Deep Learning

A Practitioner's Approach

Josh Patterson and Adam Gibson

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Deep Learning
by Adam Gibson and Josh Patterson

Copyright © 2016 Adam Gibson and Josh Patterson. 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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Tim McGovern
Production Editor: TO COME
Copyeditor: TO COME
Proofreader: TO COME
Indexer: TO COME
Interior Designer: David Futato
Cover Designer: TO COME

December 2016: First Edition

Revision History for the First Edition
yyyy-mm-dd: 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

[LSI]

Table of Contents

Preface ..... **xi**

1. A Review of Machine Learning **21**
The Learning Machines 21
How Can Machines Learn? 22
Biological Inspiration 25
What is Deep Learning? 26
Going Down the Rabbit Hole 27
Framing the Questions 29
The Math behind Machine Learning: Linear Algebra 29
Scalars 29
Vectors 30
Matrices 30
Tensors 31
Hyperplanes 31
Relevant Mathematical Operations 32
Converting Data Into Vectors 32
Solving Systems of Equations 34
The Math Behind Machine Learning: Statistics 36
Probability 37
Conditional Probabilities 39
Posterior Probability 40
Distributions 40
Samples vs Population 43
Resampling Methods 43
Selection Bias 43
Likelihood 44
How Does Machine Learning Work? 44

Regression 44
Classification 47
Clustering 48
Underfitting and Overfitting 48
Optimization 49
Convex Optimization 50
Gradient Descent 51
Stochastic Gradient Descent 54
Quasi-Newton Optimization Methods 55
Generative vs Discriminative Models 55
Logistic Regression 56
The Logistic Function 57
Understanding Logistic Regression Output 57
Evaluating Models 58
The Confusion Matrix 58
Building an Understanding of Machine Learning 62

2. Foundations of Neural Networks and Deep Learning. 65
Neural Networks 65
The Biological Neuron 67
The Perceptron 70
Multi-Layer Feed-Forward Networks 75
Training Neural Networks 81
Backpropagation Learning 82
Activation Functions 91
Linear 91
Sigmoid 92
Tanh 93
Hard Tanh 94
Softmax 94
Rectified Linear 95
Loss Functions 98
Loss Function Notation 99
Loss Functions for Regression 100
Loss Functions for Classification 102
Loss Functions for Reconstruction 104
Hyperparameters 105
Learning Rate 105
Regularization 106
Momentum 106
Sparsity 107

3. Fundamentals of Deep Networks. 109
Defining Deep Learning 109
What is Deep Learning? 109
Organization of This Chapter 120
Common Architectural Principals of Deep Networks 120
Parameters 121
Layers 122
Activation Functions 122
Loss Functions 124
Optimization Algorithms 125
Hyperparameters 128
Summary 133
Building Blocks of Deep Networks 133
Restricted Boltzmann Machines 135
Autoencoders 143
Variational Autoencoders 145

4. Major Architectures of Deep Networks. 149
Unsupervised Pre-Trained Networks 150
Deep Belief Networks 150
Generative Adversarial Networks 153
Convolutional Neural Networks 157
Biological Inspiration 159
Intuition 159
Convolutional Network Architecture Overview 161
Input Layers 163
Convolutional Layers 163
Pooling Layers 174
Fully-Connected Layers 174
Other Applications of Convolutional Networks 175
Convolutional Network Architectures of Note 175
Summary 176
Recurrent Neural Networks 176
Modeling the Time Dimension 177
3D Volumetric Input 180
Why Not Markov Models? 182
General Recurrent Network Architecture 183
Long Short-Term Memory (LSTM) Networks 184
Domain Specific Applications and Blended Networks 193
Recursive Neural Networks 194
Network Architecture 195
Varieties of Recursive Neural Networks 195

Applications of Recursive Neural Networks 195
Summary and Discussion 196
Will Deep Learning Make Other Algorithms Obsolete? 196
Different Problems Have Different Best Methods 196
When Do I Need Deep Learning? 197

5. Building Deep Networks. 199
Matching Deep Networks to the Right Problem 199
Columnar Data and Multi-Layer Perceptrons 200
Images and Convolutional Neural Networks 200
Timeseries Sequences and Recurrent Neural Networks 201
Using Hybrid Networks 203
The DL4J Suite of Tools 203
Vectorization and DataVec 204
Runtimes and ND4J 204
Basic Concepts of the DL4J API 206
Loading and Saving Models 206
Getting Input For the Model 207
Setting Up Model Architecture 207
Training and Evaluation 208
Modeling CSV Data with Multi-Layer Perceptron Networks 209
Setting Up Input Data 212
Determining Network Architecture 212
Training the Model 215
Evaluating the Model 215
Modeling Hand-Written Images with Convolutional Neural Networks 216
Java Code Listing for LeNet Convolutional Network 217
Loading and Vectorizing the Input Images 219
Network Architecture for LeNet in DL4J 220
Training the Convolutional Network 223
Modeling Sequence Data with Recurrent Neural Networks 224
Generating Shakespeare with LSTMs 224
Classifying Sensor Timeseries Sequences with LSTMs 232
Using Autoencoders for Anomaly Detection 238
Java Code Listing for Autoencoder Example 239
Setting Up Input Data 242
Autoencoder Network Architecture and Training 243
Evaluating the Model 244
Using Variational Autoencoders to Reconstruct MNIST Digits 247
Code Listing to Reconstruct MNIST Digits 247
Examining the VAE Model 250
Applications of Deep Learning in Natural Language Processing 254

Learning Word Embeddings with Word2Vec 255
Distributed Representations of Sentences with Paragraph Vectors 261
Using Paragraph Vectors for Document Classification 265

6. Tuning Deep Networks. 271
Basic Concepts in Tuning Deep Networks 271
An Intuition for Building Deep Networks 272
Building the Intuition as a Step-by-Step Process 273
Matching Input Data and Network Architectures 274
Summary 275
Relating Model Goal and Output Layers 276
Regression Model Output Layer 276
Classification Model Output Layer 277
Working with Layer Count, Parameter Count, and Memory 280
Feed-Forward Multi-Layer Networks 280
Controlling Layer and Parameter Counts 281
Estimating Network Memory Requirements 284
Weight Initialization Strategies 285
Using Activation Functions 287
Summary Table for Activation Functions 289
Applying Loss Functions 290
Understanding Learning Rates 292
Using the Ratio of Parameters to Updates 293
Specific Recommendations for Learning Rates 294
How Sparsity Affects Learning 297
Applying Methods of Optimization 297
Stochastic Gradient Descent Best Practices 299
Leveraging Parallelization and GPUs for Faster Training 300
Online Learning and Parallel Iterative Algorithms 300
Parallelizing Stochastic Gradient Descent in DL4J 303
GPUs 306
Controlling Epochs and Mini-Batch Size 307
Understanding Mini-Batch Size Tradeoffs 308
How to Use Regularization 309
Priors as Regularizers 310
Max-Norm Regularization 311
Dropout 311
Other Topics in Regularization 313
Working with Class Imbalance 314
Methods for Sampling Classes 316
Weighted Loss Functions 317
Dealing with Overfitting 318

Using Network Statistics from the Tuning UI 319
Detecting Poor Weight Initialization 322
Detecting Non-Shuffled Data 323
Detecting Issues with Regularization 325

7. Tuning Specific Deep Network Architectures. 329
Convolutional Neural Networks 329
Common Convolutional Architectural Patterns 330
Configuring Convolutional Layers 334
Configuring Pooling Layers 339
Transfer Learning 340
Recurrent Neural Networks 342
Network Input Data and Input Layers 343
Output Layers and RnnOutputLayer 344
Training the Network 345
Debugging Common Issues with LSTMs 347
Padding and Masking 348
Evaluation and Scoring With Masking 349
Variants of Recurrent Network Architectures 350
Restricted Boltzmann Machines 350
Hidden Units and Modeling Available Information 351
Leveraging Different Units 352
Using Regularization with RBMs 353
Deep Belief Networks 354
Using Momentum 354
Using Regularization 355
Determining Hidden Unit Count 356

8. Vectorization. 359
Introduction to Vectorization in Machine Learning 359
Why do We Need to Vectorize Data? 360
Strategies For Dealing with Columnar Raw Data Attributes 363
Feature Engineering and Normalization Techniques 365
Leveraging DataVec for ETL and Vectorization 373
Vectorizing Image Data 374
Image Data Representation in DL4J 375
Image Data and Vector Normalization with DataVec 377
Working with Sequential Data in Vectorization 378
Major Variations of Sequential Data Sources 379
Vectorizing Sequential Data with DataVec 380
Working with Text in Vectorization 386
Bag of Words 387

Term Frequency Inverse Document Frequency (TF-IDF) 388
Word2vec and Vector Space Model Comparison 393
Working with Graphs 394

9. Using Deep Learning and DL4J on Spark. 397
Introduction to Using DL4J with Spark and Hadoop 397
Operating Spark from the Command Line 400
Configuring and Tuning Spark Execution 403
Running Spark on Mesos 403
Running Spark on YARN 405
General Spark Tuning Guide 407
Tuning DL4J Jobs on Spark 411
Setting Up a Maven POM for Spark and DL4J 413
A Pom.xml File Dependency Template 415
Setting Up a POM File for CDH 5.X 418
Setting Up a POM file for HDP 2.4 419
Troubleshooting Spark and Hadoop 420
Common Issues with ND4J 420
DL4J Parallel Execution on Spark 421
A Minimal Spark Training Example 423
DL4J API Best Practices for Spark 426
Multi-Layer Perceptron Spark Example 427
Setting Up MLP Network Architecture for Spark 430
Distributed Training and Model Evaluation 431
Building and Executing a DL4J Spark Job 433
Generating Shakespeare Text with Spark and LSTMs 433
Setting Up the LSTM Network Architecture 436
Training, Tracking Progress, and Understanding Results 436
Modeling MNIST with a Convolutional Neural Network on Spark 437
Configuring the Spark Job and Loading MNIST Data 440
Setting Up the LeNet CNN Architecture and Training 440

A. What is Artificial Intelligence?. 443

B. RL4J and Reinforcement Learning. 455

C. Numbers Everyone Should Know. 481

D. Neural Networks and Backpropagation: A Mathematical Approach. 483

E. Using the ND4J API. 489

F. Using DataVec. 503

G. Working with DL4J From Source. 515

H. Setting Up DL4J Projects. 517

I. Setting Up GPUs for DL4J Projects. 521

J. Troubleshooting DL4J Installations. 525

Preface

What's in This Book?

The first four chapters of this book are focused on enough theory and fundamentals to give the practitioner a working foundation for the rest of the book. The last five chapters then work from these concepts to lead the reader 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

The authors would like to note that we use the names DL4J and Deeplearning4j interchangeable in this book. Both terms refer to the suite of tools in the Deeplearning4j library.

We architected the book in this manner as we felt there was a need for a book covering "enough theory" while being practical enough to build production-class Deep Learning workflows. The authors 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 us 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 for the practitioner. Chapter 3 further builds on the first two chapters in bringing the reader up to speed on how Deep Networks evolved from the fundamentals of neural networks. Chapter 4 then introduces us to the 4 major architectures of Deep Networks and gives us the foundation for the rest of the book.

In Chapter 5 we take the reader through a number of java code examples using the techniques from the first half of the book. Chapters 6 and 7 take the reader through the fundamentals of tuning general neural network 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 with a review on how to use DL4J natively on Spark and Hadoop and illustrates 3 real examples users can run on their 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:

- What is 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 gets used in many different ways. The world of data science and artificial intelligence 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 wide-spread entanglement has historical parallels to when the World Wide Web (90's) 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 the authors wanted 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 out with a series of fundamentals to take the reader on a full journey through Deep Learning.

We felt too many books left out core topics that the enterprise practitioner often needed for a quick review. Based on our machine learning experiences in the field we decided to lead off with the materials we often see our entry-level practitioners needing to brush up on to better support their Deep Learning projects.

Some readers may want to skip chapters 1 and 2 and get right to the Deep Learning fundamentals. However, other readers will appreciate having the material up front so they can have a smooth glide path into the harder topics in Deep Learning that build on these principles. Below we suggest some reading strategies for different backgrounds.

The Enterprise Machine Learning Practitioner

We split this group into two sub-groups:

- Practicing Data Scientist
- Java Engineer

The Practicing Data Scientist

This reader typically builds models already and is fluent in the realm of data science. This reader is probably ready to skip chapter 1 and will want to lightly skim chapter 2. We suggest moving on to chapter 3 as they'll probably be ready to jump into fundamentals of Deep Networks.

The Java Engineer

This reader is typically tasked with integrating machine learning code with production systems. Starting with chapter 1 will be interesting for them because it will give them a better understanding of the vernacular of data science. The ND4J Appendix should also be of keen interest to the Java engineer as 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. We suggest the executive reader

start with a quick skim of chapter 1 to re-acclimate to some terminology. The executive may want to skip the chapters that are heavy on APIs and examples, however.

The Academic

The academic reader will typically want to skip chapters 1 and 2 as 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 this reader as this information is based in research and transcends any specific Deep Learning implementation. The coverage of ND4J will also be of interest to the academic who prefers to do high-performance linear algebra on the 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 INSERT LINK.

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 2016 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. The reader can see the full import listings in the actual code repository. The API information for DL4J, ND4J, DataVec, and more are available on the websites:

http://deeplearning4j.org/documentation

All code examples are located at:

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

For more resources on the DL4J family of tools check out the website:

http://deeplearning4j.org/

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, 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, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

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.

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 (*https://twitter.com/jpatanooga*)

Follow Adam Gibson on Twitter: @agibsonccc (*https://twitter.com/agibsonccc*)

Acknowledgements

Josh's Acknowledgements

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 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 in not only reviewing the book, but also contributing content in the appendix chapters. 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 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 right answer on how to communicate a complex topic and James Long was my sounding board for arguing the case from multiple sides. While 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 NLP and word2vec examples.

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

I would like to thank the people who contributed Appendix chapters: Alex Black (Backprop, DataVec), Vyacheslav “Raver” Kokorin (GPUs), Susan Eraly (GPUs), and Ruben Fiszel (Reinforcement Learning). Other reviewers of the book at various stages include: Grant Ingersol, Dean Wampler, Robert Chong, Ted Malaska, Ryan Geno, Lars George, Suneel Marthi, Francois Garillot, and Don Brown. Any errors the reader may 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 3 chapters. I felt like he gave us the space to “get this right”, and we appreciate it.

Other folks I’d like to note that had an impact on my career leading up to this book: 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), 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’s Acknowledgements

Section content goes here

CHAPTER 1

A Review of Machine Learning

To condense fact from the vapor of nuance

--- Neal Stephenson, Snow Crash

The Learning Machines

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

Neural networks are one type of model for machine learning; they have been around for at least 50 years. The fundamental unit of a neural network is a "node," loosely based on the biological neuron in the mammalian brain. The connections between neurons, too, are 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-1980's and early 1990's many important architectural advancements were made in neural networks. However, the amount of time and data needed to get good results slowed adoption and interest cooled. In the early 2000's computational power exploded and the industry saw a "Cambrian explosion" of computational techniques that were not possible before. 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 leading into 2017 where we see deep learning mentioned in every corner of machine learning today.

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

- 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 the reader 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 the reader 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 "getting knowledge of 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 be can take many forms including:

- decision trees
- linear regression models
- 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 vs Data Mining

Data Mining has been around for many decades and like many terms in machine learning gets 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. A simple way to think of data mining is:

- 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 at IBM and Stanford[1], defined machine learning as a

> field 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, then recognizing and reinforcing successful patterns, underpins machine learning and artificial intelligence to this day.

The concept of machines that could learn to achieve goals on their own has captivated us for decades. This was perhaps best expressed by the modern grandfathers of Artificial Intelligence, Stuart Russell[2] and Peter Norvig[3], when they wrote in the book "Artificial Intelligence: A Modern Approach"[4]:

> 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, we illustrate our conception of the relationship between Artificial Intelligence, Machine Learning, and Deep Learning in the diagram below.

1 http://infolab.stanford.edu/pub/voy/museum/samuel.html

2 https://www2.eecs.berkeley.edu/Faculty/Homepages/russell.html

3 http://norvig.com/

4 http://aima.cs.berkeley.edu/

Figure 1-1. The Relationship Between Artificial Intelligence and Deep Learning

The field of Artificial Intelligence is broad and has been around for a long time. Deep Learning is a subset of the field of machine learning which is a sub-field of Artificial Intelligence. 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 over 500 trillion connections between neurons in the human brain. Even the largest artificial neural networks today come no where close to 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 (several orders of magnitude) more complex 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, the neurons of a neural network can be trained to pass along only useful signals. As we move through this chap-

ter 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:

- ants
- termites[5]
- bees
- genetic algorithms

Ant colonies[6], for instance, were shown by researchers to be a powerful decentralized computer where no single ant is a central point of failure. Ants constantly switch 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 near optimal number of workers on each task based on the relative need with no one directly coordinating the work.

What is Deep Learning?

Deep learning has been a challenge to define for many as 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 has been around since the 1980's. 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. Some of the facets in this evolution of neural networks include:

- 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 large number of parameters and layers in one of four fundamental network architectures:

5 http://scholar.utc.edu/theses/184/ and http://www.aaai.org/ocs/index.php/IAAI/IAAI09/paper/viewFile/235/1030

6 https://mitpress.mit.edu/books/ant-colony-optimization

- Unsupervised Pre-Trained Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

There are some variations of the above architectures—a hybrid convolutional and recurrent neural network, for example, as well. For the purpose of this book we'll consider the above four 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 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 starting 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 non-computer scientist. One example of this would be the art generation demos where 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 own style.

Figure 1-2. Stylized Images by Gatys et. al., 2015[7]

This begins to enter into many philosophical discussions such as "can machines be creative?" and then "what is creativity?". We'll leave that discussion for the reader 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 won "Jeopardy" or beat a Go Grand Master.

Can machines be intelligent and take on human level intelligence? What is artificial intelligence and how powerful could it become? These questions have yet to be answered and will not be completely answered in this book. We simply seek to illustrate to the reader some of the shards of machine intelligence we can imbue our environment with today through the practice of Deep Learning.

For Extended Discussion on Artificial Intelligence

The reader should take a look at the Appendix at the back of the book on "What is Artificial Intelligence?"

7 Gatys (et. al, 2015) "A Neural Algorithm of Artistic Style" http://arxiv.org/pdf/1508.06576v1.pdf

Framing the Questions

Understanding the basics in applying machine learning is best framed by asking the correct questions to start with. We need to define:

- What is the input data we want to extract information (model) from?
- 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 3 questions we can setup 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. We'll then 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.

For More on Linear Algebra

A great primer on linear algebra is James E. Gentle's "Matrix Algebra: Theory, Computations, and Applications in Statistics"[8]

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".

8 http://mason.gmu.edu/~jgentle/books/matbk/index.htm

Vectors

For our use we define a vector as:

> *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 can also 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:

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

Or

$$x = \left[x_1, x_2, x_3, \ldots, x_n\right]$$

There are many different ways to handle the vectorization and many pre-processing steps can be applied giving us different grades of effectiveness on the output models. We'll cover more on the topic of converting raw data into vectors later on 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 where we have rows and columns.

Figure 1-3. A 3x3 Matrix

If our matrix is said to be a n x m matrix then it has n rows and m columns. In Figure 1-3 above we see a 3 x 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.

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 are considered tensors.

Hyperplanes

Another linear algebra object the reader should be aware of is the hyperplane. The hyperplane is a subspace of one dimension less than its ambient space in the field of geometry. In a 3-dimensional space the hyperplanes would be of 2-dimensions. In a single dimension (a number line) hyperplanes end up as 1-dimensional dots.

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 we'll see further on in this chapter.

Relevant Mathematical Operations

In this section we briefly review common linear algebra operations the practitioner should note.

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 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 cosine 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 2 source vectors.

Outter 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 touch many types of data (text, timeseries, audio, image, video)

So why can't we just feed raw data to our learning algorithm and just 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 to have 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[9]:

```
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 of the form of a matrix but we can convert the data into intermediate representations (e.g. "svmlight" file format, as seen below). We want our machine learning algorithm's input data to look more like the serialized sparse vector format svmlight below:

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468 4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0.5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0.5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149 4:0.5217391304347826
0.0 2:0.7500000000000002
```

This format can quickly be read into a matrix and a column vector for the labels (the first number in each row above). The rest of the indexed numbers in the row are

9 http://archive.ics.uci.edu/ml/datasets/Iris

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 various formats for vectors and the process of vectorization in more detail in chapter 8.

A very common question is "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 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 3 rows of serialized sparse output from the above diagram and place the values in its linear algebra form it looks like:

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

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

What is a Feature?

A feature in machine learning is any column value in the input matrix A that we're using as in 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 had 4 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 to 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 above serialized sparse vectors would be:

Labels

1.0

2.0

2.0

The coefficients mentioned above become the x column vector (also called "parameter vector") shown in the diagram below:

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

$x = A^{-1}b$

Its 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" where we know algorithmically there are a fixed number of computations. The other approach is a class of methods known as "iterative methods" where through a series of approximations and a set of termination conditions we can derive the parameter vector x. The direct class of methods is particular 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 particular effective when our data doesn't fit into main memory on 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" as we'll discuss later in this chapter. Other techniques in this space are "Conjugate Gradient Methods" and "Alternating Least Squares" (discussed further on in chapter 3 of this book). Iterative methods have also been shown to be effective in scale-out methods where 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 detailed 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 the book we show code examples of converting the raw input data into the input matrix A, giving the reader the "how". The mechanics of how we vectorize our data also impacts the results of the learning process. As we'll see later in the book, how we handle data in the pre-process 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

- probabilities
- distributions
- likelihood

There are also some other basic relationships we'd like to highlight in descriptive statistics and inferential statistics. Descriptive statistics includes:

- histograms
- boxplots
- scatter plots
- mean
- standard deviation
- correlation coefficient

This contrasts with how inferential statistics are concerned with techniques for generalizing from a sample to a population. Examples of inferential include:

- p-values
- credibility intervals

The relationship between probability and inferential statistics is that:

- probability reasons from the population to the sample ("deductive reasoning")
- inferential statistics reasons 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 uncertaintity associated with taking a sample from a given population.

With regards to 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 to direct the reader towards relevant topics that can be investigated in further depth from other resources. With that disclaimer out of the way, let's start off 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 yet we can also express it as a percentage between 0% and 100%. We will not see valid probabilities lower than 0% and greater than 100%. An example would be a probability of 0.35 expressed as 35% (e.g. 0.35 x 100 == 35%).

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 as 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 as the total probability of the sample space must always add up to 1. We express the probability of an event as:

P(E) = 0.5

And we read this as:

> the probability of an event E is 0.5

Probability vs Odds, Explained

Many times practitioners new to statistics or machine learning will conflate the meaning of probability and odds, hence our quick side note here.

The probability of an event E is defined as:

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

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

4/52 = 0.077

Conversely, odds are defined as:

(Chances for E) : (Chances Against E)

where now our card examples becomes the "odds of drawing an ace":

4 : (52 - 4) = 1/12 = 0.0833333...

The primary difference here is the choice of denominator (Total Chances vs 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 the reader could check out O'Reilly's "Statistics in a Nutshell: A Desktop Quick Reference" by Boslaugh and Watters[10].

Further Defining Probability: Bayesian vs 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

10 http://shop.oreilly.com/product/0636920023074.do

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 we're interested in a probability for

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(ICU Death | Poor Heart Rate) > P(ICU Death | 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 as 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 where 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 we're predicting E for. 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 test's 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:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

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 in softmax activation functions, explained later in this chapter, when the softmax activation functions converts the raw input value into 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 reasonable close to the distribution then we can make assumptions based on the theoretical distribution in how we operate with the data.

Distributions can be classified as continuous (data can be any value within the range) or discrete. A discrete distribution has data that can assume only certain values. An example of a continuous distribution would be the normal distribution. An example of a discrete distribution would be the binomial distribution.

The normal distribution allows us to assume sampling distributions of statistics (e.g. "sample mean") are normally distributed under specified conditions. The normal distribution, or "gaussian distribution", was named after the 18th-century mathematician

and physicist Karl Gauss. The 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:

- binomial distribution
- inverse gaussian distribution
- log normal distribution

The distribution of the training data in machine learning is important to understand to better know 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 if 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 form 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 where a high-frequency population is followed by a low-frequency population which gradually decreases in an asymptotic fashion. These distributions were discovered by Benoit Mandelbrot in the 1950's 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 where 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") many times will form power laws. From this perspective we can consider them to be long-tailed distributions.

