosx-x75.dmg

Working with Maven

Maven is an automated build tool for Java projects (among its other uses). It locates the latest version of ND4J and DL4J project libraries (*.jar* files) and downloads them automatically. You can find those repositories on [Maven Central](#). Maven lets you to install both ND4J and DL4J projects easily. It works well with Integrated Development Environments (IDEs) such as IntelliJ.

To check if Maven is installed on your machine and which version you have, type the following into the command line:

```
mvn --version
```

If you do not have the most recent version of Maven, you'll need to update it (as of this writing, it is 3.3.x). You can find instructions to install Maven at <https://maven.apache.org/>.

Download the compressed file containing Maven's latest stable version. Lower on the same page, follow the instructions specific to your operating system; for example, "*Unix-based Operating Systems (Linux, Solaris, and Mac OS X)*".

A minimal Project Object Model file

To run DL4J in your own projects, we highly recommend using Apache Maven for Java users, or a tool such as SBT for Scala. The basic set of dependencies and their versions are shown here:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                               http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>MyGroupId</groupId>
    <artifactId>MyArtifactId</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <nd4j.version>0.5.0</nd4j.version>
        <dl4j.version>0.5.0</dl4j.version>
        <datavec.version>0.5.0</datavec.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.deeplearning4j</groupId>
            <artifactId>deeplearning4j-core</artifactId>
            <version>${dl4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-x86</artifactId>
            <version>${nd4j.version}</version>
        </dependency>
    </dependencies>
</project>

```

Project Object Model explanation. Project Object Model (POM) files can be complicated, so we'll note a few key configurations to help you better understand what's going on. The following describes a couple of core DL4J POM configuration entries and what they install in our local maven repo:

`deeplearning4j-core`

Contains the core DL4J neural network implementations

`nd4j-x86`

The CPU version of the ND4J library that powers DL4J

IDEs

An IDE will allow you to work with our API and build your networks with a few clicks. We suggest using IntelliJ or Eclipse, either of which will work with your installed version of Java and communicate with Maven to handle the dependencies.

Quickstart a DL4J project by using IntelliJ

The free community edition of IntelliJ has installation instructions.

After installation, perform the following steps to get up and running (Windows users, see the Walkthrough section at <http://deeplearning4j.org/quickstart.html#walk>):

1. In the command line, type the following:

```
git clone https://github.com/deeplearning4j/dl4j-0.4-examples.git
```

We are currently on examples version 0.0.4.x.)

2. In IntelliJ, in the menu tree, click File → New → Project from Existing Sources, and then create a new project using Maven.
3. Point to the root directory of the preceding examples. This opens them in your IDE.
4. Copy and paste the *pom.xml* file described in the next section that powers the Maven project.
5. Wait for IntelliJ to download all the dependencies. (You'll see the horizontal bar working in the lower right.)
6. Select an example from the lefthand file tree, and then click “run.”

Setting Up Other Maven POMs

In this section, we look at how to quickly set up Maven *pom.xml* files for ND4J. We also look at other components in the suite of tools.

ND4J and Maven

ND4J backends are what power the linear algebra operations behind DL4J’s neural networks. ND4J backends vary by chip where CPUs work fastest with x86 and GPUs work best with Jcublas. To find ND4J backends on Maven Central, do the following:

1. Click the linked version number under Latest Version.
2. Copy the dependency code on the left side of the subsequent screen.
3. Paste the code into your project root’s *pom.xml* in IntelliJ.

The nd4j-x86 backend will look something like this:

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-x86</artifactId>
  <version>${nd4j.version}</version>
</dependency>
```

The ND4J backend nd4j-x86 works with all examples. To install an additional dependency, OpenBlas, Windows and Linux users should refer to the DL4J Getting Started page.

APPENDIX I

Setting Up GPUs for DL4J Projects

Vyacheslav Kokorin and Susan Eraley

Training neural networks involves a host of linear algebra calculations. Graphics processing units (GPUs), with their thousands of cores, were designed to excel at such problems. They are therefore often used to speed up training and eventually attain better compute performance per dollar, per watt.

Switching Backends to GPU

DL4J can be used with nVIDIA GPUs with support currently for Cuda 7.5. Cuda 8.0 will be supported when available. DL4J is written to be plug-and-play. This means that switching the compute layer from CPU to GPU is as easy as switching out the `artifactId` line under the `nd4j` in the dependencies in the `pom.xml` file.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-cuda-7.5-platform</artifactId>
            <version>${nd4j.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Picking a GPU

In general, a high-end consumer-grade model or a professional Tesla device is recommended: as of this writing, a pair of NVidia GeForce GTX 1070 GPUs would be a solid choice with which to begin.

Here are features to consider when buying a GPU:

Amount of multiprocessors (or cores) on board

The more, the better. As simple as that. This relates to how many parallel threads the GPU can process at any given moment.

Memory available on device

This defines how much data can be uploaded to the device for processing. However, the memory type also matters because it dictates the memory bandwidth and therefore the transfer speed. GDDR5 is considered the absolute minimum. GDDR5X is better, and HBM/HBM2 is the top choice.

Another feature worth mentioning is optional native support for half precision, which is available in the top-end Tesla P100 devices and special Tegra devices. This feature is able to boost deep learning speeds up to 200 to 300 percent, depending on the data dimensionality and model size.

It is also important to consider the interconnect options available between devices. As of this writing, there are only two options available on the market, PCIe and NVLink, with NVLink being the clear winner by being capable of providing 160 GBps bandwidth. This makes NVLink a highly desirable feature for multi-GPU systems for all possible deep learning parallelism models. NVIDIA even offers a prebuilt server powered with eight Tesla P100 devices that are NVLink interconnect-enabled, called the DGX-1. It comes at a hefty price tag with the clever tagline “Supercomputer in a Box.” Clever or not, the tagline is a hard one to refute.

Training on a Multiple GPU System

DL4J also supports training on a multi-GPU system in data-parallel mode. DL4J has a `ParallelWrapper` class that you can use to convert an existing model to one that will train in parallel.

Here's a simple example:

```
ParallelWrapper wrapper = new ParallelWrapper.Builder(YourExistingModel)
    .prefetchBuffer(24)
    .workers(4)
    .averagingFrequency(1)
    .reportScoreAfterAveraging(true)
    .build();
```

`ParallelWrapper` takes the existing model as primary argument, and then carries out the training in parallel. In the case of GPUs, it's a good idea to keep the number of workers equal to or higher than the number of GPUs. Exact values are subject to tuning, because they are task- and hardware-dependent.