We see these class of long-tailed distributions manifested when we see:

- Earthquake damage
  - damage gets 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 get tuned around the mean
- Predicting fatality post-ICU visit
  - we can have events far outside the scope of what happens inside the ICU visit affecting 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. In the case that the more interesting events occur out in the tail of the distribution and we don't have

this represented in the training sample data then our model may perform unpredictably. This effect can be enhanced in non-linear 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 5x the standard deviation. We have to be mindful to get a decent representation of events in our training data to prevent overfitting the training data. We'll look at ways to do this further on in our material on overfitting and then in chapter 4 on tuning

Samples vs 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 a number of splits but researchers have found 10 splits to work well in practice. It is also 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 have a reasonable probability of happening but still may not. There may also 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 around 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 give 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 above 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 stochastic gradient descent 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 with the equation:

y = a + Bx

where

a y-intercept

B input features

x parameter vector

This equation expands to:

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

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 costs are the dependent variable and the miles of the commute are the independent variable. It's reasonable to keep track of these two quantities and then define a function

cost = f (distance)

such that we'll reasonable 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.

Other examples of linear regression modeling include:

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

Visualizing Linear Regression

Visually linear regression can be represented as finding a line that comes close to as many points as possible in a scatterplot of data as we see below in the diagram.

Figure 1-6. Linear Regression Plotted

Fitting is defining an 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 equation above Ax = b where A is the features (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. 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. "stochastic gradient descent") we can find a set of x parameters such that we minimize the error across all of the predictions vs the true outcomes.

Using stochastic gradient descent, as we discussed above, we'd have 3 components to solve for our parameter vector x:

1. A hypothesis about the data
   - *the inner product of the parameter vector x and the input features (as displayed above)*
2. A cost function

- *squared error (prediction - actual) of prediction*

3. An update function
   - *take 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 scatterplot. Ironically, absolute fit is usually a very poor outcome, because it means your model has trained too perfectly on our 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 "*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 certaintity. In this case we need to determine a threshold (typically 0.5) where we delineate between the two classes. We often hear of these classes referred to in literature as the positive (e.g. 1.0) and then negative (e.g. 0.0) classifications. We'll talk more about this in a future section on evaluating model performance.

Examples of binary classification include:

- classifying is 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 where 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 in the chapter as we talk about neural networks with multiple outputs vs 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 techniques that involves using a distance measure and iterative 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 above, optimization algorithms first attempt to solve the problem of under-fitting; that is, of taking a line that does not approximate the data well, and making it approximate the data better. A straight line cutting across a curving scatter plot would be a good example of under-fitting as we can see in the diagram below.

Figure 1-7. Under-fitting and Over-fitting in Machine Learning

If the line fits the data too well, we have the opposite problem called "over-fitting". Solving under-fitting 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 it may have a low error rate for the training data but it does not generalize well to the overall population of data we're interested in.

Another way of explaining over-fitting 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 have to fit the larger set equally well if it is to have any predictive power. We have to assume, therefore, that our sample is loosely representative of a larger set.

Optimization

The process described above of adjusting weights to produce more and more accurate guesses about the data is known as parameter optimization. We can think of this process like the scientific method. We 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; i.e. 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 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 (that is, 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 it corresponds to. 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 tells the hiker which direction is downhill, so 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 above in diagram-X. 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 give our optimization algorithm the smallest error across all training examples possible. The optimization algorithm is said to "converge" on the solution iteratively after it tries several different variations of the parameters.

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 sub-class 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, then 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.

We can also turn the idea of a cost function upside down as seen below in diagram-X:

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 make the given data most likely.

In a sense, maximum likelihood and minimum cost are two faces on 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 in what direction to take the next parameter step as we'll see in the gradient descent optimization algorithm.

Gradient Descent

In gradient descent we can imagine a landscape made of loss, whose hills represent a lot of error and whose valleys represent less error. We choose one point on that landscape at which to place your initial weight. We then may select the initial weight based on domain knowledge (if we're training a network to classify a flower species you may

know that 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 senses the actual slope of the hills with regard to each weight; i.e. 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 we see in the diagram below.

Figure 1-9. Showing weight changes towards global minimum in Stochastic Gradient Descent

The derivative measures "rate of change" of a function. In convex optimization we're looking for the point where the derivative is equal to 0 for the function. This point is also know 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 where 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; i.e. 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. Since slopes are derived from two points, how exactly does one find the slope of one point on a curve? We find the derivative by calculating the slope of a line between two points on the curve separated by a small distance and then slowing decreasing that distance until it approaches zero. In calculus, this is the idea of 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.

Linear modeling can be thought of in terms of 3 components to solve for our parameter vector x:

1. A hypothesis about the data
   - e.g. "the equation we use to make a prediction in the model"
2. A cost function
   - also called a loss function, e.g. "sum of squared errors"
3. 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) in a way 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 closer 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 non-linear problems we don't always get such a clean loss curve. The problem with these other non-linear hypothetical landscapes is that there may 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" while 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 later on in this book when we write about 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 hard 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 stochastic gradient descent 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. Stochastic gradient descent is an approximation of "full batch" gradient descent.

Mini-Batch Training and Stochastic Gradient Descent

Another variant of stochastic gradient descent 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 stochastic gradient descent 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 as the gradient is computed at each step 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") then 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 will be discussed more in later chapters of the book.

The Jacobian and the Hessian

The Jacobian is a m x n matrix containing the 1st order partial derivatives of vectors with respect to vectors.

The Hessian is the square matrix of 2nd 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 as 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'll cover more in chapter 2.

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

Generative vs Discriminative Models

We can generate different types of output from a model depending on what type of model we setup. 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 may 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 give a certain input. Generative models are typically setup 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 then for multiple labels in the form of multi-nomial logistic regression. Logistic regression is a regression model (technically) where 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 features (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 vs the true outcomes.

Again we'll use stochastic gradient descent to setup this optimization problem and we have 3 components to solve for our parameter vector x:

1. A hypothesis about the data
   - $f(x) = \frac{1}{1 + e^{-\theta x}}$
2. A Cost Function
   - "max likelihood estimation"
3. An Update Function
   - derivative of the cost function

In this case, the input is made of independent variables (e.g. the input columns or "features") while 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 if an outcome is likely or not. Let's take a closer look at the logistic function.

The Logistic Function

In logistic regression we define the logistic function ("hypothesis") as:

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

This function is useful in logistic regression as it takes any input on the range of negative to positive infinity and maps it to output on the range of 0.0 to 1.0. This allows us to interpret the output value as a probability. In the diagram below we can see 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 this chapter when we look at the sigmoid activation function.

Understanding Logistic Regression Output

The logistic function is often denoted with the Greek letter sigma, or , since 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, then the more x increases, the closer we come to 1/1, because e to the power of an infinitely large neg-

ative number approaches zero; in contrast, the more x decreases below zero, the more the expression $\left(1 + e^{-\theta x}\right)$ grows, shrinking the entire quotient. Since $\left(1 + e^{-\theta x}\right)$ is in the denominator, the larger it gets, 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, then we could paraphrase that by saying y has a 60 percent of being 1, or the email has a 60 chance of being spam, given the input. If we define machine learning as a method to infer unknown outputs from known inputs, then 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 and then in other times its important that the model gets a certain type of prediction correct more often than the others. We'll cover topics like bad positives, harmless negatives, unbalanced classes, and unequal costs for predictions in this section. Let's take a look at the basic tool of evaluating models called the "confusion matrix".

The Confusion Matrix

The confusion matrix (also called "a table of confusion") as we see below in Figure 1-11 is a table of rows and columns that represent 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 right answer at the right 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:

- 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 also known as "type II error". By tracking these metrics we can get a more detailed analysis on the performance of the model beyond the basic percent of guesses that were correct. We can calculate different evaluations of the model based on combinations of the above four counts in the confusion matrix as displayed below:

```
Accuracy:  0.94
Precision: 0.8662
```

```
Recall:    0.8955
F1 Score:  0.8806
```

In the above example we can see four different common measures in the evaluation of machine learning models. We'll cover each of them below but for now let's start with the basics of evaluating model sensitivity vs model specificity.

Sensitivity vs 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 is also called sensitivity or recall and an example would be classifying a patient as having a condition who was actually sick. Sensitivity quantifies how well the model avoids false negatives.

Sensitivity = TP / (TP + FN)

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

Specificity = TN / (TN + FP)

Many times we need to evaluate the trade-off between sensitivity and specificity. An example would to have a model which detects a serious sickness in patients more frequently because of the high cost to missing out on 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 them so our model would be considered to have a high sensitivity to this situation and its implications. In a perfect world our model is 100% sensitive (e.g. all sick are detected) and 100% specific (e.g. no one who is not sick is classified as sick).

Accuracy

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

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

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

Precision

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

Precision = TP / (TP + FP)

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

Recall

This is the same thing as sensitivity described above 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 below.

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

We see scores for F1 scores 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. Typical domains we see this in are:

- web click prediction
- ICU mortality prediction
- fraud detection

In these contexts an overall "percent correct" score may be misleading to the overall value, in practical terms, of the model. An example of this would be the PhysioNet Challenge dataset from 2012[11].

The goal of the challenge is to "predict in-hospital *mortality* with the greatest accuracy using a binary classifier." The difficulty and challenge in modeling this dataset is that predicting a patient will live is the easy part because the bulk of examples in the dataset have outcomes where the patient does live. Predicting death accurately in this scenario is the goal and 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

Score = MIN(Precision, Recall)

This was setup such that it kept the contestants focused on not just predicting 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'll illustrate practical ways to deal with class imbalance. We'll 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

11 https://physionet.org/challenge/2012/

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 where 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.

From our previous sections we wrote about 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 the x (parameter vector).

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

- 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 multi-layer 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 we see below in #multi-layer-neural-ne:

Figure 2-1. Multi-Layer Neural Network Topology

A feed-forward multi-layer 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 (discussed above in 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 get stuck in local minima but in practice generally performs well.

Historically, backpropagation has been considered "slow" but recent advances in computational power through parallelism and GPUs have driven a renewed interest in neural networks.

Much hubris and many words have been transmitted across the internet and in literature about the connection of neural networks to the human mind. Let's separate some

signal from the vast noise and start with the biological inspiration of artificial neural networks.

The Mechanistic View of the Mind

Rather than constructing a rigid tree that requires all inputs be one thing or another, we can construct a model that reflects a world sending us partial, ambiguous information, from which we 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 20th 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 understand of the perceptron we'll then see how it evolved into the more generalized artificial neuron which supports today's modern feed-forward multi-layer perceptrons. The culmination of chapter 2 will give the modern neural network practitioner the fundamentals to delve further into more exotic deep network architectures.

The Biological Neuron

The biological neuron is a nerve cell that provides the fundamental functional unit for the nervous systems of all animals. Neurons exist to communicate with each other, 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. The Biological Neuron

In Figure 2-2 above we can see how the major parts of the nerve cell include:

- 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 may branch hundreds of times, however. Dendrites are thin structures that arise from the main call body. Axons are nerve fiber with a special cellular extension that comes from the cell body.

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 dendrite. The exceptions for this case is where a neuron may 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 signal 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 fiber extending from the main cell body (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 is the long-term changes in strength of connections in response to input stimulus. Neurons have also 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 Systems

Later in this book we'll take a look at a Deep Network called a "Convolutional Neural Network". Convolutional Neural Networks image representation at different layers have shown similarities to how the brain processes visual information[1]. While this research is interesting it should not be taken to say that a Convolutional Neural Network gives us a full approximation of mammalian brain activity.

1 http://www.sciencedirect.com/science/article/pii/S1053811916305481

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 18th century the brain began to be recognized as the "seat of consciousness". By the late 19th 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 move on to 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 this 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 Threshold Logic Unit 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[2]. It was funded by the U.S. Office of Naval Research and was covered by the New York Times:

> the 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 artificial intelligence. 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 was later implemented in the Mark I Perceptron machine.

2 https://en.wikipedia.org/wiki/Frank_Rosenblatt

It should also be noted that McCulloch and Pitts introduced the basic concept of analyzing neural activity in 1943[3] 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 U.S. 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 explained below in Figure 2-3. In this diagram we can see how 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.

Figure 2-3. Single Layer Perceptron

3 http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf

We can model the decision boundary and the classification output in the heaviside step function equation below:

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

To produce the net input to the activation function (here "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 above as the input to the summation function. In the table below we give an explanation of how the summation function is performed and 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 ($\sum_{i=1}^{m} w_i x_i$)
m	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 "multi-layer perceptron".

As a basic linear classifier we consider the single layer perceptron to be the simplest form of the family of feedforward 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 a dataset that we can find a 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 0.0's at the beginning of training. The perceptron learning algorithm takes each input record as we see above 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 and 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 then no weight changes are made. If the classification is incorrect then 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 then the training algorithm will not terminate. An example of dataset that is not linearly separable would be 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 non-linear (e.g. datasets which are not linearly separable) problems was seen as a failure for the field of neural networks. The 1969 book "Perceptrons" by Minsky and Papert[4] illustrated the limitations of the sin-

4 https://en.wikipedia.org/wiki/Perceptrons_(book)

gle layer perceptron. However, what the general industry did not widely realize was that a multi-layer perceptron could indeed solve the XOR problem among many other non-linear problems.

AI Winter I: 1974-1980

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

Multi-Layer Feed-Forward Networks

The Multi-Layer Feed-Forward 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 multi-layer perceptrons later in this chapter. 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 multi-layer perceptron is similar to its predecessor, the perceptron, but adds flexibility in the type of activation layer we can use. Below in Figure 2-5 we see an updated diagram for the artificial neuron that is based on the perceptron:

Figure 2-5. Artificial Neuron for a Multi-Layer Perceptron

This diagram is similar to Figure 2-3 for the Single Layer Perceptron yet we notice a more generalized activation function. We'll evolve this diagram further in a detailed look at the artificial neuron below.

A Note About The Term "Neuron"

From this point throughout the remainder of the book when the writers use the term "neuron" we're referring to the artificial neuron based on the diagram above.

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 express more complex activation output.

Artificial Neuron Input . The artificial neuron 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 an activation.

Figure 2-6. Details of an Artificial Neuron in a Multi-Layer Perceptron Neural Network

We express the net input to a neuron as the weights on connections times activation incoming on connection as we can see in the above diagram. For the input layer we're just taking the feature at that specific index and the activation function is linear (it passes on the feature value). For hidden layers the input is the activation from other neurons. Mathematically we can express the net input (total weighted input) of the artificial neuron as:

$$input_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i$$

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

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

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

$$a_i = g(input_sum_i)$$

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

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

This activation value for neuron i is the output value passed on to the next layer through connections to other artificial neurons (multiplied by weights on 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:

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

This notation is slightly different than our 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 the reader to be able to recognize the variants.

If our activation function is the sigmoid function then we have:

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

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 you want to produce information from 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 this 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 multi-layer perceptron neural network. We'll 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 network.

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) non-linear 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 non-zero 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 a later section of this book 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 which the artificial neuron is based on, 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 in Comparisons

The authors would note (again) that the biological neuron is still more complex than the artificial variant. Research continues towards 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 multi-layer feed forward neural network. With multi-layer feed forward neural networks we have artificial neurons arranged into groups called layers. Building on the layer concept, we see the multi-layer neural network has:

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

The neurons in each layer (represented by the circles) are all fully connected to all neurons in all adjacent layers as displayed in Figure 2-7 below:

Figure 2-7. Fully-Connected Multi-Layer Feed-Forward Neural Network Topology

The neurons in each layer will all use the same type of activation function (most of the time). For the input layer the input is the raw vector input. The input to neurons of the other layers is the output (activation) of the previous layer's neurons. As data moves through the network in a feed-forward fashion it is influenced by the connec-

tion weights and the activation function type. Let's now take a look at the specifics of each layer type.

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 networks architectures the input layer may 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 model non-linear functions as we saw from the limitations on 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 on in this chapter.

Connections Between Layers. In a fully-connected feed-foward 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 will give the reader a more detailed look at the training mechanics of backpropagation, specifics of activation functions, and close 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. Input paired with a large weight will affect the network's interpretation of the data more than other 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 out 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 relating to sell price of a house). Neural networks learn these relationships blindly by making a guess based on the inputs and weights and then measuring how accurately the results were. The loss functions in optimization algorithms, such as stochastic gradient descent, reward the network for good guesses and penalize it for bad ones. Stochastic Gradient Descent moves the parameters of the network towards 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 called "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-foward neural networks. Let's setup a general intuition 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-1980's.

Algorithm Intuition

Backpropagation learning is the same general idea as 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 then we don't do anything. If the output does not match the label, then 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:

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 its easy because there is only one weight per input to influence the output value. With feed-forward multi-layer networks learning algorithms have a bigger challenge. There are many weights connecting each input to the output so it gets harder. Each weight contributes to more than one output so our learning algorithm to 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'll take a look at the mathematical notation that the reader will see in most literature on neural networks for backpropagation of feed-forward neural networks.

A Caution on Learning Algorithms in Multi-Layer Networks

Learning algorithms for multi-layer 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

Most of this book we won't throw a lot of math at the reader. However, for the topic of backpropagation and to better understand a core fundamental concept that much of the book is based on, the authors felt we should provide a section that illustrated these concepts down to the notation level.

A Note About Notation

This notation is similar to what the reader would see in a conference paper on machine learning or in a well-known machine learning text-book. Hopefully we can explain the notation in a way that the reader will feel comfortable with. Ideally this will serve as a jump off point for the reader 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 as we see in the diagram below.

Figure 2-8. Multi-Layer Perceptron Network Zoomed-In with Component Labels

In the diagram above we see our previous multi-layer 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. In the table below we have a set of notations that the reader can see from the diagram above:

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)
A_i	vector of activations 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
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
$input_sum_i$	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 in the example below:

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} ← W_{j,i} + α × a_j × Err_i × g'(input_sum_i)

            for each subsequent-layer in network do
```

```
            // compute the error at each node
            Δ_j ← g'(input_sum_j)Σ_i W_{j,i} Δ_i

            // update the weights leading into the layer
            W_{k,j} ← W_{k,j} + α × a_k × Δ_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 further on in this chapter) is not explicitly called out in the pseudocode above. We've presented backpropagation in algorithmic form here for readability purposes as its most approachable to the general practioner. However, to give consideration to the mathematics-minded practioner we also have a version of backpropagation explained mathematically in Appendix D.

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

In the pseudocode above we have inputs:

- network: a multilayer feed-forward neural network
- training-records: a set of training vectors w associated outputs
- learning-rate: the learning rate (sometimes termed by the greek alpha symbol)

To start the algorithm we initialize our neural network and start looping through the input examples (until we hit 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. We can see the output layer weights update being calculated with: $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(input_sum_i)$

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 the diagram below.

Figure 2-9. Updating Connections into the Output Layer

Breaking this diagram down further we 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) times 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:

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

The equation for computing the activation value of neuron *j*:

$$a_j = g\left(input_sum_j\right)$$

The error term for example *e* at neuron *i* is denoted with the notation Err_i. We denote the derivative of the activation function as $g'(x)$ and we see this being applied to the net input of neuron *i* with the term:

$$g'\left(input_sum_i\right)$$

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 where $g'(z)$ gave us the derivative of the activation function. The error term is commonly expressed as Δ_i giving us:

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

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

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

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

Gradient Descent in Weight Space

We consider backpropagation to be doing gradient descent in weights 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: $\Delta_j \leftarrow g'\left(input_sum_j\right)\Sigma_i W_{j,i}\Delta_i$

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

Updating the Hidden Layers. With the backpropagation algorithm we now walk back across the hidden layers updating the connection between each one until we reach the input layer. In the diagram below we can see a connection between 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 update 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 passed 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

The general idea with backpropagation is how we distribute the "blame" backwards 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 it has a forward connection to.

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 have to 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 code above we see the equation:

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

This equation takes a look at each node *i* in the current layer and takes the current error value delta_i and multiplies it by the weight on 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 are changed with each iteration, is known as the *learning rate*. Learning rate is a parameter we define (as opposed to being a measurement of the network's performance). We'll 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 stochastic gradient descent called "mini-batch" where 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 stochastic gradient descent 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 later 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 1980's and early 1990's we see an over-promotion of technologies such as expert systems and LISP machines, both of which failed to live up to expectations. The Strategic Computing Initiative cancels new spending at the end of this cycle. The fifth generation computer fails its goals.

Activation Functions

Activation functions are used for propagating 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 non-linearity 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 contain 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 is basically the identity function, where $f(x) = Wx$ 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, sigmoid represents a way of reducing extreme values or outliers in data without removing them. The vertical line below 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. 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), as well as 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 multi-class modeling problem yet we only care about the best score across these classes then 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") then 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.

In the case where 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

This is a more interesting transform that only activates a node 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 as we can see below in Figure 2-14.

Figure 2-14. Rectified Linear (ReLU) Activation Function

Rectified linear units are the current state of the art, because they have proven performant in many different situations. Since 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 how 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 issues with vanishing gradient. If we use hard max as the activation function we can induce sparsity in the activations output from the layer. Research has shown Deep Networks using ReLU activation functions to train well without using pre-training techniques.

Leaky ReLU

Leaky ReLUs are a strategy to mitigate the "dying ReLU" issue[5]. As opposed to having hte funciton 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.

Softplus

This activation function is considered to be the "smooth version of the ReLU", as we can see below in Figure 2-15 as plotted in comparison to the ReLU from above in Figure 2-14:

5 Li, Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes), http://cs231n.stanford.edu/, http://cs231n.github.io/

Figure 2-15. Visualizing the ReLU and Softplus Activation Functions

The softplus activation function has a similar shape to the ReLU as we can see above in the comparison plot. We also notice the differentiability and non-zero derivative of the softplus everywhere on the graph, contrary to the ReLU[6].

Loss Functions

Loss functions quantify how close a given neural network is to the ideal it is training towards. 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

6 http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

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 follow section will provide 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 below.

- 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 *ith* 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 P features.
- We will use the notation $h(X_i) = \widehat{Y_i}$, to denote the neural network transforming the input X_i to give the output $\widehat{Y_i}$. We will alter this notation a little later to emphasize its dependence on weights and biases.
- When referring to *jth* 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 *jth* feature observed in the *ith* sample collected.
- We will represent the loss function by $L(W,b)$.

Given the data available, the loss function notation indicates that its value depend 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, configuration etc, 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 and so 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'll cover loss functions appropriate for regression models.

Mean Squared Loss (MSE)

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 where we only have to predict one output feature ($M = 1$). The error in a prediction is squared and this averaged over the number of data points, plain and simple as we see in the equation below for mean squared loss (MSE):

$$L(W, b) = \frac{1}{N}\Sigma_{i=1}^{N}\left(\widehat{Y_i} - Y_i\right)^2$$

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 $\hat{Y}$ 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 mean squared error loss function.

$$L(W, b) = \frac{1}{N}\Sigma_{i=1}^{N}\frac{1}{M}\Sigma_{j=1}^{M}\left(\widehat{y_{ij}} - y_{ij}\right)^2$$

Readers familiar with linear algebra will recognize the inner sigma in the above 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 can be accounted 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). Below we see the version of MSE the DL4J library uses for regression:

$$L(W, b) = \frac{1}{2N}\Sigma_{i=1}^{N}\Sigma_{j=1}^{M}\left(\widehat{y_{ij}} - y_{ij}\right)^2$$

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 as we could have multiple parameter sets of values resulting in the same loss value.

Optimizing MSE

Optimizing the mean squared error is equivalent to optimizing for the mean.

Other Loss Functions for Regression

While the MSE is used widely, it is quite sensitive to outliers and this is something that should be considered when picking a loss function. When picking a stock to invest in, one wants to take the outliers into account. Perhaps when buying a house one doesn'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 (MAE) Loss. In a similar vein an alternative to the squared error loss is the mean absolute error (MAE) loss.

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

It simply averages the absolute error over the entire dataset.