Within the `ParallelWrapper`, the initial model will be duplicated, and each worker will be training its own copy of the model. After every X iteration, defined by

`averagingFrequency(X)`, all models will be averaged and then copied to the workers. Training continues in this manner.

It's worth noting that for data-parallel training, a higher learning rate is recommended. Something around +20 percent should be a good starting value.

CUDA on Different Platforms

The following list shows where to find extended instructions for CUDA on different platforms.

- CUDA on Linux
- CUDA on Windows
- CUDA on OS X

Monitoring GPU Performance

After you begin training neural networks on GPUs, you will want to monitor whether and how well the GPUs are working. In the section that follows, we list some resources to help debug and monitor GPU operations.

NVIDIA System Management Interface

Install the [NVIDIA System Management Interface \(SMI\)](#).



Using the NVIDIA SMI

There will be many things in the logs, but look for “Java” in the output to better triage what ND4J is doing.

Troubleshooting DL4J Installations

If anything goes wrong when you attempt to run the examples, you'll need to do some troubleshooting. In the following sections we discuss some of the common issues that new users of DL4J experience.

Previous Installation

If you have installed DL4J before and now see the examples throwing errors, please update your libraries. With Maven, just update the versions in your *pom.xml* file to match the latest versions on [Maven Central](#). With source, you can run a `git clone` on ND4J, Canova, and DL4J, and a `mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true` within all three directories, in that order.

Memory Errors When Installing From Source

As the code base grows, installing from source requires more memory. If you encounter a `Permgen` error during the DL4J build, you might need to add more heap space. To do that, you'll need to find and alter your hidden *.bash_profile* file, which adds environmental variables to bash. To see those variables, in the command line, type `env`. To add more heap space, in your console, type this command:

```
echo "export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=512m"" > ~/.bash_profile
```

Older Versions of Maven

Older versions of Maven, such as 3.0.4, are likely to throw exceptions like a `NoSuchMethodError`. You can fix this by upgrading to the latest version of Maven, which is currently 3.3.x. To check your Maven version, in the command line, type `mvn -v`.

Maven and PATH Variables

After installing Maven, you might receive a message like this:

```
mvn is not recognised as an internal or external command,  
operable program or batch file.
```

This means that you need Maven in your PATH variable, which you can change like any other environmental variable.

Bad JDK Versions

If you see:

```
Invalid JDK version in profile 'java8-and-higher': Unbounded range:  
[1.8, for project com.github.jai-imageio:jai-imageio-core  
com.github.jai-imageio:jai-imageio-core:jar:1.3.0
```

your installation may have a Maven issue. You should update to version 3.3.x.

C++ and Other Development Tools

To compile some ND4J dependencies, install some development tools for C and C++.



ND4J Specific Instructions

For specific instructions, see our ND4J guide at <http://nd4j.org/getstarted.html#devtools>.

Windows and Include Paths

The include path for Java CPP doesn't always work on Windows. One workaround is to take the header files from the include directory of Microsoft Visual Studio and put them in the include directory of the Java Run-Time Environment (JRE), where Java is installed.



standardio.h

This will affect files such as *standardio.h*. You can find more information at <http://nd4j.org/getstarted.html#windows>.

Monitoring GPUs

As mentioned [Appendix I](#) we can monitor graphics processing units (GPUs) with the NVIDIA System Management Interface (SMI).



For additional help with monitoring GPUs, go to <http://nd4j.org/getstarted.html#gpu>.

Using the JVisualVM

One major reason to use Java is its prebaked diagnostics in the JVisualVM. In the command line, type **jvisualvm** and the system will display visuals on the local CPU, Heap, PermGen, Classes, and Threads.



The Sampler View

A useful view in this tool is the sampler view. In the upper right, click the Sampler tab, and then select the CPU or Memory button for visuals.

Working with Clojure

When using `deeplearning4j-nlp` from a Clojure application and building an uberjar with Leiningen, it is necessary to specify the following in the `project.clj` so that the akka `reference.conf` resource files are properly merged:

```
:uberjar-merge-with {#"\\.properties$" [slurp str spit] "reference.conf"
  [slurp str spit]}
```

(Note that the first entry in the map for the `properties` file is the usual default). If this is not done, the following exception will be thrown when trying to run from the resulting uberjar:

```
Exception in thread "main" com.typesafe.config.ConfigException$Missing:
  No configuration setting found for key 'akka.version'
```

OS X and Float Support

Float support is buggy on OS X. If you see NaNs where you expect numbers running our examples, switch the data type to `double`.

Fork-Join Bug in Java 7

There is a bug in fork-join in Java 7. Updating to Java 8 fixes it. If you get an `OutOfMemoryError` error that looks like the following, fork-join is the problem:

```
java.util.concurrent.ExecutionException: java.lang.OutOfMemoryError ...java.util.concurrent.ForkJoinTask.getThrowableException(ForkJoinTask.java:536)
```

Online Support Through Gitter

Please feel free to ask us about error messages on our [Gitter Live Chat](#).

When you post your question, include the following information (it will really speed things up!):

- Operating system (Windows, OS X, Linux) and version
- Java version
- Maven version
- Stacktrace

Precautions

In the following sections we list some things to check when the DL4J examples don't build or run correctly.

Other Local Repositories

Ensure that you have not cloned other repositories locally. The main DL4J repository is undergoing continuous improvements, the latest of which might not be thoroughly tested with examples.

Check Maven Dependencies

Ensure that all of your dependencies for examples are downloaded from Maven rather than found locally. To remove older dependencies, type the following:

```
rm -rf ls ~/.m2/repository/org/deeplearning4j
```

Reinstall Dependencies

To rebuild the examples from source and ensure that its installed correctly, in the `dl4j-0.4-examples` directory, type the following:

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

If All Else Fails

If you have trouble, the first thing you should check is your *pom.xml* file.

Different Platforms

To compile certain ND4J dependencies, you will need to install some development tools for C, including gcc. To check whether you have gcc, on your terminal or command prompt, type **gcc -v**.

OS X

Some versions of the Apple developer tool Xcode will install gcc for you. If you do not already have gcc, at the command prompt, type the following:

```
brew install gcc
```

Windows

Windows users might need to install Visual Studio Community 2013, which is free. You can download it from <https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>.