Mean Squared Log Error (MSLE) Loss. Another loss function used for regression is the mean squared log error (MSLE)

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

Mean Absolute Percentage Error (MAPE) Loss. Finally we have mean absolute percentage error:

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

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. Say 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 Network

While MSLE and MAPE are approaches to handle 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

Neural networks can be build to bin data points into different categories, i.e fraud|not fraud etc. However when building neural networks for classification problems often times the focus is on attaching probabilities to these classifications (30% fraud | 70% fraud). These differing scenarios require different loss functions.

Hinge Loss

Hinge loss is the most commonly used loss function when the network has to be optimized for a hard classification. For example, 0 = no fraud and 1 = fraud which by convention 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 have to be categorized as "-1" or "1".

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

The hinge loss is mostly used for binary classifications. There are extensions for multi-class classification (e.g. "one vs all", "one vs 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 mean 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 will also give valid values between 0 and 1. However it cannot be used in scenarios where the outputs are mutually exclusive, as 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 where 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 below.

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

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

The above two equations can be combined and expressed as:

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

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 below.

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

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 probabil-

ities. And 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 the equation now corresponds to a 'loss'. So the loss function in question becomes the following commonly referred to as the negative log likelihood:

$$L(W, b) = -\Sigma_{i=1}^{N} y_i \times \log \widehat{y_i} + (1 - y_i) \times \log (1 - \widehat{y_i})$$

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

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

The equation above 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 the section below on reconstruction.

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 while it may not matter which is used it tends to confuse people.

Loss Functions for Reconstruction

This set of loss functions relate 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 are constrained such that the network is forced to compress the data and then recreate 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 kind of neural nets are restricted Boltzmann machines, auto encoders etc. These neural networks all use loss functions that are rooted in information theory.

Below is the equation for KL divergence.

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

While we briefly touched on the subject of cross entropy before and justified the taking the log of probabilities to turn our sum into product, we omit to mention that taking the log of the probability puts us squarely in the field of informational theory and the concept of entropy.

Different Approaches

While 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 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. In Deeplearning4j, we also refer to the optimization algorithms as updaters, since updates are synonymous with the steps the algorithm takes across the weight space to minimize error.

Hyperparameter selection is focused 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 networks' guesses. It is a coefficient that scales determine the size of the steps (updates) a neural network makes 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 may 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 we can't wait another week for the results then 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 tradeoff 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 hypothesis 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. Which 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 get 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 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 in order to effectively classify millions of images a network must be able to recognize many 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 which keeps the network from getting stuck.

CHAPTER 3

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 previous chapter we set up the foundations of machine learning and neural networks. In this chapter we'll build on these foundations to give the practitioner the core concepts of deep networks. This will help build up the reader's 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 start off by restating our definitions of both Deep Learning and Deep Networks.

What is Deep Learning?

Revisiting our definition of deep learning from chapter one, the facets that differentiate deep learning networks in general from "canonical" feed-forward multi-layer networks are:

- more neurons than previous networks
- more complex ways of connecting layers
- "Cambrian explosion" of computing power to train
- automatic feature extraction

When we write "more neurons" we mean that the neuron count has risen over the years to express more complex models. Layers have also evolved from each layer being fully connected in multi-layer networks to locally-connected patches of neurons between layers in Convolutional Neural Networks and recurrent connections to

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

More connections means 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 Queen[1] would have it no other way.

Defining Deep Networks

To further provide color to our definition of deep learning we're going to define the four major architectures of deep networks:

- Unsupervised Pre-Trained Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

There is continuous research in 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 how they evolved continuing our history lesson started in chapter 1 on the history of the feed-forward multi-layer neural network.

Deep Reinforcement Learning

Reinforcement Learning is defined in Sutton's book[2] as:

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

They go 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 yet let the agent discover the actions yielding the best reward by experimenting in a simulation.

In Reinforcement Learning the agent starts with no trained model of the environment and the utility function is synonymous with the reward or objective our agent is

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

2 http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf

after. The training system gives the agent input from the environment and rewards the agent when the outcome (of the cycle or "frame") of the simulation (or game) being played is positive. Many times actions will not only affect 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 where 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[3]. In this paper the authors used a standard algorithm (Q Learning with function approximation). Their function approximator for this algorithm was a Convolutional Neural Network. They demonstrated an agent playing Atari 2600 games based on using the pixels on the screen as input and a Convolutional Neural Network 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 leave the reader with an example in the Appendix of RL4J, the Reinforcement Learning sub-project in the DL4J suite of tools.

Evolutionary Progress and Resurgence

When we last left off in chapter 2 neural networks had hit a "winter period" in the mid-1980's when the promise of artificial intelligence fell short of what it could deliver. As many times when promising technology hits the Trough of Disillusionment (Figure 3-1 below) there were many researchers still doing important work in the realm of neural networks.

3 http://www.nature.com/nature/journal/v518/n7540/abs/nature14236.html

Figure 3-1. Trough of Disillusionment[4]

One important development in neural networks was Yann LeCun's work at AT&T Bell Labs on optical character recognition[5]. 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 Convolutional Neural Networks. This eventually led to the creation of the MNIST handwriting benchmark[6] (we'll 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[7].

4 *Diagram from: https://en.wikipedia.org/wiki/Hype_cycle*

5 http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

6 http://yann.lecun.com/exdb/mnist/

7 http://image-net.org/

Advances in modeling sequential data with recurrent neural networks appeared in the late-1980's and early 1990's by researchers including Sepp Hochreiter. As time went on the research community created better artificial neuron variants (e.g. "LSTM Memory Cell" and "Memory Cell with Forget Gate") over the course of the late 1990's. The stage for a neural network resurgence was being set quietly in research labs around the world.

Over the course of the 2000's researchers and industry applications began to progressively[8] apply these advances in products such as:

- Self-driving Cars
- Google Translate[9]
- Amazon Echo
- AlphaGo[10]

Self-driving cars in the 2006 Darpa Grand Challenge[11] used many techniques beyond just deep learning. The top teams (Stanford and CMU) 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).

AlexNet[12] 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 Convolutional Neural Networks from the 1990's. The recent advances in computer vision were less driven by recent algorithm advances and more driven 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

8 http://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html

9 https://translate.google.com/

10 https://deepmind.com/research/alphago/

11 http://archive.darpa.mil/grandchallenge/

12 https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

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

In the near future we'll continue to see deep learning be applied in unique and innovative ways. This applications 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 (e.g. "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. A few of the benchmark records deep learning as achieved in the last few years:

1. Text-to-speech synthesis (Fan et al., Microsoft, Interspeech 2014)
2. Language identification (Gonzalez-Dominguez et al., Google, Interspeech 2014)
3. Large vocabulary speech recognition (Sak et al., Google, Interspeech 2014)
4. Prosody contour prediction (Fernandez et al., IBM, Interspeech 2014)
5. Medium vocabulary speech recognition (Geiger et al., Interspeech 2014)
6. English to French translation (Sutskever et al., Google, NIPS 2014)
7. Audio onset detection (Marchi et al., ICASSP 2014)
8. Social signal classification (Brueckner & Schulter, ICASSP 2014)
9. Arabic handwriting recognition (Bluche et al., DAS 2014)
10. TIMIT phoneme recognition (Graves et al., ICASSP 2013)

11. Optical character recognition (Breuel et al., ICDAR 2013)
12. Image caption generation (Vinyals et al., Google, 2014)
13. Video to textual description (Donahue et al., 2014)
14. Syntactic parsing for Natural Language Processing (Vinyals et al., Google, 2014)
15. Photo-real talking heads (Soong and Wang, Microsoft, 2014).

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

- Automated image sharpening[13]
- Automating image upscaling[14]
- WaveNet: Generating human speech that can imitate anyone's voice[15]
- WaveNet: Generating believable classical music
- Speech Reconstruction from Silent Video[16]
- Generating fonts[17]
- Image auto-fill for missing regions[18]
- Automated Image captioning[19]
- Turning hand-drawn doodles into stylized artwork[20]

We probably won't realize all of the major commercial applications until they right in front of our faces. Understanding the advances in deep network architecture is important for the practitioner to get a better idea on application ideas going forward.

Advances in Network Architecture

As research pressed the state of the art forward from Multi-Layer Feed-forward networks towards newer architectures like Convolutional Neural Networks and Recurrent Neural Networks the discipline saw changes in how layers were setup, 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 demonstrated success with using Restricted Boltz-

13 https://github.com/alexjc/neural-enhance

14 https://github.com/nagadomi/waifu2x

15 https://deepmind.com/blog/wavenet-generative-model-raw-audio/

16 http://www.vision.huji.ac.il/vid2speech/

17 https://erikbern.com/2016/01/21/analyzing-50k-fonts-using-deep-neural-networks/

18 https://bamos.github.io/2016/08/09/deep-completion/

19 http://cs.stanford.edu/people/karpathy/deepimagesent/https://github.com/karpathy/neuraltalk2

20 https://github.com/alexjc/neural-doodle

mann Machines as layers in pre-training to build features. Convolutional Neural Networks used new and different types of 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 LSTMs. 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 convolutional networks and recurrent neural networks into a single hybrid network. Hybrid neural network architectures can allow us to leverage the best of both worlds in some cases.

From Feature Engineering to Automated Feature Learning

While deep networks may 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 [21]

Geoffrey Hinton talks about this theme in Deep Belief Networks when he explains [22] how Restricted Boltzmann Machines are used to decompose the data into higher-order features.

Categorizing Deep Belief Networks

For the purposes of this book we place Deep Belief Networks (and AutoEncoders) in the "Unsupervised Pre-Trained Networks" 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

21 http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/facebook-ai-director-yann-lecun-on-deep-learning

22 https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf

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. Hand crafting 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 hand crafted 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 have to spend more energy to get the same out of it as the latter.

Image classification is an interesting example because hand-crafting image features is harder than creating features for tabular data. The information in images is 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 convolutional neural networks.

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 convolutional neural networks we train the network to understand the edge of the nose and then the general shape of the nose from those lower level "nose-edge" features. The first layers in the network may 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 get combined into a "face" feature at the latter layers of the convolutional neural network. This allows a convolutional neural network 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 setup 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:

- Inceptionism
- Modeling Artistic Style
- Generative Adversarial Networks
- Recurrent Neural Networks

We'll quickly review each below.

Inceptionism. Inceptionism[23] is a technique where 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 where the input involves images of the sky we might see fish faces appear in clouds of the output image. This line of research out of 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. 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).

23 https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

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

Gatys (et. al, 2015) published a paper [24] titled "A Neural Algorithm of Artistic Style" where they separate the style and the content of a painting. The convolutional neural network 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.

Generative Adversarial Networks (GANs). The generative visual output of a generative adversarial network[25] 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 chapter 5 we'll take a look at an example of recurrent neural networks where 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 where 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. A sample beer review generated by the program:

24 Gatys (et. al, 2015) "A Neural Algorithm of Artistic Style" https://arxiv.org/abs/1508.06576

25 https://arxiv.org/abs/1406.2661

"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."[26]

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 hard questions on complex data easier to answer.

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

Organization of This Chapter

In this chapter we'll 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'll 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 architecture components relevant to deep networks.

Common Architectural Principals of Deep Networks

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

- parameters
- layers
- activation functions
- loss functions

26 http://spectrum.ieee.org/computing/software/the-neural-network-that-remembers

- optimization methods
- hyperparameters

We'll then take these concepts and build on them to better understand the building block networks of deep networks such as:

- Restricted Boltzmann Machines
- autoencoders

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

- Unsupervised Pre-Trained Networks
- Convolutional neural networks
- Recurrent neural networks
- Recursive neural networks

As we work our way through this chapter, we'll also drop references to how Deeplearning4j implements certain aspects of deep networks. First, let's now continue our review parameters of 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 (we 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 Deep Belief networks we see two parallel sets of feed-forward connections with two separate networks. One network's layers are composed of Restricted Boltzmann Machines (sub-networks in their own right, which we'll review below in this chapter) and is used to extract features for the other network. The other network in a Deep Belief Network is a regular feed-forward multi-layer neural network and uses the features extracted from the RBM-layer network to initialize its weights. This is just one example of many we'll see over the course of this chapter in how parameters / weights have specialized in different deep network architectures.

Parameters and NDArrays

In terms of working with the core linear algebra of deep networks Deeplearning4j 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 defined feed-forward neural networks. In chapter 2 we'll further expand this architecture with more types of layers and how they related to specific architectures of deep networks. Layers can also be represented by sub-networks in certain architectures as well. In the previous section we used the example of deep belief networks having layers composed of Restricted Boltzmann Machines.

Layers are a fundamental architectural unit in deep networks. In deeplearning4j we customize a layer by changing the type of activation function it uses (or sub-network type in the case of Restricted Boltzmann Machines). We'll also look at how combinations of layers can be used together to achieve a goal (e.g. classification or regression). 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'll 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 vs. sparse). We group these design decisions for network architecture into 2 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 illus-

trate these patterns more across deep belief networks, convolutional neural networks, and recurrent neural networks. In chapter 4 we'll 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 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 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 get 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 a whole family of Rectified Linear Units emerge in Deep Learning research such as the "Leaky ReLU". We'll see more on this topic in the chapters on tuning neural networks.

Output Layer for Regression. This design decision is motivated by what type of answer we expect out model to output. If we want to output a single real-valued number from our model we 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 val-

ues) for the single class. This real-valued output is typically interpreted as a probability distribution.

Output Layer for Multi-Class Classification. If we have a multi-class modeling problem yet we only care about the best score across these classes then 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") then 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 the last chapter 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 these 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 right activation function. An example of this would be using the Multi-Class Cross Entropy as a loss function in a layer with a softmax activation function for classification output. We cover a specialized loss function below in the reconstruction cross entropy loss function.

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 pre-train phase that involves Restricted Boltzmann Machines.

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 where 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 we want to define best set of values for the parameter vector the ones with the lowest loss function value.

In chapter 1 we introduced the basic concepts of optimization, wrote about gradient descent, and the parameter vector. In this section we'll take a look at more advanced methods of optimization and how they are used in training deep networks. For the coverage of this book we'll put 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 yet each step will take longer to calculate.

Practical Usage of Optimization Algorithms

We provide much of the details on optimization algorithms so that the practitioner is aware of the mechanics involved for reference. In later chapters in this book we simplify the discussion around optimization algorithms to give the practitioner 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:

- 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, then first-order methods calculate a gradient (Jacobian) at each step to determine which direction to go next. That means 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 Stochastic Gradient Descent 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 Stochastic Gradient Descent to converge faster.

The strengths of SGD are easy implementation and the quick processing of large datasets. SGD can be adjusted by adapting the learning rate (e.g. methods such as Adagrad, discussed below) or using second-order information (i.e., the Hessian), as we'll discuss next. SGD is also 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:

- L-BFGS[27]
- Conjugate gradient[28]
- Hessian Free[29]

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 Tradeoffs in Optimization

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

27 http://machinelearning.wustl.edu/mlpapers/paper_files/ICML2011Le_210.pdf

28 http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

29 http://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_Martens10.pdf

Limited-Memory BFGS (L-BFGS). Limited-memory BFGS (L-BFGS) is an optimization algorithm, and a so-called quasi-newton method. As it s 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 towards more promising areas of parameter space. Whereas BFGS stores the gradient's full n x 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 successive step in the line search process 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

For the purposes of this book we define hyperparameters as any configuration setting that is free to be chosen by the user, that may impact 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'll expand on the concepts from chapter 1 about hyperparameters with some new hyperparameters relevant to deep learning training.

A Few Cautionary Notes About Hyperparameters

The reader should note that some hyperparameters only apply some of the time. We'll cover the details of this more in Chapters 6 and 7. The reader should also note that changing a specific hyperparameter may 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 ouput. For the input layer this will match up to the number of features in the input vector. For the output layer this will either by 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 may push the practitioner to start 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 may vary. However, the weights on the connections as we've seen in chapter 1 are the parameters we have to train. As we include more parameters in our model we increase the amount of effort needed to train the network. Large parameter count can lead to long training times and models that struggle to find convergence.

Large Parameter Count and Overfitting

There also exists the situations where a larger model can sometimes converge easier as it will simply "memorize" the training data. In Chapter 6 we'll further discuss ways to deal with overfitting.

In chapter 6 we'll look at heuristics on how to approach 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 gets too high, we can move toward our goal faster (least amount of error for the function being evaluated), but we may 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 may take a lot longer than we'd like for our training process to complete. A small 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.

Learning rates can also be scheduled to decrease over time according to some rule. We'll learn more about this in the later chapters on tuning neural networks.

The Importance of Learning Rate as a Hyperparameter

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

Nesterov's Momentum. The "vanilla" version of Stochastic Gradient Descent uses gradient directly and this can be problematic as gradient can be nearly zero for any parameter. This causes Stochastic Gradient Descent to take tiny steps in some cases, and steps that are too big in situations where the gradient is too large. To alleviate these issues we can use techniques such as:

- Nesterov's Momentum
- RMSProp
- Adam
- AdaDelta

DL4J and Updaters

The techniques listed above (Nesterov's Momentum, RMSProp, etc) 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, the authors just wanted to note this variation for DL4J specifically.

We can speed our training up by increasing momentum, but we may 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. AdaGrad[30] 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[31] 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 towards convergence, allowing for a smoother training process.

RMSProp. The only difference between RMSProp[32] and Adagrad is that the gradient term is calculated by exponentially decaying the average and not the sum of gradients.

AdaDelta. AdaDelta[33] is a variant of Adagrad that only keeps the most recent history rather than accumulating it like Adagrad does.

ADAM. ADAM (a more recently developed updating technique at 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 in the previous chapter. Regularization is a measure taken against overfitting. Overfitting occurs when a

30 http://jmlr.org/papers/v12/duchi11a.html

31 http://cs231n.github.io/neural-networks-3/

32 http://arxiv.org/abs/1212.5701

33 http://arxiv.org/abs/1212.5701

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

- dropout
- drop connect
- L1 penalty
- L2 penalty

Dropout and drop connect 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. Dropout[34] 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 such that it will not contribute to the forward pass and back propagation.

Dropout Related to Model Averaging

We can also relate dropout to the concept of averaging the output of multiple models. If we use a dropout coefficient of 0.5 then 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. DropConnect[35] 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 on the non-sparse case, has sparse outputs, and built in feature selection. L1 regularization multiplies the

34 Bengio et al. 2015, Deep Learning (In Preparation), http://www.iro.umontreal.ca/~bengioy/dlbook/

35 Bengio et al. 2015, Deep Learning (In Preparation), http://www.iro.umontreal.ca/~bengioy/dlbook/

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 having analytical solutions, has non-sparse outputs, but 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[36] 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.

Summary

In chapter 2 we came to understand some of the basic regularization tools for feed-forward multi-layer neural networks. In this chapter we expanded this definition to some newer techniques and options in 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 multi-layer neural networks. Deep Networks combine smaller networks as building blocks (in some cases) into larger networks and use specialized set of layers in other cases. The specific building blocks we want to highlight are:

- Feed-forward Multi-layer Neural Networks
- Restricted Boltzmann Machines
- AutoEncoders

36 https://arxiv.org/abs/1206.5533

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:

- Restricted Boltzmann Machines
- Autoencoders

Both Restricted Boltzmann Machines and AutoEncoders are characterized by an extra layer-wise step for training. They are often used for the pre-training phase in other larger Deep Networks.

Unsupervised Layer-Wise Pre-Training

Unsupervised layer-wise pre-training[37] can help in some training circumstances. Over time, better optimization methods, activation functions, weight initialization methods have lessened the importance of pre-training-based Deep Networks. A case where pre-training becomes interesting is when we have a lot of unlabelled data yet only a relatively smaller set of labelled training data. Pre-training does, however, add extra overhead regarding tuning and extra training time.

Layer-Wise pre-training works by performing unsupervised pre-training of the first layer (e.g. "Restricted Boltzmann Machines") based on the input data. This gives us the first layer of weights for our main neural network (e.g. "Feed-forward Multi-Layer 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 pre-training process allows us to initialize the main neural network's parameters with good initial values.

Restricted Boltzmann Machines model probability and are great at feature extraction. They are feed-forward networks where 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. Auto encoders after training are then used as a normal feed forward neural network for activations. This is an unsupervised form of feature extraction since the neural network only uses the

37 https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf

original input for learning weights rather than backpropagation which has labels. Deep networks can use either Restricted Boltzmann Machines or Autoencoders as building blocks for larger networks (yet rare that a single network would use both). In the following sections we take a closer look at both networks.

Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBMs) are used in deep learning for:

- feature extraction
- dimensionality reduction

The "restricted" part of the name 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 deep learning pioneer who popularized Restricted Boltzmann Machines[38] use almost a decade ago, describes the more general Boltzmann machine as

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

Restricted Boltzmann Machines are also a type of AutoEncoder, which we'll talk about in a following section. Restricted Boltzmann Machines are used for pre-training 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.[39] Dr. Hinton's team is credited for key work in Restricted Boltzmann Machines 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". AlexNet[40] 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:

38 https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf

39 https://www.cs.toronto.edu/~hinton/

40 https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

> ...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 Restricted Boltzmann Machine:

- Visible Units
- Hidden Units
- Weights
- Visible Bias Unit
- Hidden Bias Unit

A standard RBM has a visible layer and a hidden layer, as seen in Figure 3-3 below. We can also see a graph of weights (connections) between the hidden and visible units. Think of these weights in the same way you think of weights in the classical neural network sense.

Figure 3-3. Restricted Boltzmanm Machine Network

With Restricted Boltzmann Machines every visible unit is connected to every hidden unit, yet no units from the same layer are connected. Each layer of an Restricted Boltzmann Machines 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 a Restricted Boltzmann Machines, 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 multi-layer 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 computation. "On" means they pass a signal further through the net; "off" means they don't.

Usually, being "on" means the data passing through the node is valuable, as 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 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 where an input node is always on or always off.

Training

The technique known as "pre-training" using Restricted Boltzmann Machines 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. Restric-

ted Boltzmann Machines learn to reconstruct the input dataset. We'll review the concept of reconstruction in the next section.

Contrastive Divergence

Restricted Boltzmann Machines calculate gradients using an algorithm called contrastive divergence[41]. Contrastive divergence is the name of the algorithm used in sampling for the layer-wise pre-training of a restricted boltzmann machine. 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 pre-training (RBMs, AutoEncoders) perform feature engineering from unlabelled data through reconstruction. In pre-training, the weights learned through unsupervised pre-train 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).

In Figure 3-4 below we see a visual explanation of the network involved in reconstruction in restricted boltzmann machines.

41 https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf

Figure 3-4. Reconstruction in Restricted Boltzmann Machines

We can visually explain reconstruction in Restricted Boltzmann Machines work by looking at the MNIST dataset[42]. 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 zero through nine. A sample of some of the handwritten digits in MNIST are below.

42 http://yann.lecun.com/exdb/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 Restricted Boltzmann Machine to learn the MNIST dataset we can sample from the trained network to see[43] how well it can reconstruct the digits. In the next image below we can see renders of MNIST digits being progressively reconstructed with a Restricted Boltzmann Machine:

43 http://yosinski.com/media/papers/Yosinski2012VisuallyDebuggingRestrictedBoltzmannMachine.pdf

Figure 3-6. Reconstructing MNIST Digits with Restricted Boltzmann Machines

If the training data has a normal distribution then 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, then we can reconstruct that curve. But let's say we don't know the mean and variance. Those are parameters we then have to guess at. 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 you 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.

Other Uses of Restricted Boltzmann Machines

Other places we see Restricted Boltzmann Machines used are:

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

Autoencoders

Autoencoders are used to learn compressed representations of datasets. Typically Autoencoders are used to reduce dataset's dimensionality. The output of the Autoencoder network is a reconstruction of the input data in the most efficient form.

Similarities to Multi-Layer Perceptrons

Autoencoders share a strong resemblance with Multi-Layer 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 Multi-Layer Perceptron network diagram (from earlier chapters) and Autoencoder diagram is the output layer in an Autoencoder has the same number of units as the input layer does.

We can see this below in Figure 3-7 where we see an example of Autoencoder network:

Figure 3-7. Autoencoder Network Architecture

Beyond the output layer there a few other differences as we outline in the next section.

Defining Features of Autoencoders

Autoencoders differ from Multi-Layer Perceptrons in that they:

- use unlabelled data in unsupervised learning
- build a compressed representation of the input data

Unsupervised Learning of Unlabelled Data. The Autoencoder learns directly from unlabeled data. This is connected to the second major difference between Multi-Layer Perceptrons and Autoencoders.

Learning to Reproduce the Input Data. The goal of a Multi-Layer Perceptron network is to generate predictions over a class (e.g. "fraud" vs "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 Restricted Boltzmann Machines 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 in Figure 3-7 above. The network input must pass through a "bottleneck" region of the network before being expanded back into the output representation.

Denoising Autoencoders. The Denoising Autoencoder[44] is the scenario where the Autoencoder is given a "corrupted" version (e.g. "some features 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 may 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 network

44 http://www.jmlr.org/papers/volume11/vincent10a/vincent10a.pdf

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 where we know what the normal data will look like yet its 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 Welling[45]. The variational autoencoder 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 VAE use to perform training is quite different. In a compression/denoising autoencoder, activations are mapped to activations throughout the layers as in a standard neural network; comparatively, a variational autoencoder uses a probabilistic approach for the forward pass.

Figure 3-8. Variational Autoencoder Network Architecture

45 https://arxiv.org/abs/1312.6114

The VAE model assumes that the data x is generated in 2 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, then 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[46]. Overall, the network is trained by backpropagation to maximize a lower bound on the marginal likelihood of the training data, $\log p\left(x^1, ..., x^N\right)$. The variational autoencoder model has also been extended to allow unsupervised learning on time series with the variational *recurrent* autoencoder[47]. Later on in the book in Chapter 5 we'll see the VAE in a practical example generating MNIST digits.

46 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 Guassian distribution, or the mean value for Bernoulli distribution.

47 https://arxiv.org/abs/1412.6581

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 Pre-Trained Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

In this chapter we'll take a look in more detail of each of these four major architectures. In chapter 2 we gave the reader a deeper understanding of the algorithms and math that underlie neural networks in general. In this chapter we'll 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 practitioners will see in the wild: Convolutional Neural Networks for image modeling and Long Short-Term Memory Networks (Recurrent Networks) for sequence modeling.

Unsupervised Pre-Trained Networks

In this group we cover three specific architectures:

- AutoEncoders
- Deep Belief Networks
- Generative Adversarial Networks

A Note About the Role of AutoEncoders

We previously covered AutoEncoders in Chapter 3 as a fundamental structure in Deep Networks as its often used as part of larger networks. Like many other networks, they serve that role and then are used as a stand-alone network as well.

Given the previous coverage of AutoEncoders, we'll move on to looking closer at Deep Belief Networks and Generative Adversarial Networks.

Deep Belief Networks

Deep Belief Networks are composed of layers of Restricted Boltzmann Machines for the pre-train phase and then a feed-forward network for the fine-tune phase. Below in Figure 4-1 we can see a diagram showing the network architecture of a deep belief network:

Figure 4-1. Deep Belief Network Architecture

In the sections following we'll explain more of how Deep Belief Networks leverage Restricted Boltzmann Machines to better model the training data.

Feature Extraction with Restricted Boltzmann Machine Layers

We use Restricted Boltzmann Machines to extract higher-level features from the raw input vectors. To do that 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 Restricted Boltzmann Machines in the context of Deep Learning and Deep Belief Networks are 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 Restricted Boltzmann Machines learn progressively higher level features using the learned features from a lower level Restricted Boltzmann Machine pre-train layer as the input to a higher-level Restricted Boltzmann Machine pre-train layer.

Learning Higher Order Features Automatically. Learning these features in an unsupervised fashion is considered the pre-train phase of deep belief networks. Each hidden layer of Restricted Boltzmann Machines in the pre-train 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 Restricted Boltzmann Machines let's look at the activation renders on a Restricted Boltzmann machine as it learns MNIST digits:

Figure 4-2. Activation Render at Start of Training

progressively...

Figure 4-3. Features Emerge in Later Activation Render

and then ...

Figure 4-4. Portions of MNIST Digits Emerge Towards End of Training

We'll learn more about how these renders are produced in chapter 6. We can see how a layer of Restricted Boltzmann Machine extracts fragments of digits as it trains. These features are then combined in higher level layers of Restricted Boltzmann Machines to build progressively more complex (and non-linear) features.

As this raw data is modeled in the generative modeling process with each layer of RBMs the system is able to extract further 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 back-propagation driven feedforward neural network.

These initialization values help the training algorithm guide the parameters of the traditional neural network towards better regions of parameter search space. This phase is known as the fine-tune phase of deep belief networks.

Fine Tuning a Deep Belief Network with a Feed-Forward Multi-Layer Neural Network

In the fine-tune phase of a deep belief network we use normal backpropagation with a lower learning rate to do "gentle" backprogation. We consider the pre-training 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 pre-train phase with Restricted Boltzmann Machine layers 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 Deep Belief Networks

We don't cover Deep Belief Networks as extensively as the other network architectures in this book. This is because the field has largely seen Convolutional Neural Networks take over the image modeling space and so we chose to emphasize that architecture more as we'll see in the next section.

The Role of Deep Belief Networks in the Rise of Deep Learning

Although we don't emphasize Deep Belief Networks as much in this book this network played a non-trivial 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. The authors felt it important to note Deep Belief Networks role in the evolution of Deep Networks.

Generative Adversarial Networks

A network of note is the Generative Adversarial Network (GAN)[1]. Generative Adversarial Networks have been shown to be quite adapt at synthesizing novel new images based on other training images. This concept can be extended to model other domains such as:

- sound[2]
- video[3]

1 https://arxiv.org/abs/1406.2661

2 https://github.com/usernaamee/audio-GAN

3 http://web.mit.edu/vondrick/tinyvideo/paper.pdf

- generating images from text descriptions[4]

GANs are an example of a network that leverages unsupervised learning to train two models in parallel. A key aspect of GANs (and generative models in general) are how they use a parameter count that is significantly smaller than normal with respect to the amount of data we're training the network on. 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[5] dataset) we could build a generative neural network which outputs images (as opposed to a classification). We'd consider these generated output images to be "samples from the model". The generative model in GANs generate such images while a secondary discriminator network try to classify these generated images.

This secondary "discriminator" network attempts to classify the output image 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 make images realistic enough that the discriminator network is fooled to the point it cannot tell the difference between the "real" and the "synthetic" input data.

An example of efficient model representation in GANs is how GANs typically have around 100 million parameters when modeling a dataset such as ImageNet. In the course of training an input dataset such as ImageNet (200GB) becomes close to 100MB 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 more of later in this chapter in the Convolutional Neural Network section).

The Discriminator Network. When modeling images, the discriminator network is typically a standard Convolutional Neural Network (as introduced in a later section). Leveraging 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 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.

4 https://arxiv.org/abs/1612.03242

5 http://image-net.org/

The Generative Network. The generative network in GANs generate data (or images) with a special kind of layer called a "deconvolutional layer" (described in the section on "deconvolutional networks" below).

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 where 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 NYU as part of the development of ZF Net in the paper "Visualizing and Understanding Convolutional Neural Networks" (2013)[6]. A deconvolutional network helps us examine different feature activations and their relation to the input space.

Figure 4-5. Visualizing Deconvolutional Layers

The deconvolutional network ("deconvnet" for short) maps features to pixels when modeling images as we see above in #visualization-of-feat which is the reverse of what a normal convolutional layer does. This aspect of deconvolutional networks is what enables us to generate images as output from neural networks.

6 https://arxiv.org/abs/1311.2901v3

Building Generative Models and DCGANs

One variant of GANs is the Deep Convolutional Generative Adversarial Network[7] (DCGAN). In Figure 4-6 below we see images of bedrooms generated from the DCGAN:

Figure 4-6. Generated Images of Bedrooms from a DCGAN Network[8]

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 GANs[9] can also 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 from the model distribution or the real distribution. When the discriminator model makes a prediction where there a difference between the two distributions then 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.

7 https://github.com/Newmu/dcgan_code

8 *Image is from the DCGAN authors' github repository at: https://github.com/Newmu/dcgan_code*

9 https://arxiv.org/abs/1411.1784

With Variational AutoEncoders we're setting this same problem up with probabilistic graphical models to reconstruct the input in an unsupervised fashion as seen previously in chapter 3. Variational AutoEncoders 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 an 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 so we can compare the generated image to the original 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 different type of noise in the generated output image. The downside of VAE generated images is due to 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. "its an image of a dog, but the dog doesn't look quite right").

Convolutional Neural Networks

A convolutional neural network's goal is to learn higher order features in the data via convolutions. Convolutional networks 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. Convolutional networks overlap with text analysis via optical character recognition, but they are also useful when analyzing words[10] as discrete textual units, as well as sound.

The efficacy of convolutional networks in image recognition is one of the main reasons why the world has woken up to the power of deep learning. Convolutional networks are good at building position and (somewhat) rotation invariant features from raw image data as we can conceptually below in Figure 4-7:

10 https://arxiv.org/abs/1610.10099

Figure 4-7. Convolutional Neural Networks and Computer Vision

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

Convolutional Networks and Structure in Data

Convolutional Neural Networks tend to be most useful when there is some structure to the input data. An example would be how images and audio data have a specific set of repeating patterns and input values next to each other are related spatially. Conversely, the columnar data exported from a 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.

Convolutional neural networks have also been used in other tasks, such as natural language translation/generation[11] and sentiment analysis[12]. A convolution is a powerful concept for helping to build a more robust feature space based on a signal.

11 https://arxiv.org/abs/1611.02344

12 http://www.aclweb.org/anthology/C14-1008

Biological Inspiration

The biological inspiration for convolutional networks is the visual cortex in animals[13]. The cells in the visual cortex are sensitive to small sub-regions of the input. We call this the "visual field" (or receptive field). These smaller sub-regions 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 have a larger receptive field and are invariant to the position of the pattern.

Intuition

Feed-forward multi-layer 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 these networks don't scale well with image data as input. An example would be modeling the CIFAR-10 dataset (described below in the breakout). The images to train on are only 32 pixels wide by 32 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 multiple those weights as well.

What is the CIFAR-10 Dataset?

The CIFAR-10 dataset is a well known image classification benchmark dataset[14] put together by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset has 60,000 color images with 10 different classes with 6,000 images per class. The images are 32x32 pixels each. There are 50,000 training images and 10,000 test images. The classes of images in the dataset are:

13 http://www.sciencedirect.com/science/article/pii/S1053811916305481

14 http://www.cs.toronto.edu/~kriz/cifar.html

Figure 4-8. CIFAR-10 Dataset

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

A normal image could easily be 300 pixels in width by 300 pixels in heights with 3 channels of RGB information. This would create 270,000 connection weights per hidden neuron. This shows how quickly a fully connected multi-layer network becomes as image size scales up. The structure of image data allows us to change the architecture of a neural network in a way such that we can take advantage of this structure. With convolutional networks we can arrange the neurons in a three dimensional structure where we have

- Width
- Height
- Depth

These attributes of the input match up to an image structure where we have

- The image width in pixels
- The image height in pixels

- The RGB channels as the depth

We can consider this structure to be a 3D volume of neurons. A significant aspect to how convolutional neural networks evolved from previous feed-forward variants is how they achieved computational efficiency with new layer types. We'll cover this arrangement in more depth below. Let's now take a look at the high-level general architecture of convolutional neural networks.

Convolutional Network Architecture Overview

Convolutional networks 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 convolutional neural network architecture but they are based on the pattern of layers below in Figure 4-9:

Figure 4-9. High-Level General Convolutional Neural Network Architecture

We see 3 major groups inside diagram above:

1. Input Layer
2. Feature Extraction ("Learning") Layers
3. Classification Layers

The input layer accepts 3-dimensional input generally in the form spatially of the size (width x height) of the image and has a depth representing the color channels (generally 3 for R,G,B color channels).

Mini-Batch as the 4th Dimension

When we batch examples together into a mini-batch for training, we end up with 4 dimensions – one more dimension to index the example within the mini-batch. Thus in DL4J, image training data arrays have 4 dimensions, not just 3.

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

1. Convolution Layer
   - We express the 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 where features are automatically learned as opposed to traditionally hand engineered.

Finally we have the classification layers where we have 1 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 produce typically a 2-dimensional output of the dimensions [b x 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 layers of a convolutional network are arranged in 3 dimensions to match the input volumes. Here depth means the third dimensions of the activation volume and 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 only connected to a small region of neurons in the layer before it. Convolutional networks 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 may or may not have parameters as we'll see visually demonstrated in a diagram below.

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 will have a width, a height, and a number of channels. Typically the number of channels is 3 for the R, G, B values for each pixel.

Figure 4-10. Input Layer 3D Volume

Convolutional Layers

Convolutional layers are considered the core building block of convolutional network architectures. 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 they are locally connected to 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 though the use of fourier transforms. It takes input, applies a convolution kernel, and gives us a feature map as output.

The convolution operation is known as the "feature detectors" of convolutional neural networks. The input to a convolution can be raw data or a feature map output from another convolution. It is often interpreted as a filter where the kernel filters input data for certain kinds of information (e.g. where an edge kernel only lets information from the edge of an image pass through).

Figure 4-12. Convolution Operation

As we see in the diagram above, 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 inside 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 hyper parameters. Gradient descent is used to train the parameters in this layer such that the class scores will be consistent with the labels in the training set. Major components of convolutional layers are:

- filters
- activation maps
- parameter sharing
- layer specific hyperparameters

Let's take a look of the specifics of these below.

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 NLP Applications

We may have a filter size equal to the input volume but typically only in one dimension, not both. This would be something to watch out for in the cases where we'd apply Convolutional Neural Networks to use cases in natural language processing.

Filters (e.g. convolutions) are applied across the width and height of the input volume in a sliding window manner as we saw above 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 convolutional neural network diagrams we often see lots of small activation maps and how these are produced can be confusing sometimes.

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 3rd 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 convolutional networks is setup 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 receptive field. As we get farther along in layers in a convolutional network we get filters that can recognize non-linear combination of features and are increasingly global in how they can detect patterns. High performing convolutional architectures that we'll see later in this section have shown network depth to be an important factor in convolutional networks.

Activation Maps

If we recall from chapter 1, an activation is numerical result of 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 the filter let 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 a convolutional network. This produces a 2-dimensional output called an activation map for that specific filter. In Figure 4-13 below we can see 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 is rendered differently to illustrate how convolutional activation maps are commonly rendered in 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 which 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 3-dimensional output volume for the convolution layer by stacking these activation maps along the depth dimension in the output. The output volume

will have entries that we consider the output of a neuron that looked 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 we can see below in Figure 4-15:

Figure 4-15. Generating an Activation Output Volume

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

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 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) we discussed above.

In this situation the input volume has the size 32x32x3 and our receptive field hyperparameter is set as 5x5. Each neuron in the convolutional layer will have weights to a [5x5x3] region in the input volume. This gives us 5 * 5 * 3 = 75 weights for each neuron in our convolution layer.

We note the depth of the input volume is 3 which is always the depth of 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 may be local but the neurons themselves remain unchanged. We're still computing a dot product of the weights with the input with a non-linear 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 above. 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 with a technique called parameter sharing.

Parameter Sharing

Convolutional networks 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 convolutional networks we first denote a single 2-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 Convolutional Neural Networks invariance to translation/position as well.

Learned Filters and Renders

Below in Figure 4-16 we see an example of the learned 96 filters of size [11x11x3]. 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 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[15] (96 filters, [11x11x3])

Breaking this up a bit, let's think about a 2D image. If we subdivide an image into 4 sections, the neural network will then learn position-invariant features of the image. The reason this is 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 4.

Rotation Invariance in Features for Convolutional Neural Networks

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

15 *https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf*

ReLU Activation Functions as Layers

With convolutional networks we often see ReLU layers used. The ReLU layer will apply an element-wise activation function over the input data thresholding (e.g. 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 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

The hyperparameters[16] 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 Convolutional Neural Network 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 may have a 5x5x3 sized filter. This would mean the filter is 5 pixels wide, by 5 pixels tall, and 3 represents the color channels assuming the input image was in 3-channel RGB color.

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.

16 Another hyperparameter commonly seen in Convolutional Neural Networks is called dilation. DL4J current does not support dilation as of version 0.7.

Edges and Activation

Different neurons along the depth dimension learn to activate when stimulated with created input data (eg: 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 (described above) 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 (eg: 1 gives only a single unit step) will allocate more depth columns in the output volume. This will also yield more heavily overlapping receptive fields between the columns, leading to larger output volumes. The opposite is true for 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 and allows us to control the spatial size of the output volumes. We'd want to do this in cases where we want to maintain the spatial size of the input volume in the output volume.

Batch Normalization and Layers

To accelerate training in Convolutional Neural Networks we can normalize the activations of the previous layer at each batch[17]. 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.

Applying batch normalization in Convolutional Neural Networks has shown to speed up training[18] 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 towards weight initialization and act as a regularizer (reducing the need for other types of regularization)[19]. Batch normalization has also been applied to LSTM networks[20], another type of deep network we'll discuss later on in this chapter.

17 https://arxiv.org/abs/1502.03167

18 https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md

19 https://www.quora.com/Why-does-batch-normalization-help

20 https://arxiv.org/abs/1603.09025

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 helps 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 would be 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 2x2 filter size effectively what we see happening here is the MAX operation is taking the largest of 4 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 if the input image was 32 pixels wide by 32 pixels tall the output image would be smaller in width and height (eg: 16 pixels wide by 16 pixels tall). The most common setup for a pooling layer is to have filters of size 2x2 applied with a stride of 2. This will downsample each depth slice in the input volume by a factor of 2 on the spatial dimensions (width and height). This downsampling operation will result in 75% of the activations being discarded.

Pooling layers do not have parameters for the layer but do have additional hyperparameters as listed below in the summary table. 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

This layer is used to compute class scores that we'll use as output of the network (e.g. "output layer" at the end of the network). The dimensions of the output volume is [1 x 1 x 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 convolutional neural network architectures that will use multiple fully-connected layers at the end of the network. AlexNet[21] is an example of this where it has 2 fully-connected layers followed by a softmax layer at the end.

Other Applications of Convolutional Networks

Beyond normal 2-dimensional image data we also see Convolutional Neural Networks applied to 3-dimensional datasets. Examples of these alternative uses are:

- MRI data[22]
- 3D shape data[23]
- Graph data[24]
- NLP applications[25]

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

Convolutional Network Architectures of Note

Below we list some of the more popular architectures of Convolutional Networks.

- LeNet[26]
  - One of the earliest successful architectures of Convolutoinal Networks.
  - Developed by Yan Lecun
  - Originally used to read digits in images
- AlexNet[27]
  - Helped popularize Convolutional Networks in Computer Vision
  - Developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton

21 http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf

22 https://arxiv.org/abs/1606.04797

23 https://www.ri.cmu.edu/pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf

24 https://arxiv.org/abs/1506.05163

25 https://arxiv.org/abs/1606.01781

26 http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

27 https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

  - Won the ImageNet Large-Scale Visual Recognition Challenge 2012 (ILSVRC 2012)
- ZF Net[28]
  - Won the ImageNet Large-Scale Visual Recognition Challenge 2013 (ILSVRC 2013)
  - Developed by Matthew Zeiler and Rob Fergus
  - Introduced the visualization concept of the Deconvolutional Network
- GoogLeNet[29]
  - Won the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC 2014)
  - Developed by Christian Szegedy and his team at Google
  - Codenamed "Inception", one variation has 22 layers
- VGGNet[30]
  - Runner-Up in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC 2014)
  - Developed by Karen Simonyan and Andrew Zisserman
  - Showed that depth of network was a critical factor in good performance
- ResNet[31]
  - Trained on very deep networks (up to 1200 layers)
  - Won the 1st place on the ILSVRC 2015 classification task

Summary

Convolutional neural networks evolved due to the need for specialized feature extraction from image data. We see 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 network architecture focused on modeling the temporal domain: Recurrent Neural Networks.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are in the family of feedforward neural networks. They are different from other feedforward networks in their ability to send

28 https://arxiv.org/pdf/1311.2901v3.pdf

29 http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf

30 https://arxiv.org/pdf/1409.1556v6.pdf

31 https://arxiv.org/abs/1512.03385

information over time steps. An interesting explanation of recurrent neural networks from leading RNN 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, RNN 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 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 are considered Turing Complete and can simulate arbitrary programs (with weights). If we view neural networks as optimization over functions then we can consider recurrent neural networks as "optimization over programs". Recurrent neural networks are well suited for modeling functions where 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 feedforward 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 (eg: the previous, current, and next input together as a single input vector).

A drawback of these tools is 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 network could compete in Alan Turning's famed "Turing Test", for instance, to attempt to fool another user into thinking it was a real person.

Temporal Feedback and Loops in Connections

Recurrent networks can have loops in the connections. This allows recurrent networks to model temporal behavior and allows them to gain accuracy in domains such as timeseries, 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. The authors 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. Recurrent network's wiring allows for feedback in ways that allow it to capture these temporal effects. We primarily see Recurrent architectures applied in the timeseries 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, where 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 on in this chapter.

Sequences and Timeseries Data

We find sequential data in many problem domains in industry. In problem domains where our model needs to output a sequence of vectors:

- Image captioning[32]
- Speech synthesis[33]
- Music generation[34]
- Playing video games

32 http://cs.stanford.edu/people/karpathy/sfmltalk.pdf

33 http://www.jmlr.org/proceedings/papers/v32/graves14.pdf

34 https://cs224d.stanford.edu/reports/NayebiAran.pdf

- Language Modeling[35]
- Character-Level Text Generation Models[36]

In other domains we need a sequence of input vectors:

- Timeseries prediction
- Video analysis
- Music information retrieval

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

- Natural language translation[37]
- Engaging in dialogue
- Robotic control

Recurrent neural networks contrast with other deep networks in what type of input Recurrent Networks can model (non-fixed input):

- Non-fixed computation steps
- Non-fixed output size
- We can operate over sequences of vectors
  - eg: 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
- fixed output size

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

Recurrent neural networks change up this input dynamic to include multiple input vectors, one for each timestep, and each vector can have many columns. In the list below we see examples of how Recurrent Networks operate on sequences of input and output vectors:

35 http://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf

36 https://karpathy.github.io/2015/05/21/rnn-effectiveness/

37 https://arxiv.org/abs/1409.3215

- one-to-many
  - "sequence output"
  - e.g. "image captioning takes an image and outputs a sequence of words"
- many-to-one
  - "sequence input"
  - eg: "sentiment analysis where a given sentence is input"
- many-to-many
  - e.g. "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 Convolutional Neural Networks. We have 3 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 timeseries 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 timesteps is how we represent the change in the input vector over time. This is the timeseries 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 Timeseries and Masking

We previously described how with recurrent neural network input we have the concept of timesteps in addition to features in our input vector. We can see this visually below in Figure 4-18:

Figure 4-18. The Timestep Aspect of Recurrent Neural Network Input

Every column value likely will not occur at every timestep, especially in the case where we're mixing descriptor data (e.g. "columns from a static database table") with timeseries data (e.g. "measurements of an ICU patient's heartrate every minute"). In this case where we have "jagged" timestep values we need to use "masking" to let DL4J know where we our real data is located in the vector. We do this by providing an extra matrix for the mask indicating the timesteps that have input data for at least one column. We can see this visually in Figure 4-19 below:

Figure 4-19. Masking Specific Timesteps

In the examples in chapter 3 we'll see real code examples on how we set these masks up to let the modeling engine know where our data is located.

Why Not Markov Models?

When dealing with the time-dimension in our models we naturally could consider Markov models[38] 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 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 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. These networks are able to do this because 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

38 https://en.wikipedia.org/wiki/Hidden_Markov_model

makes recurrent 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 the number of nodes in the hidden layer

If the output was 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 while the expressive power of the network grows exponentially with the number of these hidden nodes.

General Recurrent Network Architecture

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

Recurrent Architecture and Timesteps

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 time 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 multi-layer neural networks.

Vanishing Gradient Problem

Recurrent neural networks are known to have issues with the "vanishing gradient problem". This issue occurs when the gradients get too large or too small and make it

difficult to model long-range dependencies (10 timesteps or more) in the structure of the input dataset [27]. The most effective way to get around this issue in recurrent neural networks is to use the Long Short-Term Memory (LSTM) variant of recurrent neural networks which DL4J supports.

Long Short-Term Memory (LSTM) Networks

Long short-term memory (LSTM) networks are the most commonly used variation of Recurrent Neural Networks. LSTM Networks were introduced in 1997 by Hochreiter and Schmidhuber[39].