Setting up PATH Environment Variables

You will need to add Visual Studio's path to your PATH environment variable manually. The path will look something like this:

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin
```

To determine whether Visual Studio's path is set up correctly at the command prompt, type the following:

```
cl
```

You might get a message informing you that certain *.dll* files are missing. Ensure that that the VS/IDE folder is in the path (see the preceding Warning). If the command prompt returns usage information for the *cl* command, it's in the right place.

Setting up Visual Studio

To set up the and configure Visual Studio, perform the following steps:

1. Set up the PATH environment variable to point to *\bin* (for *cl.exe*, etc.).

- Also try running `vcvars32.bat` (also in bin) to set up the environment before invoking `mvn clean install` on ND4J (it can save you from copying headers around).



vcvars32

`vcvars32` might be temporary, so you might need to run it every time you want to do ND4J `mvn install`.

After installing Visual Studio 2015 and setting the PATH variable, you need to run the `vcvars32.bat` to set up the environment variables (INCLUDE, LIB, LIBPATH) properly so that you don't need to copy header files. But, if you run the `.bat` file from Explorer, because the settings are temporary, they're not properly set. Therefore, run `vcvars32.bat` from the same CMD window as your `mvn install`, and all the environment variables will be set correctly.

Here is how they should be set:

```
INCLUDE = C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include LIB =
"C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\lib" //so you can
//link to .lib
//files^^
```



Remember to Click C++

In Visual Studio, you also need to click C++. It is no longer set by default.



Java CPP and Windows

In addition, the include path for **Java CPP** doesn't always work on Windows. One workaround is to take the the header files from the include directory of Visual Studio and put them in the include directory of the Java Run-Time Environment (JRE), where Java is installed. This will affect files such as `standardio.h`.

Working with Windows on 64-bit platforms

You can get the `.dll` for the Windows 64 platform at <http://avulanov.blogspot.cz/2014/09/howto-to-run-netlib-javabreeze-in.html>.

Download the `.dll` libraries and place them in the Java bin folder (e.g., `C:\prg\Java\jdk1.7.0_45\bin`).



Other Dependencies

The library *netlib-native_system-win-x86_64.dll* depends on: *libgcc_s_seh-1.dll*, *libgfortran-3.dll*, *libquadmath-0.dll*, *libwinpthread-1.dll*, *libblas3.dll*, and *liblapack3.dll*. (*liblapack3.dll* and *libblas3.dll* are just renamed copies of *libopenblas.dll*)

You can download compiled libraries from <http://www.openblas.net/>.

Linux

If you use either Ubuntu and Centos, you will need to follow two separate sets of instructions.

Ubuntu

For Ubuntu, first type:

```
sudo apt-get update
```

Then, you'll need to run a version of this command:

```
sudo apt-get install linux-headers-$(uname -r) build-essential
```

`$(uname -r)` will vary according to your Linux version. To get your Linux version, open a new window of your terminal and type this command:

```
uname -r
```

You will see something like this: `3.2.0-23-generic`. Whatever you see, copy and paste it into the first line of the script in place of `$(uname -r)`. Then, insert one space and type **build-essential**. Watch out for typos. You can press tab to complete any command.

Centos

In your terminal (or ssh session), type the following as a root user:

```
yum groupinstall 'Development Tools'
```

After that, you should see a lot of activity and installs on the terminal. To verify that you have, for example, gcc, type this line:

```
gcc --version
```

For more complete instructions, go to <http://www.cyberciti.biz/faq/centos-linux-install-gcc-c-c-compiler/>.

Index

Symbols

.fit() method, 174
3D volumetric input, 146
<scope> tag, 377

A

accuracy, 38
acknowledgements, xix
activation functions
 definition of term, 53
 evolution of in practice, 95
 for general architecture, 94
 for hidden layers, 94
 hard tanh, 68, 255
 in deep networks, 93-95
 (see also deep networks)
 linear, 66
 output layer for binary classification, 95
 output layer for multiclass classification, 95
 output layer for regression, 95, 242
 rectified linear, 69, 242
 sigmoidal, 65, 94, 254
 softmax, 68, 245
 summary table for, 255
 tanh, 67, 252
 tuning techniques, 253-256
activation maps, 133
AdaDelta, 102, 261
AdaGrad, 102, 261
Adam, 102, 261
adjacency matrices, 355
adversarial training, 279
AlexNet, 85, 142
AlphaGo, 420

Alternating Least Squares, 15
anomaly detection
 using autoencoders, 207-213
 using variational autoencoders, 214-220
Apache Hadoop, 268, 325, 357
 (see also Spark and Hadoop)
Apache Nutch project, 359
Apache Spark (see Spark and Hadoop)
ApplicationMasters, 364
Area Under the Curve (AUC), 280
artificial intelligence (AI), 405-415
 Artificial Winters of, 412
 vs. deep learning, 406
 definition of term, 407
 history of, 405, 407
 vs. machine learning, 409
 modern definitions of, 408-412
 renewed interest in, 413
 study of intelligence, 407
 underpinnings of, 3
artistic style, modeling, 90
Async N-step Q-Learning, 438
Asynchronous Reinforcement Learning, 438
attributions, xvii
audio data, 240
autoencoders, 112, 118, 207-213
automatic feature extraction, 6, 13, 88, 119, 322

B

backpropagation learning
 algorithm intuition, 57
 backpropagation pseudocode, 61-65
 fractional error responsibility and, 64
 gentle, 120

- loss functions, 61
mathematics underlying, 443-447
mini-batch training and, 65, 104, 129, 147
neural network notation, 58-60
origins of, 57
process of, 42
backpropagation through structure (BPTS), 161
backpropagation through time (BPTT), 157, 310
bag-of-words model, 348
batch learning algorithms, 266
batch normalization, 139
Bayes's theorem, 19
Bayesianism, 17
bell-curve (see normal distribution)
Bellman equation, 427-429
Bernoulli distribution, 334
biases, 53
Bidirectional Recurrent Neural Networks (BRNN) , 150
big data, 267
binarization, 334
binary classification
 examples of, 25
 hinge loss, 257
 output layer for, 95
 single output vs. two outputs, 243
biological neural networks, 4, 43-45, 54
blended networks, 159
bootstrapping, 22
Broyden-Fletcher-Goldfarb-Shanno (BFGS), 99, 263
- C**
- C and C++, 488
categorical data attributes, 325
centering, 329
central limit theorem, 21
CIFAR-10 dataset, 126
class imbalance, 39, 280-283
classification
 definition of term, 25
 loss functions for, 75
 model output layer, 243-245
 more than two labels, 243
 multiclass models, 244
 multilable models, 245
 single-label, 243
Clojure, 489
- clustering, 26
code
 full import listings, xvii
 using examples, xvii, 166
Collaborative Filtering, 26
columnar data, modeling, 166, 240, 325, 335, 465
comments, xviii
computational efficiency, 274
ComputationGraph, 245
computer vision, 85, 159
conditional probability, 18, 34
confusion matrix, 36-40
Conjugate Gradient Methods, 15, 99, 263
connection weights, 53, 251, 282
connectionist models, 148
 (see also Recurrent Neural Networks)
contact information, xviii
contrastive divergence, 109
convergence, 28
convex optimization, 29
convolution operation, 131
convolutional layers
 activation maps, 133
 batch normalization, 139
 concept of convolution, 131
 filters in, 132, 299-301
 hyperparameters in
 filter size, 138
 output depth, 139
 overview of, 138
 stride, 139, 298
 zero-padding, 139, 299
 learned filters and renders, 137
 parameter sharing, 136
 purpose of, 130
 receptive fields, 136
 ReLU activation functions as, 138
 tuning techniques, 297-303
Convolutional Neural Networks (CNNs)
 architecture overview, 128
 biological inspiration for, 126
 convolutional layers, 130-139
 correct application of, 126, 166
 vs. Deep Belief Networks, 121
 efficacy of, 125
 evolution in applications of, 167
 evolution of, 84
 fully-connected layers, 140

goal of, 125
image representation in, 45, 337, 467
input layers, 130
intuition, 126
layers in, 81, 87
modeling hand-written images, 182-190
normalization and, 333
notable CNNs, 141
other applications of, 141
pooling layers, 140, 303
raw image data and, 167
Spark model, 397-403
 job configuration and data loading, 400
 LeNet architecture and training, 401
tuning techniques
 common architectural patterns, 294-297
 configuring convolutional layers, 297-303
 configuring pooling layers, 303
 convolution stage, 293
 detector stage, 294
 transfer learning, 304
cosine similarity, 11
cross entropy, 78, 96, 258
cross-validation, 22
CSV data (see columnar data)
CUDA, 272, 485
curriculum learning, 279

D

data cleaning, 328
data mining, 3, 265
data parallelism, 267
data types
 audio, 240
 columnar, 166, 175-182, 240, 335, 465
 columnar raw data attributes
 interval, 326
 nominal, 325
 ordinal, 326
 ratio, 327
 graph structures, 354
 images, 166, 182-190, 207-213, 236, 240, 336-339, 467
 nonshuffled data, 288
 requiring vectorization, 322
 text, 167, 347-354
timeseries sequences, 167, 240, 340-346
video, 240, 337

data wrangling, 470
DataSet class, 456
DataSet objects, 173
DataSetIterator class, 173
DataVec
 benefits of, 170
 ETL and vectorization with, 334
 goal of, 463
 image data and vector normalization with, 339
loading CSV data for multilayer perceptrons, 465-467
loading data for machine learning, 463-465
loading image data for CNNs, 467-468
loading sequence data for RNNs, 469-470
main categories of functionality, 463
transforming data, 470-474
vectorizing sequential data with, 341-346

Dean, Jeff, 270

debugging
 bad JDK versions, 488
 C++ and other development tools, 488
 Clojure, 489
 fork-join bug in Java 7, 490
 JVisualVM, 489
 Maven and PATH variables, 488
 memory errors, 487
 monitoring GPUs, 489
 older versions of Maven, 487
 OSX and Float support, 489
 previous installations, 487
 Spark and Hadoop, 379
 understanding debug output during training, 257
 visual monitoring for, 435
 Windows and include PATHS, 488

decentralized systems, 5

decision boundary, 28

deconvolutional network (deconvnet), 122

Deep Belief Networks (DBNs)
 automatic learning, 119
 categorization of, 88
 vs. Convolutional Neural Networks, 121
 feature extraction, 119
 fine-tune phase, 120
 layers in, 87
 network architecture, 118
 role in deep learning, 121
 tuning techniques

- hidden unit count, 319
 - two-phase training, 317
 - using Momentum, 318