The critical component of the LSTM[40] is the memory cell and the gates (including the forget gate[41], but also the input gate). The contents of the memory cell are modulated by the input gates and forget gates[42]. Assuming 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 a major problem ("vanishing gradient") with most recurrent network models.

Properties of Long Short-Term Memory Networks

LSTMs are known for:

- Better update equations
- Better backpropagation

Some example use cases of LSTMs are:

- Generating sentences (e.g. "character level language models")
- Classifying timeseries
- Speech recognition
- Handwriting recognition
- polyphonic music modeling

39 http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf

40 http://www.cs.toronto.edu/~graves/preprint.pdf

41 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.5709&rep=rep1&type=pdf

42 http://www.cs.toronto.edu/~graves/phd.pdf

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

- Image captioning
- Language translation
- Handwriting recognition

LSTM Networks consist of many connected LSTM cells (covered below) and perform well in how efficient they are during learning.

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 Multi-Layer Feed-Forward Neural Network as seen in Figure 4-20 below:

Figure 4-20. Multi-Layer Feed-Forward Neural Network

If we were to show each layer from the network above in Figure 4-20 as a single node in a flattened representation it would look like Figure 4-21:

Figure 4-21. Visually Flattened Multi-Layer Feed-Forward Network

With Recurrent 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. This recurrent connection allows us to take input from the previous "timestep" into the neuron as part of the incoming information.

By flattening the network visually above in Figure 4-21 it allows us to more easily visualize the recurrent connections in a LSTM. We can see Figure 4-21 above with these added recurrent connections in Figure 4-22 below:

Figure 4-22. Showing Recurrent Connections on Hidden Layer Nodes

We can further visually understand this by "unrolling" the network diagram above in Figure 4-22 to show how the information "flows" through the network in a feedforward manner and "across time". We see this with the time dimension illustrated in the un-rolled diagram below:

Figure 4-23. Recurrent Neural Network Un-Rolled 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 units has two types of connections:

1. connections from the previous timestep (outputs of those units)
2. connections from the previous layer

The memory cell in a 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 we see below in Figure 4-24.

Figure 4-24. LSTM Block Diagram

The components in a LSTM unit are:

- 3 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 3 gate units which learn to protect the linear unit from misleading signals.

1. The input gate protects the unit from irrelevant input events
2. The forget gate helps the unit forget previous memory contents
3. The output gate is for exposing the contents of the memory cell (or not) at the output of the LSTM unit

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 a 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[43] we have the vector formulas for a LSTM layer forward pass below:

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

Figure 4-25. Vector Formulas for LSTM Layer Forward Pass

Below we have a description for what the individual variables in the equations above represent.

Table 4-1. Variable table for LSTM vector formulas

variable name	description
x^t	input vector at time t
W	rectangular input weight matrices
R	square recurrent weight matrices
p	peephole weight vectors
b	bias vectors

43 http://arxiv.org/pdf/1503.04069v1.pdf

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 discover longer-range events in sequences. These events can span up to 1000 discrete timesteps compared older recurrent architectures that could only model events across around 10 timesteps.

For More Variants of LSTMs

Check out the paper "LSTM: A Search Space Odyssey":

http://arxiv.org/pdf/1503.04069v1.pdf

Gated Recurrent Units (GRU)

Another similar recurrent unit to the LSTM is the Gated Recurrent Unit (GRU)[44]. The GRU has a reset gate and an update gate, similar to the forget/input gates in the LSTM unit. The major difference is the GRU fully exposes its memory content using only leaky integration (but with 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 (non-fixed) 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 gets 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 non-input unit computes its current activation at any given timestep. This activation value is computed as the non-linear 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 computing by the net-

44 https://arxiv.org/abs/1406.1078

work. Let's now take a look at the variant of backpropagation used with recurrent neural networks, including LSTMs, called "back propagation through time".

Backpropagation Through Time and Truncated Backpropagation Through Time

Recurrent neural network training can be computationally expensive. The traditional option is to use backpropagation through time.

> **Backpropagation Through Time**
>
> Backpropagation through time is fundamentally the same idea 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 backwards from future time steps to current time steps, not just from the layer above (as occurs in standard backprop).

When our recurrent network is dealing with long sequence with many time steps, we recommend alternatively using "truncated backpropagation through time"(truncated BPTT). Truncated BPTT reduces the computational complexity of each parameter update in a recurrent neural network.

> **Recurrent Neural Networks and Back Propagration**
>
> Computing the gradient for a recurrent neural network on sequence of length 1000 has the same computational cost as doing a forward and backward pass on a multi-layer perceptron network that has 1000 layers [28].

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 we'd do a backward pass of the 12 time steps as we see below in Figure 4-26:

Figure 4-26. Standard Back Propagation Through Time

In the example above the 12 time steps are not that difficult for the training process. As we move into models that train on timeseries data of a few hundred steps or more we find training to be more difficult. If the number of steps in the timeseries input was 1,000 steps we'd see that standard backpropagation training would require 1,000 time steps for each forward and backward passes (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. As we can see in the diagram below truncated BPTT 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 diagram below we can see that the BPTT truncated pass size is 4 time steps.

Figure 4-27. Truncated Back Propagation Through Time

Truncated BPTT is considered the current most practical method for training recurrent neural networks [28]. With truncated BPTT we can capture long-term dependencies with less of a computational burden than regular BPTT.

The overall complexity for standard BPTT and truncated BPTT are similar and do 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 may end up being shorter than the full BPTT algorithm's results. In practice this speed tradeoff is generally worth it as long as truncated BPTT lengths are set correctly.

Domain Specific Applications and Blended Networks

As mentioned above RNNs perform many domain specific applications such as speech transcription to text, machine translation, and generation of handwritten text. Recurrent neural networks have shown adept in the world of computer vision. These tasks include:

- Frame-level video analysis[45]
- Image captioning
- Video captioning[46]
- Visual question answering[47]

Another emerging area of research in computer vision with RNNs is a network that can extract information from an image by processing only a small region at a time, called "Recurrent Models of Visual Attention"[48]. These models are effective at working with images that are cluttered with multiple objects and are harder for Convolutional Neural Networks to classify. These networks blend together Convolutional Neural Networks 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 where the network generates natural language descriptions of images and their regions[49]. The model is able to generate captions for images by leveraging datasets of images and their corresponding description sentence as we can see in Figure 4-28 below:

45 https://arxiv.org/abs/1502.04681

46 https://arxiv.org/abs/1412.4729

47 https://arxiv.org/abs/1603.02814

48 https://arxiv.org/abs/1406.6247

49 https://arxiv.org/abs/1412.2306v2

Figure 4-28. Labeling Images with a Blended CNN + RNN Network[50]

This type of network is a combination of a Convolutional Neural Network and a bidirectional Recurrent Neural Network.

Recursive Neural Networks

Recursive neural networks, like recurrent neural networks, can deal with variable length input. The primary difference is that recursive networks have the ability to model the hierarchial structures in the training data set. Images commonly have a scene composed of many objects. Deconstructing scenes is often a problem domain of interest yet non-trivial. The recursive nature of this deconstruction not only challenges us to identify the objects in the scene but how the objects relate to form the scene.

50 *Image is from the paper https://arxiv.org/abs/1412.2306v2*

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 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. The feedforward pass happens bottom-up, and backpropagation is top-down. Think of the objective as the top of the tree, while 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 feedforward cousin, recursive autoencoders learn how to reconstruct the input. In the case of natural language processing, it learns how to reconstruct contexts. A semi-supervised recursive autoencoder learns a likelihood of certain labels in each context.

Another variation of this is a supervised neural network, called a recursive neural tensor network. Recursive neural tensor networks compute a supervised objective at each node of the tree. The tensor part of this means 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. Recursive neural networks are traditionally used in natural language processing 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 networks it is a constraint that we use a parse that builds the tree structure (typically constituency parsing).

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

- image scene decomposition
- natural language processing
- audio to text transcription.

Two specific network configurations we see in practice are recursive autoencoders and recursive neural tensors. Recursive autoencoders are used to break up sentences into segments for natural language processing. Recursive neural tensors are used 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 and are typically used 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:

- To generate data (e.g. "images", "audio", or "text") we'd use:
  - Generative Adversarial Networks
  - Variational Autoencoders
  - Recurrent Neural Networks
- To model images we'd likely use
  - Convolutional Neural Networks
  - Deep Belief Networks
- To model sequence data we'd likely use
  - Recurrent Neural Networks / LSTMs

In the following chapters, we will illustrate real world code examples of most of these networks and cover considerations of training and tuning for different kinds of neural networks. In the next chapter we'll see how these concepts come together in API examples where we see the DL4J Deep Learning library in action. Before we get on to some more examples, let's discuss a few topics that come up frequently on 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 have to gauge level of effort against 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 hand-crafted feature construction.

Different Problems Have Different Best Methods

Machine learning on the whole is about applying the right approach in the right situation. We're not to the point yet where a single technique dominates the landscape

so we have 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 may 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 the reader try and fit the data with a non-linear model (e.g. a "multi-layer perceptron")? No, we'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 get better as dataset size increases. Neural networks have larger representational capacity than linear models and so they 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 5000 training input labeled examples.

When Do I Need Deep Learning?

To close out this chapter we want to leave the reader with a simple set of rules to help triage the discussion of:

> Does this project need to use Deep Learning?

Use Deep Learning When

We'd use deep learning when:

- Simpler models (logistic regression) don't get the accuracy level our use case needs
- We have complex pattern matching in images, nlp, or audio to deal with
- We have high dimensionality data

- We have the dimension of time in our vectors (sequences)

Stick with Traditional Machine Learning When

We'd use a traditional machine learning model when:

- We have high-quality low-dimensional data
  - e.g. columnar data from a database export
- We're not trying to find complex patterns in image data

We'll get 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'll take a look at the tools in the DL4J suite of tools and some real-world examples the practitioner can leverage in their own projects. We'll start out reviewing how we map specific deep networks to the right problem. We'll then review the DL4J suite of tools. The reader will notice more materials for installation and support of DL4J in the Appendix chapters. We'll end the chapter with a deep dive into many of the core examples that come with the library.

Matching Deep Networks to the Right Problem

A theme we introduced in chapter 4 was how Deep Learning is about architecting the network architecture to match the problem as opposed to hand engineering features in the input data. In this chapter we'll see examples of deep networks matched to specific types of problems. For the purposes of this chapter we'll call out applications specifically of:

- modeling columnar data
- modeling image data
- modeling sequence / timeseries data
- natural language processing applications

The applications in this chapter will illustrate the concepts of deep networks we've been building since chapter 1. While we don't have an example of every architecture referenced in Chapter 4 we compiled a set of examples to illuminate core concepts in Deep Learning such that the reader could easily extend most any example towards a new purpose. Let's start out by reviewing the data types in the context of the correct network architecture pairings as introduced in previously in the book.

Finding the Examples Online

We've created a fork of the examples github repository at:

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

While a project such as DL4J will evolve over time we wanted to provide the readers with a snapshot of the code that will always work with this version of the book.

Columnar Data and Multi-Layer Perceptrons

Common columnar data has a static structure to it and is best modeled in DL4J by a classic Multi-Layer 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 Multi-Layer Perceptrons. We'll cover a variety of techniques to help guide the reader with hyperparameter selection in chapter 6.

Images and Convolutional Neural Networks

Convolutional Networks have shown to be adept at finding structure in raw image data. Historically the image modeling field has been dominated by excessive amounts of pre-processing techniques to get the input images aligned and transformed into a form modeling techniques could better handle. Slight variations in rotation or scale made image processing a hard task. Convolutional Neural Networks 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 hand-written digits later with Convolutional Neural Networks later on in this chapter.

Convolutional Neural Networks and Raw Image Data

Columnar data, through pain-staking 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 and provide non-trivial challenges in the feature extraction phase for classical machine learning techniques. A major strength of Convolutional Neural Networks is their ability to take raw image data where the objects in the scene are in arbitrary poses and be able to adeptly recognize the features.

Evolution in the Applications of Convolutional Networks

Applications of Convolutional Neural Networks continue to evolve at breakneck speed. While we recommend the practitioner start with Convolutional Networks for image modeling problems, applications of Convolutional Networks have begun to appear for text modeling problems as well. Some of these applications include:

- Machine Translation[1]
- Sentence Classification[2]
- Sentiment Analysis[3]

In the same way that we slide windows over regions of an image Convolutional Networks can leverage the same concept to slide windows over sequences of characters. This feature location invariance aspect of Convolutional Networks make them a powerful tool in the deep network toolbox. In this chapter we focus on an example of applying Convolutional Network to image data. The reader should take this not as a limitation of the network architecture, but a fundamental starting point to be investigated further.

Timeseries 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 then we'd consider this data to be timeseries data.

1 https://arxiv.org/abs/1610.10099

2 http://arxiv.org/abs/1408.5882

3 http://www.aclweb.org/anthology/C14-1008

Timeseries and sequential data represent a waveform when graphed on a two-dimensional plot. Many times we are looking for areas of the graph that make particular patterns over time. This is in contrast to how image data does not have a time domain but is arranged in two-dimensional grid to form a picture. However, in both timeseries and images, we are looking for specific objects that may occur in the data. These objects may be scaled in size and usually do not appear in the same place every time in the data providing us with a challenege.

Recurrent Neural Networks have evolved from Multi-Layer Perceptrons to better model the time domain for timeseries data. As we saw in chapter 4, Recurrent Neural Networks are better able to model the time domain by allowing for 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 over time.

Recurrent Neural Networks can be used for Classification, Regression, and generating novel output. We'll see examples of both classification and novel generation later on in this 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 (and others). In the world of sequential data, we have not seen this to be generally a good path outside of specific situations.

The argument goes that we can't apply random forests directly to time series / sequential data in the same way that we can with Recurrent Neural Networks. We'd have to convert the time series into some kind of flat vector representation[4] (e.g., extract features) first before any modeling was performed.

Typically we've found the debate where 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 they find all of the tricks in a particular dataset (given a long enough timeline).

4 An example approach would be: http://www.cs.ucr.edu/~eamonn/iSAX.pdf

The real advantages of deep learning for time series (and really any problem) basically amount to the following:

If we go the "tailored features/algorithm" approach, then we find ourself in a situation where we need to write new code for a dozen completely different algorithms/models and then have apply all of them to each new problem/data set. 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 right features, with current state of the art in Deep Learning it's easier to semi-automate hyperparameter tuning of neural networks than it is to semi-automate feature engineering. A well-engineered deep learning system (including hyperparameter tuning) can be efficiently re-used 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 where Recurrent Neural Networks are especially warranted. If the input is a fixed length sequence (i.e., all time series are same lengths), then the practitioner may as well use an Multi-Layer Perceptron or Convolutional Neural Network.

However, if we want to allow arbitrarily long sequence inputs that can (hypothetically) take the full input data history into account, then a Recurrent Neural Network (or a carefully crafted Markov model with a large hidden state space) is basically our only option.

Using Hybrid Networks

When we see data that is the combination of both time and image (e.g. "video"), we'd use a special hybrid network of LSTM and Convolutional layers. We recommend this hybrid as 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:

- 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 in the past decade. The DL4J community also has paid attention to needs beyond basic modeling to provide better integration with platforms such has Spark and 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 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 setup the tools on your laptop.

Vectorization and DataVec

Since neural networks can only train 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 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.

Deeplearning4j's Java API allows programmers to configure their own extensible neural networks within a composable framework, adjusting hyperparameters as needed. They provide 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 computations in the world of Deep Learning. ND4J provides faster linear algebra execution in 3 key ways:

- Use of JavaCPP
- CPU Backends
- GPU Backends

We provide a few notes below on how improvements in these key areas help ND4J accelerate training deep models.

JavaCPP.

- Auto generate 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 support as soon as it comes out.
- Leverages cuDNN as well

Setting Up GPU Support

Check out the guide on setting up GPUs with DL4J in the Appendix for more detailed information.

Benchmarking ND4J and DL4J

To see the current performance of DL4J compared to other major Deep Learning libraries check out the github project:

https://github.com/deeplearning4j/dl4j-benchmark

Today DL4J is competitive with the major libraries as seen in the table below:

Table 5-1. Lenet Example w/ cuDNN

Package	CPU	GPU	Multi	Accuracy
Dl4j	20m08s	3m13s	1m18s	~99.0%

Package	CPU	GPU	Multi	Accuracy
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 topics in the appendix or the DL4J online documentation.

Loading and Saving Models

In this section we'll highlight a few common things the reader may 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.

```
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.

```
Path modelPath = new Path( hdfsPathToSaveModel );
BufferedOutputStream stream = new BufferedOutputStream( os );
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

HDFS file paths 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 seen in the code snippet below.

```
InputStream stream = ...
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

Reading From HDFS. When we'd like to load the model directly from HDFS we'd use the same API call yet we'd create the InputStream with the Hadoop FileSystem class as seen below.

```
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 together 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 the Appendix on DataVec and the Appendix on ND4J.

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 up into mini-batches.

Loading Data on Spark

Spark data loading patterns are similar but use special classes to load data on Spark and HDFS. We'll cover those specifically in chapter 9 on Spark.

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.

Setting Up Model Architecture

Each neural network we model with DL4J requires us to configure a specific architecture. We've discussed the major architecture previously in this book and these concepts match up directly to how we setup 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 seen in the example code snippet below.

```
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.

```
                .learningRate(learningRate)
                .updater(Updater.NESTEROVS).momentum(0.9)
```

In the code snippet above we can see hyperparameters such as the learning rate, an updater, and a momentum value being set for a given MultiLayerConfiguration object. This allows the practitioner 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

Once we have our network architecture setup 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.

We call the .fit(...) method on our model to train our model with the input DataSet.

```
model.fit( trainingDataIter );
```

This method will loop though all of the training records in the dataset referenced by the iterator and return once the network has completed the loop.

Each training epoch is considered a full pass over the input dataset. In the code snippet below 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 this chapter.

Making a Prediction

To better understand making predictions the reader should check out the Appendix on using ND4J's API.

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 Multi-Layer Perceptron Networks

Getting started with DL4J can seem complex for the new Deep Learning practitioner so we'll start out with building a Multi-Layer Perceptron model with DL4J. This will allow the reader to see basic usage of the DL4J API in the context of an older neural network architecture many will recognize. In the Github repo accompanying the book we have java example modeling a synthetic non-linear dataset[5] called "saturn". Below we see the code from the example[6]:

Example 5-1. Multi-Layer Perceptron Example

```
public class MLPClassifierSaturn {


    public static void main(String[] args) throws Exception {
        Nd4j.ENFORCE_NUMERICAL_STABILITY = true;
        int batchSize = 50;
        int seed = 123;
```

5 This is a dataset generated by Dr. Jason Baldridge to test neural network frameworks for basic functionality. http://www.jasonbaldridge.com/

6 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/classification/MLPClassifierSaturn.java

```
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").get
final String filenameTest  = new ClassPathResource("/classification/saturn_data_eval.csv").getFi

//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 lables = t.getLabels();
    INDArray predicted = model.output(features,false);

    eval.eval(lables, 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 bac
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,


//Get test data, run the test data through the network to generate predictions, and plot those p
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, predictionsA

        System.out.println("****************Example finished********************");
    }

}
```

In the following sections we example how the different parts of the code work together to model the "Saturn" dataset.

Setting Up Input Data

The input data is a non-linear dataset of the 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 with a 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 which column the label is represented by.

Determining Network Architecture

We know we want a basic Multi-Layer Perceptron and we want to set this up (and any DL4J network architecture with the MultiLayerConfiguration object as we see below:

```
 //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 2 layers:

- DenseLayer
- OutputLayer

Let's look at the specific sections of the network architecture.

General Hyperparameters

We set our optimization algorithm with the parameter:

```
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

This obviously let's DL4J know we intend to use the Stochastic Gradient Descent optimization algorithm. The learning rate is set simply with the method:

```
.learningRate(learningRate)
```

We're also telling DL4J we intend to use Nesterov's parameter updating strategy:

```
.updater(Updater.NESTEROVS).momentum(0.9)
```

and the momentum factor will be set as 0.9 as shown above.

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 on 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 th 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 numHiddenNodes. For this layer we're also initializing the weights specifically with 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. As the reader will recall from previous chapters, we use the softmax output layer for multiple labels. In this problem we're building a binary classifier and could also use a sigmoid output activation function as well.

```
.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 2 as 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 2 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 sidebar below.

We're also specifying our output layer to use the Negative Log Likelihood loss function here as 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 where only a single column's value in the vector can have the value 1.0. All of the other column value's will be 0.0. This method of vector representation is often used to represent categorical integer features using a "1 of K" scheme. In the table below we see 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	00000001

Value	Binary Encoding	One-Hot Representation
1	001	00000010
2	010	00000100
3	011	00001000
4	100	00010000

Training the Model

To train the model we setup 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 as 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:

```
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 our model has gotten close to learning the training data.

Evaluating the Model

The section of code dealing with evaluating our new Multi-Layer Perceptron model is listed below. 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());
```

As the reader will recall from chapter 1 we introduce the concept of the F1 score and other evaluation-related metrics. The eval.stats() method call will result in the console output as seen below:

```
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 Hand-Written Images with Convolutional Neural Networks

As we discussed at the start of this chapter Convolutional Neural Networks are good at image classification. In this example we'll show the reader how to load and model images of hand-written digits. The resultant Convolutional Neural Network model will be able to classify new hand-written digit images that it has not seen before.

In this example we'll setup 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 is the MNIST hand-written image dataset[7] for this example.

The model architecture is different than the previous example in that it has different layers and parameters as we'll see below. The reader will remember our note previ-

7 http://yann.lecun.com/exdb/mnist/

ously about focusing on model architecture. This particular Convolutional Neural Network is known as the "LeNet"

LeNet

The LeNet Convolutional architecture is focused on a series of convolutional layers followed by max-pooling layers. Below we see this Convolutional Network Architecture implemented in DL4J.

The citation for the original paper[8] is:

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998d). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324.

Java Code Listing for LeNet Convolutional Network

The code[9] for the MNIST LeNet Convolutional Neural Network example is below.

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()
```

8 http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

9 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/convolution/LenetMnistExample.java

```
        .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).lrPolicy
        .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 o
                .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/sub
    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)
    layer based on the size of the previous layer (but it won't override values manually set by
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,
```

```
        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 below we'll 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 due to the fact that the MNIST dataset is in a custom binary format (e.g. the files are not a set of individual JPG or PNG files in a directory as the reader might normally expect). To make this example simpler the reader can just trust the necessary functions are being performed behind the scenes and the raw image data is being extracted into an NDArray for training with DL4J.

```
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);
```

As we can see above, 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 as we saw in the previous Multi-Layer Perceptron example. However, the reader will notice that this network has many more layers and the types of layers are different than in the Multi-Layer 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).lrPolicyPowe
        .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 fi
                .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/subsamp
    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) valu
    layer based on the size of the previous layer (but it won't override values manually set by the
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
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 below.

```
        .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).lrPolicyPowe
        .weightInit(WeightInit.XAVIER)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .updater(Updater.NESTEROVS).momentum(0.9)
```

The following sections comment on how these hyperparameters are set and what we've set their values to.

Regularization. Here we see regularization turned on (set to "true") with a 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 Stochastic Gradient Descent. The reader will commonly see Stochastic Gradient Descent in many Deep Learning examples as it tends to work well in many cases. We'll discuss variations in optimization algorithms in later chapters on tuning in this book.

Updater. We've chosen Nesterov's as the updater for this example. We see Nesterov's used a lot in our examples as 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 are going down a smooth but very shallow hill. The direction is clear, but we might want to take larger steps.

Convolution Layers

Above we see a general pattern of convolutional layers and then max-pooling layers as described in the architecture chapter on Convolutional Networks. Below 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
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
```

Here we can see the java builder pattern setting up a convolution layer properties. We discuss these properties below.

Filter Size. Here we are creating a filter size for this layer with a 5x5 dimension.

Input Data Channels. For this example the number of channels was set to 1 as the custom dataset iterator converts the raw image automatically into a black and white image. Other examples may often have 3 channels of input representing the R, G, and B channels in the image data.

Stride. The stride for this layer is set to (1, 1) such that it only takes a single step over and then down as it slides the filter across the input volume.

Activation Function. The identity function is used as the output of this convolution layer.

Notice the Identity Activation Function?