 - using regularization, 319
 - Deep Convolutional Generative Adversarial Network (DCGAN), 123
 - deep learning
 - vs. artificial intelligence, 406
 - benchmark records achieved by, 86
 - definition of learning, 2
 - definition of term, 6, 81
 - generative mechanics demonstrated by, 7
 - impacts of, 87
 - vs. machine learning, 3, 6, 162
 - role of Deep Belief Networks in, 121
 - successful application of, 91, 162
 - when to use, 163
 - deep networks
 - architectural principals of
 - activation functions, 93
 - hyperparameters, 100-104
 - layers, 93
 - loss functions, 95
 - optimization algorithms, 96
 - parameters, 92
 - architecture selection, 165-169, 240-242
 - building blocks of, 105-115
 - autoencoders, 112
 - overview of, 105
 - unsupervised layer-wise pretraining, 106
 - variational autoencoders, 114
 - vs. feed-forward multilayer, 81
 - intuition for building, 238-240
 - major architectures of
 - Convolutional Neural Networks, 125-143
 - evolution of, 82
 - overview of, 117
 - Recurrent Neural Networks, 143-160
 - recursive neural networks, 160
 - selecting, 162, 165-169, 240-242
 - unsupervised pretrained networks, 118-125
 - Deep Q-Networks (DQNs), 417
 - (see also Q-Learning)
 - deep reinforcement learning, 83, 422
 - Deeplearning4j library
 - activation functions in layers, 138
 - basic concepts of, 172-175
 - getting input for models, 173
 - loading and saving models, 172
 - setting up model architecture, 173
 - training and evaluation, 174
 - benchmarking, 171
 - computational efficiency of, 274
 - DL4J project set up
 - creating new projects, 477-479
 - GPU set up, 483-485
 - IDEs, 480
 - other Maven POMs, 480
 - focus of, 170
 - functions performed by, 169-171
 - scientific computing, 170
 - vectorization, 170
 - image data representation in, 337
 - image training data arrays in, 129
 - matching networks with data types, 165-169
 - memory and precision and, 251
 - multilayer perceptron model setup, 175-182
 - network statistics tool, 284-291
 - parallelization architecture, 269
 - Spark best practices, 385
 - support for Hadoop environment, 360, 381
 - troubleshooting
 - bad JDK versions, 488
 - C++ and other development tools, 488
 - Clojure, 489
 - fork-join bug in Java 7, 490
 - JVisualVM, 489
 - Maven and PATH variables, 488
 - memory errors, 487
 - monitoring GPUs, 489
 - older versions of Maven, 487
 - OSX and float support, 489
 - previous installations, 487
 - Windows and include PATHS, 488
 - tuning DL4J jobs on Spark, 371
 - updaters in, 102, 259
 - using with ND4J, 459-461
 - working from source, 475
- DeepMind, 430
- DeepWalk, 354
- dimensionality reduction, 333
- discrete data attributes, 325
- discriminative models, 33
- DistBelief, 270
- distributed file systems, 386
- distributions, 19-22

central limit theorem, 21
continuous vs. discrete, 19
long-tailed, 21
normal, 19
rank-frequency, 21
DL4J (see Deeplearning4j library)
Doc2Vec, 227
document classification, 231-236
dot product, 11
Downpour SGD, 269
DropConnect, 104, 279
Dropout, 103, 277-278, 319
“dying ReLU” issue, 70, 242, 255

E

element-wise product, 11
entropy, 77
enumerated data attributes, 325
epochs, 273-275
executors (Spark), 358, 365-392
experience replay, 430
expert domain knowledge, 324
exploding gradients, 311
Extract, Transform, and Load (ETL), 325, 328, 334, 387, 463, 470

F

F1 score, 39, 280
FaceNet, 236
facial recognition, 236
feature binarization, 334
feature detectors, 131
feature engineering
 definition of term, 327
 feature copying, 328
 move toward automation, 322
 techniques of, 328
feature extraction, 6, 13, 88, 119
feature maps (see activation maps)
feed-forward multilayer network
 artificial neuron evolution
 activation functions, 53
 artificial neuron input, 51-53
 biases, 53
 connection weights, 53, 251
 diagram of, 50
 backpropagation in, 445-447
 correct application of, 166
 vs. deep networks, 81

definition of term, 50
learning algorithm caution, 58
modeling CSV data with, 175-182
multilayer Spark example, 387-392
network architecture
 connections between layers, 56
 hidden layer, 55, 246
 input layer, 55
 layer concept, 54
 output layer, 55
 topology, 42
 tuning techniques, 246-251
filter renders, 137
fine-tune phase, 120
.fit() method, 174
fitting, 25
forget gate, 155
fork-join bug, 490
forward propagation, 53
fractional error responsibility, 64
frequentism, 17
fully-connected layers, 140
functional parallelism (see task parallelism)

G

garbage collection, 370
Gated Recurrent Units (GRUs), 87, 156, 253
gaussian distribution (see normal distribution)
Gaussian Elimination, 14
Generative Adversarial Networks (GANs)
 conditional GANs, 124
 DCGAN networks, 123
 description of visual output, 90
 discriminator network, 122
 drawbacks of, 125
 generative network, 122
 key aspect of, 121
 training generative models, 121
 vs. variational autoencoders, 124
generative mechanics, 7
generative models
 building, 123
 vs. discriminative, 33
 drawbacks of, 125
 Generative Adversarial Networks, 90, 121-125
inceptionism, 89
modeling artistic style, 90
Recurrent Neural Networks, 90

types of, 89
GitHub projects, cloning, 476
global minimum, 32
global vectors, 227
GloVe, 227
Google DeepMind, 430
Google File System, 266
GoogLeNet, 142
Gov2Vec, 235
GPUs (graphical processing units)
 Mesos and, 367
 monitoring, 489
 setting up for DL4J projects, 483-485
 vectorized math and, 272
gradient descent, 30, 98
graphs, modeling, 354

H

Hadamard product (see element-wise product)
Hadoop (see Spark and Hadoop)
Hadoop Distributed File System (HDFS), 169, 358, 386
Hadoop YARN (Yet Another Resource Negotiator) framework, 358
HAL 9000 computer, 86
hard tanh activation function, 68, 255
Hessian matrix, 33, 97
Hessian-free optimization, 100, 263
hidden layers
 determining count, 246
 in feed-forward multilayer networks, 55
 in RBM network, 107
Hidden Markov Chains, 419
hinge loss, 75, 257
hit rate (see true positive rate)
hybrid architectures, 169, 314
hyperparameters (see also tuning techniques)
 categories of, 100
 in convolutional layers, 138
 definition of term, 100
 large parameter count, 101
 layer size, 100
 learning rate, 78, 101-103, 258-263
 magnitude group, 101
 mini-batching, 104
 momentum, 79, 258
 Nesterov's momentum, 102, 261
 normalization and, 329
 purpose of, 28

receptive fields, 136
regularization, 79, 103
selecting, 78, 100
sparsity, 80, 263, 319
hyperplanes, 10

I

identity covariance matrix, 333
image classification benchmark dataset, 127, 182

image data
 anomaly detection, 207-213
 facial recognition, 236
 modeling, 166, 182-190, 240, 467
 vectorization of, 336-339

ImageRecordReader (DataVec), 339

inceptionism, 89

INDArrays, 450, 459

informational theory, 78

inner product (see dot product)

input layers
 in Convolutional Neural Networks, 130
 in feed-forward multilayer networks, 55
 lack of activation functions in, 94

IntelliJ, 480

interval values, 326

iris dataset, 11

Item2Vec, 236

iterative methods, 14

J

Jacobian matrix, 33, 97
jar size, controlling, 377
Java Development Kit (JDK), 478, 488, 490
Java serialization, 377
Jeff Dean's 12 Numbers, 441
JVisualVM, 489
JVM tuning, 370

K

K-means clustering, 26
Kerberos, 361, 386
kernel hashing, 353
kernels, 132
Kyro dependency, 377, 380

L

L-BFGS optimization algorithm, 99, 263

L1 and L2 penalty methods, 104, 242, 275
law of large numbers, 422
layers
 deconvolutional layers, 122
 in deep networks, 93
 evolutionary progress of, 87
 in CNNs, 128-141
 in feed-forward multilayer networks, 42
 in LSTM networks, 156
 layer count, parameter count, and memory, 246-251
 layer size hyperparameter, 100
leaky ReLUs, 70, 252, 254
learned filters, 137
learning rate
 definition of term, 65, 101-103
 parameter adjustment and, 78
 tuning techniques, 258-263
lemmatization, 347
LeNet CNN, 142, 182-190, 401
likelihood, 23
linear activation function, 66
linear algebra
 converting data into vectors, 11
 hyperplanes, 10
 mathematical operations
 dot product, 11
 element-wise product, 11
 outer product, 11
 matrices, 10
 ND4J library and, 170
 scalars, 9
 solving systems of equations, 13-15
 direct method, 14
 iterative methods, 15
 methods for, 14
 tensors, 10
 vectors, 9
linear regression
 goal of, 23
 model setup, 23
 plotted visualization, 24
 relating the model, 25
local minima, 32, 42, 248
logistic loss functions, 75, 258
logistic regression, 34-36
Long Short-Term Memory (LSTM) networks
 advances in neuron types, 87
 applications for, 169
critical component of, 150
example use cases, 150
vs. Gated Recurrent Unit, 156
layers in, 156
LSTM block, 153
modeling sequence data
 classifying sensor time-series sequences, 200-207
 generating the works of Shakespeare, 191-200
network architecture, 151
normalization and, 333
orthogonal weight initialization, 253
properties of, 150
Spark model, 392-397
 LSTM architecture, 395
 training, tracking and results, 396
training, 156
 training complexity in, 151
 tuning techniques, 311
 unit connections, 153
loss functions
 for classification, 75
 for reconstruction, 77, 96
 for regression, 72-75, 242
 loss function notation, 71
 matching with activation functions, 239
 pseudocode representation, 61
 purpose of, 28, 71, 95
 summary table of, 257
 tuning techniques, 256-258
 weighted, 282
lower upper (LU) decomposition, 14
LSTM block, 153
LSTM Memory Cell, 87, 153

M

machine learning
 vs. artificial intelligence, 409
core concepts
 linear algebra, 8-15
 statistics, 15-23
vs. data mining, 3
vs. deep learning, 6
definition of term, 2, 96
history of, 1
hyperparameters, 78-80
logistic regression, 34-36
model evaluation, 36-40

optimization methods, 23-34 (see also optimization)
classification, 25
clustering, 26
convex optimization, 29
generative vs. discriminative models, 33
gradient descent, 30
parameter optimization, 27
Quasi-Newton optimization methods, 33
regression, 23
stochastic gradient descent, 32, 65
underfitting and overfitting, 26, 79
successful application of, 162
when to use, 164
workflow setup, 8

MapReduce, 266-269, 357, 359, 366

Mark I Perceptron, 46

Markov Decision Process (MDP), 417-418

Markov models, 148

masking, 147, 312-314, 346

matrices, 10

matrix decomposition, 14

matrix inversion, 14

Maven Project Object Model (POM)
DL4J project creation, 477-479
IDEs, 480
major dependencies, 372
older versions of Maven, 487
other Maven POMs, 480
platform dependency, 374
POM file for CDH 5.x, 378
POM file for HDP 2.4, 378
pom.xml file dependency template, 374-378
working with DL4J from source, 476

max pooling, 140

Max-Norm regularization, 276

maximum likelihood estimation (MLE), 29

MCXENT loss function, 243

mean, 330

mean absolute error (MAE) loss, 74

mean absolute percentage error (MAPE) loss, 74

mean squared error (MSE) loss, 72, 242

mean squared log error (MSLE), 74

memory cells, 153

memory requirements, 250, 274, 368, 371, 451, 484, 487

Mesos, 363, 367

min-max scaling, 332

mini-batch training, 32, 65, 104, 129, 147, 273-275, 337

minimum point (see stationary point)

missing values, dealing with, 329

MLLibUtil class, 459

MNIST handwriting benchmark
anomaly detection, 207-213
evolution of, 84
modeling hand-written images, 182-190
reconstructing MNIST digits, 214-220
reconstruction in RBMs and, 110
Spark model, 397-403

model-free reinforcement learning, 419

models
building from raw data, 324
DL4J basic concepts, 172-175
getting input for models, 173
loading and saving models, 172
setting up model architecture, 173
training and evaluation, 174
evaluating, 36-40, 280, 313
forms of, 2
generative models, 89
generative vs. discriminative, 33
matching networks with data types, 165-169, 240-242
model calibration, 281
purpose of, 1
relating model goal and output layers, 242-245
momentum, 79, 258

Momentum, 318

Monte Carlo methods, 422

multiclass classification, 244

multilabel classification, 244

multilayer feed-forward network (see feed-forward multilayer network)

multiple classifications, output layer for, 69, 95

multivariate Bernoulli distribution, 334

N

n-dimensional arrays, 449

n-gram vectorization, 353

Nash equilibrium, 420

Natural Language Processing (NLP)
bag-of-words model, 348
distributed representations of sentences, 227-231
document classification, 231-236

- filter sizes in, 132
learning word embeddings, 221-226
sentiment analysis, 126
techniques for deep learning, 221
- ND4J (N-Dimensional Arrays for Java) library
acceleration of training with, 171
benchmarking, 171
creating input vectors, 457-458
DataSet class, 456
debugging, 380
design and basic usage, 450-456
general syntax, 452
understanding NDArrays, 450
working with NDArrays, 453-456
- interoperability, 459
lack of support for first-class type uint8, 431
main features, 449
Maven and, 480
memory and precision and, 251
runtimes and, 170
user guide, 449
using with DL4J, 459-461
- ND4S (N-Dimensional Arrays for Scala), 449
- NDArrays, 173, 450
negative log likelihood, 76, 180, 258
Nesterov's momentum, 102, 261
neural networks
activation functions, 65-70
biological inspiration for, 4, 43-45, 54
evolution of, 6
evolutionary progress and resurgence, 83-91
automated feature learning, 88
better labeled data, 84
computer vision, 85
feature learning, 89
generative modeling, 89
hybrid architectures, 87, 169
layer types, 87
network architecture, 87
neuron types, 87
optical character recognition, 84
technologies benefiting from, 85, 87