The LeNet network architecture[10] (1998) predates ReLUs[11] (2012) by quite some years. Changing convolutional layers over to use the ReLUs was show to accelerate the convergence of stochastic gradient descent. For historical purposes we are keeping the original LeNet architecture intact in this example.

10 http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

11 http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf

Convolution Layers and Activation Functions

In more recent Convolutional Neural Networks we often we see a convolution layer setup with a Rectified Linear (ReLU) activation function.

Max-Pooling Layers

Below we 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())
```

We briefly discuss the layer properties below.

Max Pooling. This layer is setup as a max-pooling layer. This pooling layer has a filter size of (2,2) meaning we're down-sampling from a 5x5 filter in the previous layer to a 2x2 grid in the current layer.

Setting the Pooling Layer Stride. The filter stride is set as (2,2) as it will steps of 2 values as it "slides" horizontally and a step of 2 values when it steps down to the next "row".

Output Layer

Since 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 as we can see in the highlighted code below:

```
.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 Convolutional Network

Now that we have a LeNet convolutional model architecture setup we can intialize the MultiLayerNetwork object to train on out input datasets. The code snippet below shows how we take the configuration from the previous section and setup the network.

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

With the model setup we now can train on our input MNIST dataset for the desired number of epochs as we see below in the code sample.

```
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 for 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 with Recurrent Neural Networks

In this section we show off the generative and discriminative capabilities of Recurrent Neural Networks with the generating Shakespeare and classifying timeseries examples, respectively.

Generating Shakespeare with LSTMs

We start off the examples on Recurrent Neural Networks with a fun example of modeling the works of Shakespeare. Here we train a LSTM Recurrent Neural Network to generate new lines of Shakespeare (well, "Shakespeare sounding"). The reader will note that we're treating text as sequence of characters and our model is predicting the next most likely character based off the past characters encountered. This model can also be adapted 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"[12].

Source Text for Training LSTMs

This example is set up to train on the Complete Works of William Shakespeare, downloaded from Project Gutenberg[13]. Training on other text sources should be relatively easy to implement.

High-Level Modeling Workflow

In this example we're going to introduce the concept of setting up a Recurrent Neural Network architecture in DL4J. We're going to build off some of the concepts we've learned in previous examples such as:

- 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 shakes character by character. We then ask the model, progressively, to generate a new line of shakespeare based on the model it has learned.

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 fojlond thivh uweet.
   Thy lametom theuegfast lart souclalitoloe ilntangylrt or

----- Sample 1 -----
l,, ne agly
   Lot Bolncanbom bavantenfircasle womlidibl.
  NTERIOO. IrdmisfUoItolleeeddortiss hot buye.
   The hetenle of ile,
   'merlliydingiponI, bomgule? Shurtstarer of ate,
```

12 http://karpathy.github.io/2015/05/21/rnn-effectiveness/

13 The Complete Works of William Shakespeare, 5.3MB file in UTF-8 Encoding, ~5.4 million characters: https://www.gutenberg.org/ebooks/100

```
    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 now take a look at the java code for this example.

Java Code for Modeling Shakespeare

In this section we list the full code for the example[14] where we model the works of Shakespeare with a Recurrent Neural Network.

Example 5-3. Modeling and Generating Shakespeare with 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. T
        int tbpttLength = 50;                       //Length for truncated backpropagation through time.
        int numEpochs = 1;                          //Total number of training epochs
        int generateSamplesEveryNMinibatches = 10;  //How frequently to generate samples from the networ
        int nSamplesToGenerate = 4;                 //Number of samples to generate after each training
        int nCharactersToSample = 300;              //Length of each sample to generate
        String generationInitialization = null;     //Optional character initialization; a random charac
        // Above is Used to 'prime' the LSTM with a character sequence to continue/complete.
```

14 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/character/GravesLSTMCharModellingExample.java

```
// Initialization characters must all be in CharacterIterator.getMinimalCharacterSet() by defaul
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)
            .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength).tBPTTBackwardLengt
    .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 " + miniBa
```

```
                System.out.println("Sampling characters from network given initialization \"" + (gen
                String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,rng
                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 retu
 * 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
    //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 probl

    char[] validCharacters = CharacterIterator.getMinimalCharacterSet();    //Which characters are a
    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. Initializ
 * can be used to 'prime' the RNN with a sequence you want to extend/continue.<br>
 * Note that the initalization is used for all samples
 * @param initialization String, may be null. If null, select a random character as initialization f
 * @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 characte
    //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 sample
    //Sampling is done in parallel here
    net.rnnClearPreviousState();
    INDArray output = net.rnnTimeStep(initializationInput);
    output = output.tensorAlongDimension(output.size(2)-1,1,0); //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 generat
        for( int s=0; s<numSamples; s++ ){
            double[] outputProbDistribution = new double[iter.totalOutcomes()];
            for( int j=0; j<outputProbDistribution.length; j++ ) outputProbDistribution[j] = output.
            int sampledCharacterIdx = sampleFromDistribution(outputProbDistribution,rng);

            nextInput.putScalar(new int[]{s,sampledCharacterIdx}, 1.0f);        //Prepare next time
            sb[s].append(iter.convertIndexToCharacter(sampledCharacterIdx));    //Add sampled charac
        }

        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="+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. We can see this in the snippet below:

```
CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
```

We'll leave it to the reader to explore how this works under the covers as interest dictates.

Network Architecture of LSTM

Just as in the previous two examples we see 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)
            .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength).tBPTTBackwardLength(tb
    .pretrain(false).backprop(true)
    .build();
```

General Comments on Hyperparameters. In this example we again see Stochastic Gradient Descent 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 and this is different than the other examples we've seen so far.

RMSProp in Practice

RMSProp by design has a normalizing effect on the scale of the updates. This means 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.

Hidden Layers. We see 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 as we are using a special RnnOutputLayer to handle the different types of output that is possible from a Recurrent Neural Network.

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT).activation(Activation.SOFTMAX)
        .nIn(lstmLayerSize).nOut(nOut).build())
```

We also notice that even though again we're using a softmax activation function for the output layer, the loss function used here is LossFunction.MCXENT.

Training the LSTM Network

We train the network as seen in the code snippet below:

```
//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 " + miniBatchS
            System.out.println("Sampling characters from network given initialization \"" + (generat
            String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,rng,nCh
            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 slightly different in this example. As opposed to calling the .fit() method on the iterator itself we here see the .fit() method 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 where 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 above 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 below.

```
String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,rng,nCharactersToSa
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 Timeseries Sequences with LSTMs

In this example we'll take the reader through a sequence classification example using a LSTM Recurrent neural network. We'll use the "Synthetic Control Chart Time Series Data Set" from the UCI machine learning repository[15]. This code example will build a model to classify univariate time series as belonging to one of six categories:

1. Normal (C)
2. Cyclic (B)

15 https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series

3. Increasing trend (E)
4. Decreasing trend (A)
5. Upward shift (D)
6. Downward shift (F)

In the Figure 5-1 below we can see renders[16] of examples from each of these categories:

Figure 5-1. Renders of Synthetic Timeseries Sequences from the UCI Repository

These 6 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 to work with.

16 Image from: https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series

Java Code Listing for Recurrent Classification Example

In the code example[17] below we see the Recurrent classification example java code to build a model to classify the sequences in the "UCI Synthetic Control Chart Time Series Data Set".

Example 5-4. UCI Sequence Classification Example in Java

```
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
        SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
        trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain.getAbsolutePath() + "/%d.cs
        SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
        trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain.getAbsolutePath() + "/%d.csv",

        int miniBatchSize = 10;
        int numLabelClasses = 6;
        DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(trainFeatures, trainLabels,
            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 'trainDa
        trainData.setPreProcessor(normalizer);


        // ----- Load the test data -----
        //Same process as for the training data.
        SequenceRecordReader testFeatures = new CSVSequenceRecordReader();
```

17 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/seqClassification/UCISequenceClassificationExample.java

```
    testFeatures.initialize(new NumberedFileInputSplit(featuresDirTest.getAbsolutePath() + "/%d.csv"
    SequenceRecordReader testLabels = new CSVSequenceRecordReader();
    testLabels.initialize(new NumberedFileInputSplit(labelsDirTest.getAbsolutePath() + "/%d.csv", 0,

    DataSetIterator testData = new SequenceRecordReaderDataSetIterator(testFeatures, testLabels, min
        false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

    testData.setPreProcessor(normalizer);   //Note that we are using the exact same normalization pr


    // ----- 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
            .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) ever


    // ----- 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/sy
        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());
        }
    }
}
```

We look at the various sections of the code below.

Setting Up Input Data and Vectorization

We can see in the code above we don't have to manually go get the data as its done for us. The code will download and prepare the data (in downloadUCIData() method). It splits the 600 timeseries 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 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 is the features using with the labels file train/labels/0.csv. Because the data is a univariate time series, we only have one column in the CSV files. 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 the part of the code to load the training data as shown below, we can see how to use the SequenceRecordReader in DL4J to read the CSV file[18], create the labels, and 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
SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain.getAbsolutePath() + "/%d.csv",
SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain.getAbsolutePath() + "/%d.csv", 0, 4

int miniBatchSize = 10;
int numLabelClasses = 6;
DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(trainFeatures, trainLabels, mini
    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'
trainData.setPreProcessor(normalizer);
```

Towards the end of the 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.

18 For more details on this step, see: http://deeplearning4j.org/usingrnns#data

Network Architecture and Training

Our neural network base architecture is a LSTM (Recurrent) architecture. We can see this being setup with the MultiLayerConfiguration object in the code snippet below.

```
// ----- 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 req
        .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 only need 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. We can see this part of the code in the code snippet below:

```
// ----- 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're then using the Evaluation object to measure how well our Recurrent neural network model generalizes to our held out test data.

Using Autoencoders for Anomaly Detection

To demonstrate practical use of Autoencoders we'll show anomaly detection performed on the MNIST dataset using simple AutoEncoder without pre-training.

The goal is to identify outliers digits, i.e., those digits that are unusual or not like the typical digits. This is accomplished in this example by using reconstruction error: stereotypical examples should have low reconstruction error, whereas outliers should have high reconstruction error

In this example we'll setup the model configuration as in the other examples but our model architecture will be different. For Autoencoders we're using DenseLayers (fully-connected) for most layers but we're focused on an a series of layers that "funnels" down and then "expands" back out to the output layer.

Java Code Listing for Autoencoder Example

Below we have the java code[19] for the Anomaly AutoEncoder example.

Example 5-5. Autoencoder Example Code to Find Anomalies in Java

```
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);
```

19 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/anomalydetection/MNISTAnomalyExample.java

```
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
    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, INDArra
    @Override
```

```
        public int compare(Triple<Double, Integer, INDArray> o1, Triple<Double, Integer, INDArray> o
            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)(imageSc
                ImageIcon scaled = new ImageIcon(imageScaled);
                images.add(new JLabel(scaled));
            }
            return images;
        }
    }
}
```

In the sections below we'll comment on specific parts of the example's code.

Setting Up Input Data

Below we have the code section listed 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 -> i
        labelsTest.add(indexes);
    }
```

We're handling the training and test data differently in this example as opposed to previous examples to 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 with AutoEncoders. AutoEncoders learn to best represent the data, hence their name.

```
//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 4 layers with the last layer having the full 784 units the input layer has for the input data. We've set all of the activation functions in this network as Rectified Linear as we've found it best for working with this dataset. We're using ReLUs as the default activation function per layer as specified globally in this configuration as this works well for Autoencoders.

Training the AutoEncoder network uses the same general pattern as we've seen with the other DL4J examples in this chapter, as seen below.

```
//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 the line:

```
net.fit(data,data);
```

Here we notice that the data is also 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 slightly different 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. Below we see the generated image for the best learned (e.g. had the lowest reconstruction error) hand-written 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 seen below.

Figure 5-3. Hard to Learn Hand-Written Digits

We 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 may look like for our data, but if we base in on a model such as an Autoencoder we have a tool that doesn't have to see the anomalies beforehand.

Using Variational Autoencoders to Reconstruct MNIST Digits

In chapter 3 we introduced the concept of the Variational Autoencoder (VAE) to the reader as a way to reconstruct input data in an unsupervised learning fashion.

VAEs have many uses but the most common use cases would be:

1. unsupervised learning - unsupervised learning of features and semi-supervised learning
   - general idea: large amount of unlabelled data, small amount of labelled data (we can get much better results than using labelled data alone, when labelled data is limited)
2. Anomaly detection (unsupervised)
3. As a generative model:
   - e.g. generate samples
   - Images (as in this example), but also things like sentences[20]

In this example[21] we'll show the reader how to generate variations of the MNIST digit dataset to show the capabilities of VAEs as generative models.

Code Listing to Reconstruct MNIST Digits

This is a simple example of training a variational autoencoder on MNIST to demonstrate the generative capabilities of VAEs. This example intentionally has a small hidden state Z (2 values) for visualization on a 2-grid.

After training, this example plots 2 things:

1. The MNIST digit reconstructions vs. the latent space
2. The latent space values for the MNIST test set, as training progresses (every N minibatches)

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.

20 https://arxiv.org/abs/1511.06349

21 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/unsupervised/variational/VariationalAutoEncoderExample.java

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;
        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 p

        //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(Activation.SIGMOID.g
                .nIn(28 * 28)                       //Input size: 28x28
                .nOut(2)                            //Size of the latent variable space: p(z|x). 2 dimen
                .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();
    INDArray latentSpaceGrid = getLatentSpaceGrid(plotMin, plotMax, plotNumSteps);              //X/

    //Lists to store data for later plotting
    List<INDArray> latentSpaceVsEpoch = new ArrayList<>(nEpochs + 1);
    INDArray latentSpaceValues = vae.activate(testFeatures, false);                     //Collect an
    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, digi
    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)).a
        int yStart = plotSteps - i - 1;
        data.get(NDArrayIndex.interval(yStart * plotSteps, (yStart + 1) * plotSteps), NDArrayIndex.p
    }
    return data;
```

```
    }
}
```

In the following sections we'll explain the generated latent space grid and the generated MNIST images.

Examining the VAE Model

In Figure 5-4 below we see 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 which are derived or inferred through a mathematical model. This contrasts with variables which are observed.

The reader can generate different variations of the images based on how long the training runs and the settings for training.

Figure 5-4. MNIST Digits Generated by the Variational Autoencoder 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. In Figure 5-5 we see the network architecture for the Variational Autoencoder:

Figure 5-5. Variational Autoencoder Network Architecture

the scatter plot Figure 5-6 below of the VAE latent space of the MNIST test data is just the encoder part of the network diagram Figure 5-5 above.

Figure 5-6. Scatter-Plot of VAE Latent Space vs Epoch on MNIST Test Dataset

Understanding the Scatterplot

The scatter plot in Figure 5-6 above is the projection of the variable latent space vs 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 2 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 start with assuming 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 2 values, but using 2 only because we want pretty pictures

Understanding the Generated Images

The generated images in Figure 5-4 above are just the right hand side of the network in Figure 5-6. Specifically, we generate a 16x16 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 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 28x28 size images).

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. Techniques such as Part-of-Speech tagging[22], Character generation[23], and learning word embeddings are common applications of Deep Learning. Some NLP applications we'll focus on in this chapter are:

- Learning Word Embeddeings with Word2Vec[24]
- Distributed Representations of Sentences with Paragraph Vectors[25]
- Document Classification with Paragraph Vectors

We'll look at examples of each of this in the sections below.

22 http://jmlr.org/proceedings/papers/v32/santos14.pdf

23 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/character/GravesLSTMCharModellingExample.java

24 https://arxiv.org/abs/1301.3781

25 https://cs.stanford.edu/~quocle/paragraph_vector.pdf

Learning Word Embeddings with 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, 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 meaning and relationships in the produced word embeddings are encoding spatially and have useful characteristics such as vector arithmetic as we'll see later.

The Word2Vec Model and Algorithm

The algorithm starts 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)[26] to build these contextual word vectors in an unsupervised learning fashion. In practice this is a two layer neural network trained with stochastic gradient descent.

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), Part of Speech (POS) Tagging, and Semantic Role Labeling.

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 merge 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 been shown to preform significantly better than LSA for preserving linear regularities among words and be less computationally expensive than LDA.

26 https://arxiv.org/pdf/1301.3781v3.pdf

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 leverage 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 part of speech tagging, semantic role labeling, and named-entity recognition. 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 previously) or Cosine distance[27]. This gives us words that have short cosine similarity distances or, more generally, "close representations". Below in Table 5-3 we see some examples of words most similar to "france" based on a trained word2vec model:

Table 5-3. Word2Vec Cosine Similarity Example

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

27 https://en.wikipedia.org/wiki/Cosine_similarity

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 Embeddings

We can also do some basic vector operations with the word2vec vectors. An example is:

```
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[28] fashion to find our word vector. When our word vectors are well trained we should find the correct word ("smallest") using this technique. We can also find subtle relationships such as a capital cities of states (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[29], information retrieval[30], and question answering[31] systems. Let's now take a look at as basic example of using word2vec.

28 https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

29 https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44931.pdf

30 http://web.stanford.edu/class/cs276/handouts/lecture20-distributed-representations.pdf

31 https://arxiv.org/abs/1508.01585v2

Java Code Listing for Word2Vec Example

The code example[32] below 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)
                .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
```

32 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/word2vec/Word2VecRawTextExample.java

```
        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);

    }
}
```

Below we'll discuss specific parts of this code to better understand what is happening in the example.

Understanding the Word2Vec Example

In the code snippet below we see how the raw sentences file included with the code example repository is referenced and loaded with 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 o
    Additionally it forces lower case for all tokens.
 */
t.setTokenPreProcessor(new CommonPreprocessor());
```

The preprocessing is performed with the CommonPreprocessor object as specified in the code. 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 as 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();
```

The reader will 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 snippet below:

```
log.info("Fitting Word2Vec model....");
vec.fit();
```

Once the .fit() method completes the example will save the network and print 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 can also use Word2Vec vectors for document classification, as we'll later on in this 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 we sum their word vectors together. This gives a vector that represents the document while including the contextual information included with each word vector. We'd 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 with GloVe

An alternative method to word2vec is called GloVe[33].

GloVe is an unsupervised learning algorithm for building word embeddings and 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[34]. It's worth noting that GloVe is considered to be more difficult to train and tune as compared to word2vec.

33 http://nlp.stanford.edu/projects/glove/

34 http://clic.cimec.unitn.it/marco/publications/acl2014/baroni-etal-countpredict-acl2014.pdf

Dl4J has a GloVe example[35] similar to the above word2vec exercise included in the examples repository[36].

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:

1. we lose the ordering of words
2. 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. Research[37] has shown the paragraph vector method to outperform the bag-of-words method and other text representation approaches. Paragraph vectors have also been shown to improve accuracy on several text classification and sentiment analysis problems.

In this example we'll 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 unsupervised training method as 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 this example[38] we build a distributed representation (e.g. "paragraph vectors") of all sentences present in training corpus. The training data for the example is included in the example repository so the reader need only execute the class once the repository has been cloned and is running locally.

35 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/glove/GloVeExample.java

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

37 https://cs.stanford.edu/~quocle/paragraph_vector.pdf

38 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/paragraphvectors/ParagraphVectorsTextExample.java

Example 5-8. Building Paragraph Vectors with 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 it's own label.
              But if you have LabelAwareIterator ready, you can 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 invloved.
            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 we can see in the results the first 3 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 in the input training data, setup the network, and then train on the data. Again, the file and directory handling is done for the user by 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 it's own label.
      But if you have LabelAwareIterator ready, you can can provide it, for your in-house la
*/
LabelsSource source = new LabelsSource("DOC_");
```

The paragraph vector implementation in DL4J uses a helper class called "ParagraphVectors" as we see below:

```
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 as ParagraphVectors has a more dedicated API (like Word2Vec) as compared to the broader approach other networks use with the MultiLayerConfiguration class.

Once 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 the reader a great feel for how to build paragraph vectors with DL4J. Now let's take a look (in the next example) 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[39] of skip-gram that underlies models such as word2vec, paragraph vectors, and deepwalk[40].

Hinton on Potential Implications of Paragraph Vectors

In a 2015 speech to the Royal Society in London, Geoffrey Hinton said:

> "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.

39 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/sequencevectors/SequenceVectorsTextExample.java

40 https://arxiv.org/abs/1403.6652

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 previously mentioned in the chapter, we can leverage paragraph vectors in the context of NLP and document classification. In this example, we'll leverage a paragraph vectors model similar to the one in the last example in an application to build a document classifier that gives us 1 of 3 labels as output similar to:

```
Document 'health' falls into the following categories:
    health: 0.29721372296220205
    science: 0.011684473733853906
    finance: -0.14755302887323793
```

As we can see in the output snippet above, the label with the highest score is our classification. Let's take a look at the code listing below for our paragraph vectors classification application[41].

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:
```

41 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/paragraphvectors/ParagraphVectorsClassifierExample.java

```
                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 the above example is to use the ParagraphVectors class in the same way we use latent Dirichlet allocation (LDA)[42] (e.g. "topic space modeling").

In this example we assume 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 application of the model leverage the paragraph vectors.

Understanding the Paragraph Vectors Classification Example

This example has some similar mechanics as the last example yet has been extended to classify unlabeled documents. The main driver code is:

```
    public static void main(String[] args) throws Exception {

      ParagraphVectorsClassifierExample app = new ParagraphVectorsClassifierExample();
      app.makeParagraphVectors();
      app.checkUnlabeledData();
```

42 https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

We 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. Inside the .checkUnlabledData() method we can see 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. The reader can clearly see in the above output the document was classified as "health", which is correct as it matches the associated label in the data.

Further Exploration of the Word2Vec Approach

As we mentioned earlier in this chapter about Convolutional Networks, 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:

- extensions into specific domains
- graph analysis
- recommendation
- 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[43]. The description from the author's paper[44]:

> 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 may or may not veto a bill, and how that evolves over time. They are able to answer such questions as:

> 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[45], 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).

Recommendation Engines and Item2Vec. Another interesting application of the concept is Item2Vec where the approach is used for recommendation systems[46]. In this paper the authors show the neural word embedding approach used as an item-based collaborative filtering technique for recommendation.

Computer Vision and FaceNet. In the arena of image identification there is FaceNet[47], where we see an application of neural network embeddings used to generate representation of facial features (images)[48]. 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.

---

43 http://aclweb.org/anthology/W16-5607

44 Nay, J. J. (2016). "Gov2Vec: Learning Distributed Representations of Institutions and Their Legal Text."Proceedings of 2016 Empirical Methods in Natural Language Processing Workshop on Natural Language Processing and Computational Social Science, 49–54, Association for Computational Linguistics.

45 https://arxiv.org/abs/1607.00653

46 https://arxiv.org/abs/1603.04259

47 https://arxiv.org/abs/1503.03832

48 http://crockpotveggies.com/2016/11/05/triplet-embedding-deeplearning4j-facenet.html

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[1]

—

15th-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'll take a look at

- Matching network architecture to the problem we're working on
- The basics of hyperparameters 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 the reader to the core concepts in deep architecture tuning. For chapter 7 we then focus specifically on tuning techniques for the most well known architectures in Deep Networks:

- Deep Belief Networks
- Convolutional Neural Networks

1 https://en.wikipedia.org/wiki/Paracelsus

- Recurrent Neural Networks

Let's start out with a general idea, or intuition, on how we want to approach building different neural networks with different goals.

Tuning Restricted Boltzmann Machines

We'll cover tuning Restricted Boltzmann Machines in the context of tuning Deep Belief Networks.

An Intuition for Building Deep Networks

We should start out by asking ourselves two questions:

1. What kind of input data will I be modeling?
2. What is the output we want to know once this model is constructed?

Understanding what kind of data we're modeling will largely dictate what class of deep learning architecture and input layer we need work with. Determining what we want to know from data of this class will tell us whether we want to get class scores (classify something with probabilities) or produce a real-valued target output (regression). This will also tell us what kind of output layer we need to use. There are variations and caveats in many corners of deep networks but these two questions will help get us started down the path of putting together a coherent start to our deep network architecture. Once we understand these two things, we can move to the next tier of design decisions that focus on setting up the parameters themselves:

- Number of layers
- Parameter count per layer

We can also take into consideration the memory requirements needed for a particular network architecture decision based on parameter counts. 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 a relatively few parameters (total neuron count).