feed-forward multilayer, 42, 50-56
fundamental units of, 1, 5
history of, 1
long-term information storage in, 41
loss functions, 71-78
network architecture, 41, 441
parameter vectors, 2
- perceptron model, 45-49
training, 56-65
- NeuralNetConfiguration object, 173
no free lunch theorem, 163
- Node2Vec, 235, 354
- NodeManagers, 364
- nominal values, 325
- nonshuffled data, 288
- normal distribution, 19
- Normal Equations, 14
- normalization, 11, 329-334
- nVIDIA GPUs, 483
- nVIDIA System Management Interface (SMI), 485

0

- odds, vs. probability, 17
- one-hot vector representation, 180, 326
- online learning algorithms, 266
- optical character recognition, 84
- optimization
alternate algorithms, 97
classification, 25
clustering, 26
conjugate gradient, 99, 263
convex optimization, 29
definition of term, 23
first- vs. second-order algorithms, 96
first-order methods, 97
functions at work in, 28
generative vs. discriminative models, 33
gradient descent, 30, 98
Hessian-free optimization, 100, 263
Jacobian matrix, 97
parameter optimization, 27
practical usage of, 97
Quasi-Newton methods, 33, 99
regression, 23-25
second-order methods, 98, 263
stochastic gradient descent, 32, 65, 98
tuning techniques, 263-265
underfitting and overfitting, 26, 79
- optimization efficiency, 274
- ordinal values, 326
- orthogonal weight initialization, 253
- outer product, 11
- output layers
for binary classification, 95, 243
for multiclass classification, 95, 244

for regression, 95, 242
in feed-forward multilayer networks, 55
overall objective and, 120
relating to model goal, 242-245
overfitting, 26, 79, 101, 103, 283

P

paragraph vectors, 227-236
parallelization, 265-271, 280, 372
ParallelWrapper class, 484
parameters
 in deep networks, 92
 NDArrays and, 93
 parameter optimization, 27
 parameter sharing, 136
 parameter vectors, 2
 parameter-averaging technique, 267, 270
 tuning techniques, 246-251
Pavlovian conditioning, 417
perceptron model
 definition of, 46
 history of, 46
 influence of biological neurons on, 48
 learning algorithm, 48
 limitations of single-layer, 49
 multilayer, 50, 166, 175-182, 445-447
 multilayer Spark example, 387-392
 precursor to, 45
 single-layer, 47
 summation function parameters, 47
permission, obtaining, xvii
PhysioNet Challenge, 40, 280
pi calculation, 422
policy iteration, 422-424, 426
pooling layers, 140, 303
populations vs. samples, 22
positive prediction value, 39
post scaling, 282
posterior probability, 19
practitioners
 data scientists, xv
 definition of term, xiv
 Java engineers, xv
 matching input data to architecture, 91
precision, 39
pretraining, 106, 109
principal component analysis (PCA), 333
prior functions, 275
prioritized replay, 435

probabilistic sampling, 282
probability
 Bayesian vs. frequentist, 17
 conditional probability, 18, 34
 expression of, 16
 joint, 34
 vs. odds, 17
 posterior probability, 19
 probability distributions, 19-22

Q

Q-Learning
 Bellman equation, 427-429
 clipping, 435
 double Q-Learning, 434
 experience replay, 430
 exploration vs. exploitation, 426
 goal of, 421
 history processing, 434
 image preprocessing, 431-433
 implementation, 429
 initial state sampling, 429
 mean Q-values, 438
 modeling Q(s, a), 430
 policy iteration, 422-426
 prioritized replay, 435
 scaling rewards, 435
Quasi-Newton optimization methods, 33, 99
questions, xviii

R

ratio values, 327
raw iris dataset, 323
RDD (Resilient Distributed Datasets), 358
recall, 39
receptive fields, 126, 136
 (see also visual fields)
recommendation, 26
reconstruction, loss functions for, 77, 96
RecordReaders class, 173
rectified linear activation functions, 69, 242
Recurrent Models of Visual Attention, 159
Recurrent Neural Networks
 3D volumetric input, 146
 applications for, 90
 vs. backpropagation through time, 157
 benefits and drawbacks of, 143
 domain-specific applications, 159
 history of, 84

layers in, 81, 87
LSTM networks, 150-158
vs. Markov models, 148
model input and output, 145
modeling sequence data
 classifying sensor time-series sequences, 200-207
 generating the works of Shakespeare, 191-200
 using DataVec, 469
modeling the time dimension, 143
network architecture, 149
normalization and, 333
timeseries sequences and, 167
tuning techniques
 challenges of, 306
 debugging LSTMs, 311
 hybrid architectures, 314
 input data and input layers, 307
 output layers and RNNOutputLayer, 308
 padding and masking, 312-314
 training the network, 309-310
 vanishing gradient problem, 149

Recursive Neural Networks
 applications of, 161
 network architecture, 160
 vs. Recurrent Neural Networks, 160
 varieties of, 161

Recursive Neural Tensor Network (RNTN), 161

regression
 definition of term, 23
 linear regression, 23
 logistic regression, 34-36
 loss functions for, 72-75
 normalization and, 333
 output layer for, 95, 242

regularization, 79, 275-280, 284, 290, 310, 317, 319

reinforcement learning
 definition of term, 82, 417
 Markov Decision Process (MDP), 417-418
 model-free, 419
 observation setting, 419
 offline vs. online, 429
 single-player vs. adversarial games, 420
 terminology, 418
 visual monitoring for debugging, 435-438

relational database management system (RDBMS), 240, 325

ReLU (rectified linear units), 70, 138, 242, 252, 254
renders, 137
resampling methods, 22
ResNet, 142
Restricted Boltzmann Machines (RBMs), 87, 105-112, 118, 238
tuning techniques
 alternate unit types, 316
 hidden units, 315
 regularization, 317
 use cases, 314

RL4J (Reinforcement Learning for the JVM)
 downloading, 438
 prototype in Scala, 424
 working example, 438

RMSProp, 102, 198, 259, 261

rotation estimation (see cross-validation)

rotation invariance, 137

S

sample mean, 330
samples vs. populations, 22
scalar product (see dot product)
scalars, 9
scaling, 329, 332
<scope> tag, 377
seat of consciousness, 45
selection bias, 22
sensitivity, 38
sentiment analysis, 126
sequential data
 graphical representation of, 167
 modeling
 classifying sensor time-series sequences, 200-207
 generating the works of Shakespeare
 RNNmodelshake5, 191-200
 using DataVec, 469
 sources of, 144
 tuning techniques, 241
 vectorization of, 340-346

SerDe (serialization and deserialization), 380

serialization, 377, 380

sigmoid activation function, 52, 65, 94, 254

softmax activation function, 68, 245

softplus activation functions, 70

Spark and Hadoop
 best practices, 385

- command-line operation of Spark, 360-362
configuring and tuning Spark
 foreground vs. background clients, 362
 general tuning guide, 367-370
 running Spark on Mesos, 363
 running Spark on YARN, 364-366
 tuning DL4J jobs, 371
Convolutional Neural Networks
 job configuration and data loading, 400