Now that we have an architecture and layers, we want 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. We'll need an activation function for each layer to model non-linear relationships between the input and output data. Activation functions help us learn certain types of features along with setting up our output layers to give us answers like regression or classification. We want to match up certain layer types (activation function types) with certain loss functions in specific architectural situations.

Affecting Learning with Loss Functions

Loss functions are how we define what we want the network to learn (classification or regression, for example), and have to be matched with both an appropriate activation function and the type of data/labels we have.

Depending on our problem space we'll also want to consider different types of optimization method.

The methods of regularization keep our 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 (e.g. "examples not seen during training"). As we'll see in this chapter, many of the above 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 we can use as a guide for network architecture construction.

Building the Intuition as a Step-by-Step Process

In the above section we talked in general about what we need to get done in building a deep network architecture. In the steps below we formalize the process in a guide the practitioner can use for most any neural network modeling problem.

1. Determine what our input data should be
   a. Input data type gives us architecture
2. Determine what our intended result should be
   a. Gives us guidance on configuring architecture
   b. Also gives us the output layer type
3. Setup 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

   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 hyper parameter 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 set of steps to follow when building any deep network. By the end of this chapter we hope the reader 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 start our deep network design process by thinking in terms of what our source dataset represents. Below we'll look at:

- Columnar Data
- Image Data
- Audio Data
- Video Data
- Timeseries Data

For columnar (e.g. "CSV") data typically we're dealing with a RDBMS export of a table or the result denormalized table of multiple sets of data joined together as shown below in the diagram:

```
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 don't have any image pixels to extract features from nor any time dependencies to deal with (timeseries data) so our architecture can be relatively vanilla. Here we recommend the practitioner starts with a simple multi-layer perceptron neural network.

For any image classification tasks we want to use a Convolutional Neural Network. Convolutional Neural Networks have shown in recent years to provide best of class results for image processing tasks for reasons we explained in chapter 4.

Sequential data is any situation where have a series of inputs to our model. This is commonly seen in the wild as log data produced by servers or sensors. Timeseries data also falls into this bucket. We define timeseries data as any data that has a series of values each coupled with a value of time. These values, ordered by time, track an entity's activity over time. In most machine learning models we have to do some sort of feature extraction to produce one single input vector per set of timeseries data for our training algorithm. For any sequential or timeseries data the practitioner should focus on using Recurrent Neural Networks. Recurrent neural networks have the ability, as we saw in chapter 4, to model N number of input vectors and are not restricted to a single input vector. 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 where we see a timeseries of audio samples to produce a waveform. As discussed previously in this book, Recurrent Neural Networks work well on timeseries data and are a good match for audio data modeling problems.

When working with video data we have 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 (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 convolutional neural networks to analyze the individual frames.

Dealing with Video Data

Using optical flow preprocessing helps a lot in practice. That 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 now given the reader an idea where to start for different network architectures given a specific data input type. We include a quick lookup table below for the reader to refer back to:

Table 6-1. Matching Input Type to Network Architecture

Input Data Type	Suggested Architecture
Columnar (e.g. "CSV Data")	Multi-Layer Perceptron
Image	Convolutional Neural Network
Sequential	Recurrent Neural Network (e.g. "LSTM", specifically)
Audio	Recurrent Neural Network (e.g. "LSTM", specifically)

Input Data Type	Suggested Architecture
Video	Hybrid Convolutional + Recurrent Networks

The Revolving Door of Network Architectures

The reader should note that while the suggested architecture for each data type may work well, there is a large amount of research ongoing. New variations of architectures are published at a quick pace these days, so the reader should consider these a "nice starting point" and to also be on the lookout for research extending these architectures.

In the next section we'll further flesh out our architecture heuristic by looking at how to setup layer and neuron counts.

Relating Model Goal and Output Layers

In the previous section we matched our input data type to a specific neural network architecture. The output layer type involves more thought as it is 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're 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 below).

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.

- 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 lin-

ear variant (leaky relu, randomized leaky relu) if data is guaranteed to be in the range [0, ∞)

What about rectified linear (RELU) for regression, where the labels data is in the range [0, ∞)? Though the activation function itself can produce the correct range of output values (i.e., also [0, ∞)), there is another problem to be aware of: the so-called 'dying ReLU' problem.

Essentially, there's a chance that a ReLU unit can get stuck in the zero output section of the activation function. If this occurs, the output for that ReLU unit (e.g., our network prediction) will always be 0, regardless of the network inputs. Other ReLU variants (leaky, 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 vs 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 (e.g. "sports", "business", "politics") it belongs to. There is also the case where the document could belong to multiple categories (e.g. "sports" + "business").

Single Label Classification Models

In the basic single label example (e.g. "binary classifier") we'd want to use a sigmoid activation function for our output layer. This would give us a single output with a value somewhere between 0.0 and 1.0 indicating whether or not the document is indeed spam.

When working with a single output label (where we have a single value on the range of 0.0 to 1.0 for 2 classes) use the cross-entropy loss function for the output layer.

Single Output or or Two Outputs for Binary Classifier?

There is a variation where we might use a softmax activation function with the output layer with 2 outputs for a binary classifier. In 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 for modeling binary classification models. Mathematically using a sigmoid single output is the same as using a

softmax output layer with 2 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:

1. Sometimes we have multiple labels and we want the model to give us the most likely single label
   a. Multiclass Classification[2]
2. Other times we have multiple labels and we want multiple labels per output (e.g. "person + car")
   a. Multi-Label Classification[3]

We look at each scenario below.

Multiclass Classification Models. Recalling our discussion from chapter 2, if we have a multi-class modeling problem yet we only care about the best score across these classes then 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 a 1.0 or exactly a 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 out classification result. We'd use an arg-max() function to get the *index* of the class with the highest score (e.g., the predicted class).

We see part of a neural network setup in the example below 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())
```

The specific output layer section in the example above 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 see in the above code snippet. Sometimes we may see multi-class cross entropy used similarly, but is equivalent to negative log likelihood.

2 https://en.wikipedia.org/wiki/Multiclass_classification

3 https://en.wikipedia.org/wiki/Multi-label_classification

The data representation should be "one-hot" for the output. This would be relevant when building the input/output training vectors as we need to remember to convert the output vector into 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

In the case where the number of intended labels reaches the ten-thousands we'd want to use a hierarchial softmax output layer. Hierarchical softmax[4] is a fast(er) approximation to softmax suitable for a large number of classes[5]. We should consider it for computational reasons and not to improve accuracy.

Multi-Label Classification Models. If we want to get multiple classifications per output (e.g. "person + car") then 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 as we could have multiple label entries as 1.0 whereas one-hot can only have 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 the type (SUV, sport, truck, etc). In this scenario we'd need to use ComputationGraph + multiple output layers:

https://deeplearning4j.org/compgraph#multitask

4 http://www.iro.umontreal.ca/~lisa/pointeurs/hierarchical-nnlm-aistats05.pdf and https://arxiv.org/abs/1310.4546

5 Hierarchial softmax is not the only option. An alternative that solves the same problem is:https://arxiv.org/abs/1511.06909

Quick Reference: Softmax vs Sigmoid Activation Functions

The softmax activation function bounds the sum of the outputs to be 1.0 (e.g. "probabilities") vs 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 may have every unit with 0.9 as the output. Sigmoid layers do not have this "lateral" constraint.

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 can also 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 as more than a DenseLayer in DL4J. An example would be how a DenseLayer has more parameters than a similarly-sized Convolutional Neural Network layer.

As the number of parameters grows the function we can model becomes more complex. After a point the number of parameters gets 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 setup. There are some high level ideas about neural networks that mainly apply to feedforward multi-layer perceptron neural networks. We'll cover the specific architectures later on in this chapter.

Feed-Forward Multi-Layer Networks

For feedforward 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. Below 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 MNIST only

needs around three to four hidden layers (with accuracy decreasing beyond that depth) but Facebook's Deep Face 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 as the data set is larger we can use more hidden layers and neurons without overfitting. With more training data we're less likely to overfit. We can have a larger network and possibly get better accuracy. If the network size is too small with a larger input dataset then 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.

A good way to think about neuron count is 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 worked generally better. This effect may only work on certain types of datasets, however.

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 pre-training (Restricted Boltzmann Machines in Deep Belief Networks) neural networks. An example of this would be a hidden neuron count going from hundreds of units to thousands of units[6]. Instead, to combat overfitting it is suggested to use regularization techniques such as:

- L1
- L2
- Dropout
- Input Noise
- Dropconnect

The reasoning is smaller networks will have few local minima (but possibly bad models) and larger networks will have more local minima[7]. What we need to know here is that 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 for and use the above techniques to combat overfitting.

Don't Go Overboard with Parameters, Though

Throw more parameters at the problem but be pragmatic about it. Having a single layer with 1,000,000 neurons may not be the best idea even though the hardware could handle it.

As the variation and size of the training dataset grows we recommend the practitioner to 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:

1. hand calculating the number for an architecture
2. using the DL4J API to calculate the parameter count

6 Bengio 2012, Practical Recommendations for Gradient-Based Training of Deep Architectures, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.

7 Li, Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes), http://cs231n.stanford.edu/, http://cs231n.github.io/

The number of parameters in a network can be determined by calculating simply adding up the number of parameters in each layer. Here's the number of parameters for some commonly used layers:

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	$d^{L-1}d^{L}k^{H}k^{W} + d^{L}$
LSTM RNN (e.g. "GravesLSTM")	$4n^{L}\left(n^{L-1} + 1\right) + n^{L}\left(4n^{L} + 3\right)$
Standard ("vanilla") RNN	$n^{L-1}n^{L} + \left(n^{L}\right)^{2} + n^{L}$

where

- 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 work out the size of the activations array out 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 activations with 100 channels, this means we should use $n^{L-1} = 5\times5\times100 = 2500$ as the input to the dense/output layer[8].

Suppose we have a Deeplearning4j configuration. How do we check the number of parameters? Here's a simple way.

```
MultiLayerConfiguration configuration = ...
MultiLayerNetwork network = new MultiLayerNetwork(configuration);
network.init();
```

8 For more details on calculating the input sizes, see Stanford's CS231n notes and Deeplearning4j'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).

```
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: " + 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 due to running out of memory. 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 6 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

plus overheads for working memory/temporary arrays, shape buffer information, asynchronously preloaded data, and everything else the JVM needs to run.

Mathematically, the minimum memory requirements for training a neural network (in bytes) can be estimated as:

$$N_{bytes,\,train} = 4\left[n_{params}(2 + u) + m(2a + d)\right]$$

whereas for using the network at test time (after training), we require only

$$N_{bytes,\,test} = 4\left[n_{params} + m(a + d)\right]$$

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, Deeplearning4j (via ND4J) uses 32-bit floating point numbers for all INDArrays. This means an INDArray with *N* elements will consume *4N* bytes. 64-bit floating point 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 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 Deeplearning4j for training on GPUs using CUDA. FP16 format reduces the memory requirements by a factor of 2 compared to the default 32-bit floating point 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 we are new to tuning neural networks or are having trouble tuning our network, stick with the FP32 default, unless we really need it. As for performance: FP16 performance is very poor on most consumer GPUs, hence FP16 it 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 minibatch size *m*
2. Get better hardware (or use a cloud service such as Azure, AWS or Google Cloud)
3. Use FP16 instead of FP32 on GPUs (but: beware the reduced numerical precision)
4. Use a smaller network

Weight Initialization Strategies

Weight initialization can greatly affect the training process. The modeler should take care in matching the right weight initialization strategy with the right 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).

In Most Cases for Initializing Weights

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 have to deal with initializing the bias's of the network units as well. 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 where 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 paper by He et al. (2015) 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 paper by Glorot and Bengio (2010), Understanding the difficulty of training deep feedforward neural networks
  - For Weights connected to TanH units, other initialization methods involve concepts such as "Sparse Initialization"[9].
- For biases, these are typically initialized to 0 (this is the default in DL4J).

9 http://icml2010.haifa.il.ibm.com/papers/458.pdf

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:

1. Ensure the variance of network activations are constant for all layers
   - (e.g., activations don't grow too large or too small in later layers in the network)
2. Ensure that the variance of activation (and hence parameter) gradients are constant for all layers
   - (e.g., 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)[10].

Orthogonal Weight Initialization of RNNs

In the next chapter we'll look at tuning strategies for specific architectures such as LSTMs. It's worth noting here that LSTMs and GRUs have been shown to have better performance in some cases with orthogonal weight initialization[11].

Using Activation Functions

Below in Table 6-3 we include notes for matching data distribution types to some of the activation functions in feed-forward networks. We provide some notes on the major patterns seen in practice and close with a table explaining how to match up each to a network architecture.

10 http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdfpg5, eq12

11 https://arxiv.org/abs/1312.6120v3

Table 6-3. Target Distributions vs Output Layer Activation Functions in Feedforward 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)
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 but today have largely fallen out of favor for using in hidden layers.

Sigmoid and Information Loss

Compared to ReLU activation functions we also see that sigmoid functions more often discard information due to saturation in both forward and back propagation giving non-linear effects to the network due to a single parameter in a small neighborhood[12].

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 doesn't have the vanishing gradient problem of sigmoid/Tanh, nor the "dying ReLU" problem of the 'vanilla' ReLU.

The best practice is "don't 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 Convolutional Neural Networks as the activation function of convolution layers. Stochastic Gradient Descent has been found to speed up learning with ReLU's when

12 http://www.deeplearningbook.org/

compared to sigmoid and Tanh functions. ReLUs are computationally cheaper than Sigmoid and Tanh functions.

The Dying ReLU Problem

We have to be careful when using ReLUs, however, as some units may 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 here for the "dying ReLU problem" is to use the Leaky ReLU variation which (unlike the 'vanilla' ReLU) has non-zero 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 overall parameter count.

Maxout units have been shown to learn in conditions where rectified linear units (ReLU) 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 help them not get stuck in learning in places ReLUs tend to get stuck.

Summary Table for Activation Functions

Function Name	Where We Use It
Linear	Output Layer for Regression
Sigmoid	• Output Layer for Binary Classification • Outputs values on the range [0,1] • Fallen out of favor, not to be used in hidden layers
Tanh	• Continuous data, more than [-1,1] • LSTM Layers

Function Name	Where We Use It
Softmax	Multiclass Classification Model Output Layer
Rectified Linear Units (ReLU)	• Restricted Boltzmann Machines • Convolutional Neural Network Layers • Mutli-Layer Perceptron Network Layers

In general we recommend to explore 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 the practitioner 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 only used 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 its doing at the intended task. Previously in the chapter we referenced the usage of loss functions in the context of model goal (e.g. "output layer loss functions"). Below we list a few notes about other scenarios for loss functions, where to use it most effectively, and any caveats about how it is used. To be direct, loss functions are used only in 2 places:

1. output layers (or, LossLayer)
2. layers that support unsupervised layer-wise pre-training
   - (Autoencoders, RBMs, VAEs).

Most layers (convolutional, dense, LSTM, etc) do not (and can't) have a loss function as they can't be pre-trained 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 has to be optimized for a hard classification. For eg, 0 = no fraud and 1 = fraud which by convention 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 $ number. Predicting valid probabilities means generating numbers between 0 and 1.

Binary Classification and Hinge Loss

The hinge loss is mostly used for binary classifications. There are extensions for multi-class classification (e.g. "one vs all", "one vs 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:

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 start and end at the same levels and is dependent on the loss function used in the network architecture. If we take mean squared error for example, we'd get a different progress score as opposed to say negative log likelihood loss function.

The easiest way to understand this number is: "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));
```

Below we have a summary table for quick reference on where to use each loss function.

Loss Function	Where To Use It	Properties
Reconstruction Entropy	RBMs, AutoEncoders	Used for feature engineering
Squared Loss	Output Layer	Regression
Cross Entropy	Output Layer	Binary Classification
Multi-Class Cross Entropy	Output Layer	Classification
Root Mean Squared Error	AutoEncoder, Restricted Boltzmann Machines, Output Layer	Feature Engineering, Regression
Hinge Loss	Output Layer	Classification
Negative Log Likelihood	Output Layer	Classification

Cross Entropy, Logistic Loss, and Negative Log Likelihood

The reader may see these loss functions used in similar manners across literature and examples. As noted back 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 while it may 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. Stochastic Gradient Descent and Learning Rate

Above in Figure 6-1 we can see how too large (left side) and our network training will likely can be unstable (or diverge entirely). If our learning rate is too small (right side) then training can take orders of magnitude longer than it needs to.

We ideally want to see a large learning rate in the beginning of training and then have the learning rate 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 the reader is using an updater such as Adam or RMSProp, the momentum schedules are not used.

A Note About Increasing Learning Rate Setting

Increasing the learning rate initial setting is not always helpful because while we may initially learn well, our algorithm will learn more slowly in the long term.

Ideally we want to use some type of non-static learning rate scheme where learning rates may be set on multiple tiers:

- Globally
- Per layer
- Per neuron
- Per Parameter
  - (where we have more parameters per neuron, bias, etc)[13]

We want the learning rate drop to only start to drop near the end of learning. We want a learning rate schedule that decays over time as if we can 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 Parameters to Updates

A simple and effective approach for setting the learning rate is to use the ratio of parameters to updates (or specifically, the mean magnitudes there-of). Recall that the updates are what we have after we apply our updater (RMSProp, momentum, etc - and learning rate) to the gradients - i.e., 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, then the ratio of updates to parameters is given by:

$$update{:}\,parameterratio = \frac{\frac{1}{N}\Sigma_{n=1}^{N}\left|u_i\right|}{\frac{1}{N}\Sigma_{n=1}^{N}\left|\theta_i\right|}$$

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 (bottom 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 below in Figure 6-2:

13 Bengio 2012, Practical Recommendations for Gradient-Based Training of Deep Architectures, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.

Figure 6-2. DL4J's Tuning UI Showing the Update:Parameter Ratios

When using DL4J, to see the above screen once the network starts training we can go to http://localhost:9000/ to view the training progress.

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 log10 ratio = –3 in Deeplearning4j'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 that we should decrease our learning rate[14].

Specific Recommendations for Learning Rates

Learning rate settings tend to start with a larger setting and fall over time through a decay mechanic. If the learning rate starts 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.

14 Other measures, such the ratio of L2 norms could also be used for this purpose: these aren't available in DL4J's training UI, but could be calculated manually.

Good Starting Learning Rates

Ideally we start 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 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 start.

We suggest trying learning rate methods in the following order:

1. Adam[15]
   - currently recommended as the default method to use
2. Nesterov Momentum
3. RMSProp
   - Uses moving average of squared gradients
   - Doesn't have issue where learning rates get monotonically smaller as in Adagrad
4. Adagrad

Other supported options in DL4J include:

- Adadelta

We Still Have to Watch Learning Rates

If we set the learning rate too high, and 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 mononotically decreasing[16].

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 built in so users can worry less about hand tuning the learning rate.

15 http://cs231n.github.io/neural-networks-3/

16 http://cs231n.github.io/neural-networks-3/#ada

Working with Momentum/Nesterovs in DL4J

Momentum helps the learning process escape these local minima and find better solutions to the optimization process[17].

We typically see momentum values around 0.9. Other common values near this setting that are found to be commonly used are: [0.5, 0.9, 0.95, 0.99]. A typical setting is to start the momentum value at 0.5 and 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 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 mechanics can make tuning the learning rate a challenge.

Other Things to keep in mind:

- Learning rates are network specific: what works in one network may 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 initalization 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 start 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 to 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.

17 Sutskever 2013, Training Recurrent Neural Networks (PhD Thesis), http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf

— Adam and RMSProp updaters may help to 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[18].

Sparsity helps training get 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 increase our sparsity factor should increase. How sparsity interplays with network size will also 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 stochastic gradient descent. Stochastic gradient descent is relatively easy to implement yet it remains challenging to tune and parallelize.

18 Le, Jaitly, Hinton 2015, A Simple Way to Initialize Recurrent Networks of Rectified Linear Units, http://arxiv.org/abs/1504.00941

Some researchers have made the argument that adding additional parameters to neural networks fail to reduce under-fitting on large datasets and waste capacity. These researchers have pointed to SGD as the cause and suggested using second-order methods to mitigate these issues, including:

- BFGS (L-BFGS)
- Conjugate Gradient (CG)
- Hessian Free

Below we have a quick reference table on best practices for initial optimization method for each network architecture:

Network	Common Training Methods and Notes
Deep Belief Networks	Stochastic Gradient Descent
Convolutional Networks	Stochastic Gradient Descent (+Dropout)
Recurrent Networks	Stochastic Gradient Descent, Hessian Free

We'd like to make 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 set the batch size to the full dataset. When dealing with larger datasets we suggest using Stochastic Gradient Descent with a mini-batch setting. The practitioner could also try a second-order method here with a mini-batch setting lower than the full dataset size.

Below is a table with a comparison synopsis of each method.

Method	1st order info	2nd order info	Pros	Cons
Stochastic Gradient Descent	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 gradient descent, higher cost per parameter update, higher memory costs
Conjugate Gradient	Gradient	curvature information is estimated from the gradients	.	higher cost per parameter update, higher memory costs

Stochastic 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

While we list the table above as a quick comparison, for the most part the reader should focus on starting out with Stochastic Gradient Descent. In the following section, we dive into specific best practices for Stochastic Gradient Descent.

Stochastic Gradient Descent Best Practices

Stochastic Gradient Descent works well with momentum and a well-designed random weight initialization scheme. It has been shown effective at training both Deep Belief Networks and Recurrent Neural Networks[19] to levels on par with hessian-free optimization. Researchers have seen that a carefully tuned stochastic gradient descent method is hard to beat on large classification problems[20].

A quick summary of the best ways to start out training with stochastic gradient descent in deep learning:

- Use a good shuffle method on the input training set[21]
- 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, RMSProp
- Set learning rate appropriately: too large or small is bad
- Normalize the input data (not specific to SGD, but worth repeating as it's often forgotten)

Smaller problems with real-valued outputs (function approximation, control problems, etc) we see conjugate gradient offers great performance. With the second order methods we typically see larger mini-batch sizes (10,000) than with the SGD methods.

19 Sutskever et al. 2013, On the importance of initialization and momentum in deep learning, http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf

20 LeCun et al. 1998, Efficient BackProp, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.

21 Bottou 2012, Stochastic Gradient Descent Tricks, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.

Stochastic Gradient Descent as Most Frequent Starting Place

In practice people seems to use Stochastic Gradient Descent + (momentum or adagrad or RMSProp etc) more frequently than second order methods.

Leveraging Parallelization and GPUs for Faster Training

As our machine learning application goals get more ambitious we need larger models to accommodate them. This means we have 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). In the context of a single machine we are at the mercy of 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 quick hit the limits of sequential learning in modern machine learning activities (Zinkevich, 2011)[22]. Bandwidth of storage and network per computer has not been able to keep up with the increase in data. We have to begin thinking about data analysis algorithms which are able to perform most steps in a distributed fashion.

Touching Disk at Scale

At current disk bandwidth and capacity (2TB at 100MB/s throughput) it should take approximately 6 hours to read the content of a single HD

Online Learning and Parallel Iterative Algorithms

The move from batch learning algorithms to online learning algorithms have been able to deal with increasing data set sizes for over a decade now. However, whenever we have more than a single disk 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 looking for methods to parallelize execution of programs.

We can see recent work where traditionally serial learning processes have been redesigned for parallel computations. This is accomplished by have a core perform the local computations and then combining the work from several cores centrally to produce a global result (Dean and Ghemawat, 2004[23]). As our dataset exceeds the capaci-

22 http://martin.zinkevich.org/publications/nips2010.pdf

23 https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

ties of a few disks we begin to rethink how we architect 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 data parallelism and task parallelism.

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 up 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 Stochastic Gradient Descent we can use local thread-parallelism effectively with the parameter-averaging technique[24]. 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 "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 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 its used far and wide. The authors of this book define the term "big data" in the practical sense that the data is

24 https://arxiv.org/abs/1110.4198, http://hunch.net/~vw/, and http://www.cslu.ogi.edu/~bedricks/courses/cs506-pslc/articles/week3/dpercep.pdf

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.

How Long Does it Take to Scan a Petabyte?

A great example of the constraint of big data is that moving data becomes a non-starter 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 disk (around 40MB/s). This basic scan/copy operation will take around 310 days on a single processing unit at 40MB/s.