 LeNet architecture and training, 401
 modeling MNIST, 397-403
DL4J parallel execution on Spark, 381
DL4J support for, 357
Hadoop security and Kerberos, 361
history of Hadoop, 359
LSTM networks, 392-397
Maven Project Object Model (POM)
 major dependencies, 372
 platform dependency, 374
 POM file for CDH 5.x, 378
 POM file for HDP 2.4, 378
 pom.xml file dependency template,
 374-378
minimal Spark training example, 383-385
multilayer perceptron Spark example,
 387-392
 distributed training and model evalua-
 tion, 390
 DL4J job execution, 392
 MLP network architecture, 390
overview of parallel-processing, 358, 372
Spark key components, 358
Spark training on single machines, 382
troubleshooting, 379
sparse Initialization, 252
sparsity, 80, 263, 319
specificity, 38
standardization, 330-332
stationary point, 31
statistical whitening, 333
statistics, 15-23
 conditional probability, 18, 34
 distributions, 19-22
 likelihood, 23
 means and variances, 330
 posterior probability, 19
 probability, 16
 resampling methods, 22
 samples vs. population, 22
selection bias, 22
stemming, 347
stochastic gradient descent (SGD)
 benefits of, 98
 best practices, 265
 definition of term, 32
 iterative methodology of, 15
 learning rate and, 254, 258
 mini-batch variant, 65
 parallelizing, 269
 in Q-Learning, 423
 standardization and, 331
 tuning techniques, 263
stochastic pooling, 279
stride, 139, 298
structural descriptions, 2
 (see also models)
super sampling, 282
Symbolic Aggregated approXimation (SAX),
 343
- T**
- table of confusion (see confusion matrix)
tanh activation function, 67, 252
task parallelism, 267, 372
TD (Temporal difference) error, 428, 435
TD-gammon algorithm, 419
tensor product, 11
tensors, 10
term frequency vector, 349
text data, modeling, 167, 347-354
TF-IDF (term frequency-inverse document fre-
 quency), 349-352
Threshold Logic Unit (TLU), 46
time dimension, modeling, 143
time-series data, 144, 147, 167, 200-207, 240
time-steps, 149
tokenization, 347
transfer learning, 304
true positive rate, 39
truncated BPTT, 157
tuning techniques
 activation functions, 253-256
 basic concepts of, 237-240
 epochs and mini-batch size, 273-275
 for CNNs, 293
 for DBNs, 317-320
 for Recurrent Neural Networks, 306-314

for Restricted Boltzmann Machines, 314-317
GPUs, 272, 367
layer count, parameter count, and memory, 246-251
learning rates, 258-263
loss functions, 256-258
matching input data to architecture, 240-242
network statistics tool, 284-291
optimization methods, 263-265
overfitting, 283
parallelization, 265-271, 372
regularization, 275-280
relating model goal and output layers, 242-245
for Spark, 367-370
sparsity, 263
weight initialization strategies, 251
Turing Complete networks, 143
typographical conventions, xvi

U

underfitting, 26
uneven time-series, 147
unit variance, 331
unsupervised pretrained networks (UPNs)
 architectures discussed, 118
 benefits and drawbacks of pretraining, 106
 Deep Belief Networks, 118
 Generative Adversarial Networks, 121-125
updates-to-parameters ratio, 259

V

vanishing gradient problem, 149, 311
variances, 330
variational autoencoders (VAEs), 114, 124, 214-220
Vector Space Model (VSM), 347, 353
vectorization
 additional resources on, 173
 challenges of, 321
 columnar raw data attributes, 325
 considerations for, 325
 data types requiring, 324
 dealing with missing values, 329
 definition of term, 9, 322

feature engineering, 327-328
of graph structures, 354
handcrafted vs. algorithmic approaches to, 324
of image data, 336-339
kernel hashing, 353
n-gram vectorization, 353
in ND4J, 457-458
normalization methods, 329-334
of text data, 347-354
phases of, 323
process of, 327
purpose of, 11
of sequential data, 340-346
using DataVec, 170, 334, 463-474
VGGNet, 142
video data, 240, 337
visual fields, 126

W

webapp-rl4j dashboard, 435
weight initialization strategies, 251, 287, 309
weighted loss functions, 282
whitening, 333
word embeddings, 221-226
Word2Vec
 additional uses for, 226
 alternatives to, 227
 applications for, 221, 235
 example, 225
 Java code listing, 224
 model and algorithm, 221
 modeling context, 222
 similar meaning and semantic relationships, 222
vector arithmetic and word embeddings, 223
vs. Vector Space Model, 353

Y

YARN, 364-366

Z

zero mean, 331
zero-padding, 139, 299, 312-314
ZF Net, 142

About the Authors

Josh Patterson currently is the head of field engineering at Skymind. Josh previously ran a consultancy in the big data/machine learning/deep learning space. Previously, he worked as a principal solutions architect at Cloudera and as a machine learning/distributed systems engineer at the Tennessee Valley Authority, where he brought Hadoop into the smart grid with the openPDC project. Josh has a Master's in computer science from the University of Tennessee at Chattanooga where he published research on mesh networks (tinyOS) and social insect optimization algorithms. Josh has more than 17 years in software development and is very active in the open source space, contributing to projects such as DL4J, Apache Mahout, Metronome, IterativeReduce, openPDC, and JMotif.

Adam Gibson is a deep learning specialist based in San Francisco. He works with *Fortune 500* companies, hedge funds, PR firms, and startup accelerators to create their machine learning projects. Adam has a strong track record helping companies handle and interpret big realtime data. He has been a computer nerd since the age of 13, and actively contributes to the open source community through <http://deeplearning4j.org>.

Colophon

The animal on the cover of *Deep Learning* is the oarfish (*Regalecus glesne*), a large lampriform (ray-finned) fish native to temperate and tropical oceans. They have a long and slender body with spiny dorsal fins running down their back. Oarfish can grow up to 11 meters in length, making them the largest bony fish in the world.

Oarfish are solitary animals and are rarely seen by humans. They spend much of their time in the mesopelagic zone (200 to 1,000 meters deep) and only go to the surface when they are sick or injured. Oarfish are carnivores that feed primarily on zooplankton as well as small fish, jellyfish, and squid.

The meat of the oarfish has a gelatinous consistency, so it is not targeted by commercial fishermen. Humans generally encounter the species only when dead or dying oarfish wash up on shore. Because of their size and shape, it is thought that oarfish may be the basis for sea serpent legends. Although their total population is unknown, there are no known environmental threats to oarfish.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *Braukhaus Lexicon*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.