Apache Hadoop[25] is a great place to store and manage a large amount of data such as web logs or sensor readings as it has been shown in enterprises around the world to be Petabyte-scale. The traditional parallel framework that was originally included in Hadoop distributions[26] was the MapReduce framework. MapReduce parallelization is based around leveraging 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 where 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.

25 http://hadoop.apache.org/

26 http://www.cloudera.com, and http://hortonworks.com/

Issues With Iterative Algorithms and MapReduce

If we wanted to make 100 passes over the dataset and our Map-Reduce scheduling cost is 30 seconds per dataset pass, this gives us an overhead of (100 x 30) 3000 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 candidate 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 more iterative methods of parallelism we're seeing many forms of parallel speedup. Stochastic Gradient Descent has been the focus on many research papers[27] that have parallelized SGD without using the Map-Reduce framework. We also see the evolution of open source frameworks (Vowpal Wabbit[28], Apache Spark[29]) for the parallelization of iterative methods where parameter averaging is a central theme in the implementation.

Parallelizing Stochastic Gradient Descent in DL4J

Another publication[30] out of Google describes such a parallel system for iterative class algorithms such as parallel stochastic gradient descent. This system is called Downpour SGD and leverages 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 seen below in Figure 6-3:

27 http://cs.stanford.edu/~jngiam/papers/LeNgiamCoatesLahiriProchnowNg2011.pdf

28 http://hunch.net/~vw/

29 http://spark.apache.org/

30 https://research.google.com/archive/large_deep_networks_nips2012.html

Figure 6-3. DL4J's Parameter Averaging Parallelization Strategy

One of the major research papers that inspired DL4J's parallelization strategy was the work by Jeff Dean and team at google on their Sandblaster tool.

Who is Jeff Dean?

Jeff Dean is an distinguished research engineer at Google where he worked on some of the core systems. Jeff joined Google in mid-1999 and and is a Google Senior Fellow in the Knowledge Group. As we mentioned above Jeff led the design and implementation of DistBelief and SandBlaster (ICML'12). 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, and other machine learning tasks. Some of the other influential machine learning and distributed systems projects he's led include

- five generations of our 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 a documentation for other developers.

DL4J supports many possible runtimes (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 above 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 Stochastic Gradient Descent Execution

A high level execution flow of this pattern is as follows. Each worker 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 they can perform the next mini-batch inside 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 the section on distributed systems in the appendix for diagrams and more information).

We can illustrate this with an example of training in parallel. Say 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 imple-

mentations show that the speed improvements are close to linear in the number of machines.

Parallel training may see different types of training effects in practice than serial training.

Mini-Batching. 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 harder 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 leverage 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 leverage 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.

GPUs: Lots of Cores

Modern GPUs have > 1000 CUDA cores[31]

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

31 https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#GeForce_10_series

sense in this context, we see that in most cases this transfer time dominates even large matrix x matrix computations on the GPU (example: matrix operation taking only 0.5% of the time in a 1000x1000 matrix multiplication). We can get 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 the appendix on "Jeff Dean's 12 numbers", we want to batch memory operations when we touch hardware. Once we have an efficient transfer of data into the GPU hardware we can then leverage the thousands of extra threads operating on separate blocks of data inside the training batch sent to the GPU at one time. These factors are the major driver 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 in mind that we'd need to operate on CPUs, GPUs, or on a parallel framework. The practitioner will find considerable flexibility working with swappable ND4J backends, giving us options on execution.

The reader will find an appendix on setting up GPUs as a ND4J backend in the back of the book.

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.

Let's lay out a few 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 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 at a time to train on.

The relationship between how fast our algorithm can learn the model is typically U-shaped (batch size vs training speed). This means that initially as the batch size gets larger the training time will decrease. Eventually we'll see the training time begin to increase once we get beyond a certain batch size that is too large.

As Mini-batch Size Increases

As mini-batch size increases, more computation means gradients may 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 Tradeoffs

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:

- 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 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[32], or multiples of 16, 8, 4 or 2 if multiples of 32 can't be used (or results 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 2, compared to other sizes.

32 https://svail.github.io/rnn_perf/

Powers of Two

We should also consider powers of 2 when setting our layer sizes. For example, we should use a layer size of 128 over size 125, or size 256 over size 250, etc.

With regard to *optimization efficiency*, it's sufficient to note that the mini-batch size cannot be chosen totally in isolation to 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) may allow for faster learning for a given number of parameter updates. The tradeoff 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 data sets.

So, what mini-batch size should we choose? In practice, 32 to 256 is common for CPU training, and 32 to 1024 is common for GPU training. Usually something in this range is 'good enough' for smaller networks, though should probably be tested for larger ones (where training time can be prohibitive).

However, memory requirements may limit the maximum batch size for large networks anyway.

A Note About Mini-Batch and Distributed Training

For distributed training, it's not uncommon to use smaller mini-batch sizes per executor on shared hardware (such when training on Spark with multiple executors per machine).

Finally, don't forget: 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 2, we need to increase the number of epochs by a factor of 2 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 vector 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 regularizaton 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 keep our parameter values from become too large. Finding the right combination of settings for regularization is typically done with manual tuning but can also be accomplished with "random search" or "grid search". Below we'll give a short summary of each regularization method and how it effects 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 combined[33] which is common for deep networks. The table below shows how we use a different prior function depending on what type of input data we're modeling.

Prior Function Name	How It's Used
L1	Sparse Models
L2	Dense Models

In literature we see both L1 and L2 regularizers used at the same time with separate settings for each function. In cases where we're using early-stopping we may want to not use L2 regularization at all as early stopping is more efficient at performing the same mechanic as L2[34]. L1 regularization has been consistently shown to act as a useful form of feature selection.

In practice, L2 regularization is more frequently used. Some notes comparing and contrasting the two forms of regularization:

- 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 the resultant weight vector can be sparse.

Both L1 and L2 can be combined in the one network.

33 "elastic net", Zou and Hastie 2005, http://users.stat.umn.edu/~zouxx019/Papers/elasticnet.pdf

34 Bengio 2012, Practical Recommendations for Gradient-Based Training of Deep Architectures, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.

A Note of Prior Function Performance

In practice we see L2 regularization give better performance over L1 outside of explicit feature selection[35].

As we'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 whole weight vector as we'd normally do with the L2 Norm. Using a method based on constraint 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 [17]. It has been shown to work well with stochastic gradient descent in deep neural networks even without dropout [18].

Dropout

Dropout is a powerful method of regularization that can be used across many types of models. Dropout is a computational 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 Stochastic Gradient Descent. 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.

Input Layers and Dropout

Sometimes we'll use no dropout at all on the input, especially for noisy or sparse data sets.

35 Li, Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes), http://cs231n.stanford.edu/, http://cs231n.github.io/

In hidden layers dropout is done with a probability of 0.5. By randomly omitting neurons we can prevent co-adaptation 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[36].

We can apply dropout (with probability 0.5) in Deeplearning4j by adding the following to our network configuration:

```
.regularization(true).dropout(0.5).
```

The Dropout Probability

In DL4J, the dropout probability (set 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% of the time, and set to zero 30% of the time. Note that in DL4J dropout(0.0) is used to mean "disable dropout for this layer" (as 0% 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. Though if we have a very large network (relative to the amount of training data), we may want to use a lower probability of retaining an activation (which corresponds to stronger regularization).

More notes when using dropout:

- Dropout is often combined with other regularization methods, such as L2 regularization, in practice

36 Srivastava et al. 2014: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

- As a general rule, we should avoid using dropout on the first layer in our network (disable it by setting dropout(0.0) on that layer)
  — this will avoid wiping out important parts of our input data set.

DL4J, Dropout, and Recurrent Neural Networks

For RNNs, dropout is generally only applied 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 RNN.

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 2-3x than the same network architecture with no dropout enabled. Noise in the gradients for learning is higher with dropout than standard stochastic gradient descent.

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[37].

Dropconnect

Dropconnect[38] 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 Topics in Regularization

A few other topics of note in regularization:

- Stochastic Pooling[39]
  — A regularization technique for convolutional networks (specifically) where we use a form of randomized pooling for building ensembles of convolu-

37 Srivastava et al. 2014: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

38 http://cs.nyu.edu/~wanli/dropc/

39 http://www.matthewzeiler.com/pubs/iclr2013/iclr2013.pdf

tional networks. Each network attends to a different spatial location of each feature map.

- Adversarial Training[40]
  - Inputs that can be created by using small but intentionally badly formed variations of the training dataset. This will generate a network model that gives the wrong answer with a high confidence. In practice this has been shown to reduce overfitting and result in models resistant to the adversarial examples.
- Curriculum Learning[41]
  - This variation of training involves starting with easier to learn examples from the training set for earlier training mini-batches and then progressively train on harder examples over time. This strategy gives the effect of faster training convergence to better solutions while acting as a regularizer for the network as well. Results have shown to have mixed success.

Parallelization and Regularization

We'd like to remove noise from the weights in the parameter vector 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. This can also be accomplished through the parameter averaging effect in the parallel runtime modes described later in this chapter.

In some cases of parallelization we'd use no prior because parameter averaging achieves the intended effect itself.

Working with Class Imbalance

Often we work with training datasets in machine learning where the classes are significantly unbalanced with respect to the number of each type of class. This typically occurs in domains where 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 on an ad on a webpage

40 https://arxiv.org/abs/1412.6572

41 Bengio et al. 2009, Curriculum Learning, http://www.machinelearning.org/archive/icml2009/papers/119.pdf

PhysioNet and Predicting ICU Death

The PhysioNet Challenge[42] 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 ICU death can be a challenge.

For the majority of learning methods, if we train the model on data that is 99% negative and 1% positive the model will just learn to always predict the dominant class (e.g. "negative"). The junior practitioner may not initially think anything is wrong as the average correct score will appear exceptionally high and we may 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% correct by always predicting "positive".

A good practice for building models in imbalanced data situations is to not optimize for percent predictions correct, strictly, and 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 its the most appropriate.

A Word on Model Calibration

In the domains where we see 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 the patient actually had the disease (false negative) and the patient dying. In this example, the consequences of not detecting a fatal disease are much greater than predicting falsely the patient has the disease. From this perspective, better understanding model evaluation is key for 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. We summarize these calibration techniques in the table below.

Table 6-4. 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 different from real proportion.	Local	Probability/confidence

42 https://physionet.org/challenge/2012/

Type	Task	Problem	Global/Local	What is Calibrated?
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] http://users.dsic.upv.es/~flip/papers/BFHRHandbook2010.pdf

As noted in the original paper, types CD and RD need to modify predictions in order to calibrate the results. While we don't further cover these methods in this book the authors felt compelled to note the topic for further study.

There are two main ways we can deal with class imbalance during the training process:

1. Subsampling from the larger class(es) or oversampling from the more infrequent class(es)
2. 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% positive, 50% negative"). A criticism of this method may be that we're throwing away good data but the upside is we're making training more efficient.

One approach is to use "post scaling". This is where we train network normally but scale outputs post-training. The results are less consistent compared to the other methods.

Another approach is called "probabilistic sampling". This is where we select samples by first randomly (e.g. through some probability) selecting a class then randomly select 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[43].

43 With regards 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.

Weighted Loss Functions

An alternative method to dealing with class imbalance during training is to use a lower weight to training examples that are in the larger class. This helps balance out the training impact of unbalanced dataset and tends to create models that have a better ability to detect the more rare labels in generalization.

To help the reader understand this concept further we'll illustrate it with a small section of code. In the code section below we can see an NDArray being created to hold the class weights for the weighting of the classes in the weighted loss funciton.

```
INDArray weights = Nd4j.create(new double[]{0.1, 0.4, 1.0});
```

We'd then pass these weights into a new instance of our loss function as highlighted in the code snippet below.

```
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)
    .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 1.0's 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 the larger datasets.

If our model picks up chance occurrence patterns in the training data then 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 tradeoff becomes that we have to 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 held-out test set. It is important that this test set isn't used for training, otherwise it won't tell us 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)
- Use early stopping
- Obtain a larger training data set
- Use a smaller network

Regularization is typically the first step in handling overfitting. L1 and L2 regularization can help (and has already been discussed earlier). Dropout is a frequently used and usually quite effective as an 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 that allows us to 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 Deeplearning4j'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 persist the stats to disk, change the UI port, or post results to a remote UI. Note that the listenerFrequency is used to set how frequently the information (activations, updates etc) are collected. We may 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).

Figure 6-4. DL4J Network Statistics Tool

DL4J's training UI has a number of pages, the first of which being the overview page. The overview page has 4 sections:

1. Top left: Model score vs. iteration (and a moving average of the score)
2. Top right: Summary network information
3. Bottom left: update:parameter ratios
4. Bottom right: activations, gradients and updates vs. 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 get added to the loss function. The second page in the UI is the model page as seen below in Figure 6-5. This allows us to visualize the model structure, and get more details for each layer individually

Figure 6-5. DL4J Training UI Model Page

The model page above 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). The right contains a summary table (layer configuration) number of graphs, including:

- 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, once 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 (e.g., 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 start 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:

1. Ensure 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)
2. Ensure that the variance of activation (and hence parameter) gradients are constant for all layers (i.e., 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 (for example, a layer of size 1024 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 may need to use a different initialization. This is more of an issue in deep networks as may be able to get away with a sub-optimal 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 seen below 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 3. 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 vs. iteration, on the model page as seen below in Figure 6-7:

Figure 6-7. Layer Activation Charts

Activations can also grow too large, if the weight initialization is too large. We can identify this using the same two charts as shown above.

Log Standard Deviation of Activations

In general, We want the log standard deviation of activations to be somewhere on the order of 1.0 for the hidden layers.

One thing to be aware of, 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 (e.g., we can see that the input standard deviation is 1 in the above figure, $log_{10}(1) = 0$. This is a good scale for the inputs). Consequently, we should also check our data normalization if we see unusually large or small activations.

Detecting Non-Shuffled Data

Neural network training is generally done using mini-batches where a mini-batch is defined as a subset of our data (typically 32 to 1024 examples). To get better results during training our data should be shuffled. This means that examples should be presented in random order, and not (for example) all of 1 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 the Figure 6-8 below:

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, then with only on class 1, and so on.

As we can see below in Figure 6-9, 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 take away here for the practitioner 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 Inside 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. If we set 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 in the Model page. The "param" tab shows the mean magnitudes of the parameters as we see below in Figure 6-10:

Figure 6-10. Weight Values Falling Towards Zero on the UI

We can see the mean weight magnitudes start 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 we can see below in Figure 6-11:

Figure 6-11. Parameter 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 we see below in Figure 6-12:

Figure 6-12. Parameter Histogram with a Few Large Weights

In Figure 6-12 above, it can be a little difficult to see what's happening, as 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.

CHAPTER 7

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, Hitchhikers Guide to the Universe

In this chapter we build on the concepts learned in the previous chapter on general deep network tuning. We take a deeper look at tuning specific architectures such as:

- Convolutional Neural Networks
- Recurrent Neural Networks
- Deep Belief Networks

Given that computer vision is one of the more popular applications in the world of Deep Learning we'll start out with learning to tune the Convolutional Neural Network architecture.

Convolutional Neural Networks

Convolutional Networks have some general design patterns and then specific convolutional architecture design patterns. We reviewed general network design patterns in the previous chapter. In this section we review techniques that are relevant to the Convolutional Neural Network architecture. Most of the content for this section is focused specifically on laying out patterns of:

1. Convolutional Layers
2. Pooling Layers

The convolution stage uses multiple filters to learn different features from the input to the layer as we previously outlined in chapter 4. The output of this stage is transformed with an activation function (e.g. the rectified linear activation function).

The Detector Stage

Some Convolutional Neural Network literature will separate out the detector stage as its own layer or stage in the Convolutional Neural Network 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 the above pattern demonstrated in the example layer list below:

- Input Layer
- Convolutional layer
- Pooling Layer
- Convolutional layer
- Pooling Layer
- Fully Connected Layer
- Fully Connected Layer

Sometimes we'll see multiple convolutional layers in a sequence before we see a pooling layer.

A Note About Larger Networks

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 down sample 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 the example previously in the book. We list the specific layer architecture below:

- Input Layer
- Convolutional layer
  - 20 filters [5x5]
- Max Pooling Layer [2x2]
- Convolutional layer
  - 50 filters [5x5]
- Max Pooling Layer [2x2]
- Fully Connected Layer

We see this architecture represented in a configuration object below in java code from the example in Chapter 5. We've included only the configuration portion of the code here for brevity and to allow the reader to compare the above architecture layout to the specific LeNet configuration in the code below.

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 filter
                .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();
```

As we can see above, 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 to detect 1 of the 10 MNIST digits in the training / test dataset. Another method to note is:

```
.setInputType(InputType.convolutionalFlat(28,28,1))
```

This method call does a few things, including:

- 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
  - can be used with other layer types too (RNNs, 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 non-linear 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 2 more than

once so we have enough information to down sample over time and build features. An example of an input size for a convolutional network 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 seen below:

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn and nOut specify depth. nIn here is the nChannels and nOut is the number of fi
        .nIn(nChannels)
```

This is configurable to the type of input, but will typically be 1 for monochrome and 3 for RGB. Of course, if the input was not image data and had a multi-dimensional representation, this number could be much larger.

Some Caveats About First Convolutional Layer Sizing

In general input layer unit count depends on stride/filter size.

An interesting example is AlexNet: 224x224x3 input, with 11x11 kernels (uncommonly large these days)[1].

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., 2) convolutional layers consecutively before we see a pooling layer.

In general, convolution neural network configurations tend to begin with larger filter sizes before progressively decreasing in size.

No Magic Bullet on Convolutional Layer Sequences

There's no one magic convolutional architecture (today) that will be best for all of the practitioner's image modeling problems. The authors suggest starting with a known successful architecture such as LeNet, VGG Net, Inception[2], or AlexNet and getting it to work on our intended problem. From there we can tweak the layer sequences and the hyperparameters, using the known architectures as "known good starting solutions" to begin from.

1 https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

2 https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf

Our output layer is going to follow the patterns set forth in the previous chapter on general network architecture tuning.

Configuring Convolutional Layers

We want to setup the spatial arrangement of the convolutional layer and then choose the number of filters.

How we setup the hyperparameters in a convolutional layer dictates how many neurons there are in the output volume and how they are arranged. The key hyperparameters are:

- Filter Size
- Stride
- Padding

For example, if we were to build a convolutional layer with 5x5 filters the code might look similar to:

```
ConvolutionLayer convLayer = new ConvolutionLayer.Builder()
   .kernelSize(5,5).stride(1,1).padding(2,2)
   .name("first_layer")
   .nOut(out)
   .biasInit(bias)
   .build();
```

In the sample above we can see the filter size (.kernelSize) is 5x5 and then we're taking integer parameters for both the stride (1,1) and padding (2,2) parameters. We're giving this convolutional layer the name "first_layer" with the .name() method and setting the number of filters with the .nOut(out) method call. The output of the layer itself is more complex than just the number of units out, and further below 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 example above 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 as we previously described in Chapter 4. If our stride is 2 then 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 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.

Leveraging Padding

In some components of certain architectures (like Inception) we want to preserve the spatial size of the input volume (covered in further detail below). To do 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 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. In Figure 7-1 below, we can see learned filters from the first convolutional layer in a network.

Figure 7-1. Visualized 96 Learned Filters, 11x11x3 size[3]

3 *Krizhevsky, Sutskever, Hinton, Advances in Neural Information Processing Systems (2012) - "ImageNet Classification with Deep Convolutional Neural Networks"https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf*

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 farther along in the progression of layers in a convolutional network the activation map size decreases. The layers closer to the input layer will tend to have fewer filters. As we move towards the output layer in a convolutional network we tend to see more filters.

While there is no specific heuristic to set the number of filters per convolutional layer, the authors mention that some of the larger CNN architectures start around 64, end around 1024[4] (at the high end) or so, increasing as we get further from the input.

Table 7-1. Filter Count Examples

Convolutional Network	Conv Layer Filter Count Progression
LeNet	20, 50
AlexNet	96, 256, 384, 384, 256
VGG Net	64, 128, 256, 512, 512

Computation Cost vs 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 as we build of 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 better[5] as opposed to big filters in fewer convolutional layers[6].

As filter size 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 disproportionally expensive in terms of computation. For example, a 5 ×

4 http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/

5 https://arxiv.org/abs/1409.1556

6 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 as well as GoogleLeNet (Szegedy) won ImageNet in 2014.

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 (3x3 or 5x5) and a stride of 1[7]. For comparison, GoogleLeNet[8] is a relatively complex convolutional network and the filters vary between 1 x 1, 3 x 3, and 5 x 5 in size.

Another popular network, VGGNet[9], has some interesting properties that have proven successful. The convolution layer filter size is usually 3 x 3 which is smallest size that retains the concept of of capturing pixels (e.g., left, right, up, down and center).

Comparison of Filter Size vs Consecutive Layers

Stacking 3 layer convolutional layers with filters of 3 x 3 and pooling is similar to a single convolutional layer with a 7 x 7 filter.

Adding more convolutional layers and filters allows us to leverage more regularization in the learning process. More layers applies more non-linear transformations and reduces the amount of parameters applied.

Another approach to get going on layer sequences and filter shapes is to start with successful architectures like VGGNet or GoogLeNet and draw inspiration from patterns[10] observed in those networks.

A Caution on Filter Size from the v3 Inception Paper

The authors note:

"The above results suggest that convolutions with filters larger 3 × 3 a might not be generally useful as they can always be reduced into a sequence of 3 × 3 convolutional layers."[11]

7 A good example is MNIST input images have a spatial dimension of [28x28] and generally have [5x5] sized filters on the first convolutional layer.

8 https://arxiv.org/abs/1409.4842, 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)

9 https://arxiv.org/abs/1409.1556

10 Filter/kernel sizes in networks such as AlexNet are a bit big by modern standards. As always it may be necessary to test and keep in mind we can set our image size after the fact to help things to work better (e.g. "crop or scale to some size").

11 https://arxiv.org/abs/1512.00567v3

Convolution Mode and Calculating Spatial Size of Output Volume

The general formula we use[12] on how to calculate the spatial size of the output volume as a function of the input volume size:

```
Output volume size = (W - F + 2P)/S + 1
```

Where

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 and if it is not then 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 using the Truncate mode as described below. 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 make sure 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 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 Convolutional and Subsampling layers, for a given input size and network configuration (specifically stride/padding/kernel sizes).

Currently, 3 modes[13] are provided:

12 http://cs231n.github.io/convolutional-networks/

13 https://github.com/deeplearning4j/deeplearning4j/blob/master/deeplearning4j-nn/src/main/java/org/deeplearning4j/nn/conf/ConvolutionMode.java

1. Strict
2. Truncate
3. Same

With strict mode, output size for Convolutional and Subsampling layers are calculated as follows, in each dimension:

outputSize = (inputSize - kernelSize + 2 * padding) / stride + 1.

If outputSize is not an integer, an exception will be thrown during network initialization or forward pass.

With truncate mode, output size for Convolutional and Subsampling layers are calculated in the same way as in Strict in each dimension. If outputSize is an integer, then 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: 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 may represent a significant proportion of the original input), or with large kernel sizes and strides.

The Same mode operates differently to Strict/Truncate, in three key ways:

1. Manual padding values in convolution/subsampling layer configuration is not used; padding values are instead calculated automatically based on the input size, kernel size and strides
2. The output sizes are calculated differently (see below) compared to Strict/Truncate. Most notably, when stride = 1 the output size is the same as the input size
3. The calculated padding values may different for top/bottom, and left/right (when they do differ: right and bottom may 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 since 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 [2x2] as seen below in the 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)

Receptive Field Limits and Max-Pooling

If we increase our pooling receptive field beyond these setting 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 Convolutional Neural Network 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 (e.g. 1.2M labeled training images). Although benchmark training datasets such as ImageNet have gotten better we seldom get a large labeled training datasets to perform large scale image training with Convolutional Neural Networks. One way to augment this issue is to start with a

previously trained Convolutional Neural Network model that had good results and then train it further on a specific image dataset. This is referred to as "transfer learning"
[14].

An Alternative To Training From Scratch

Convolutional Neural Networks 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 "building blocks" of vision processing in general. Two use cases of note in transfer learning:

1. fine-tuning an existing model
2. using an existing convolutional model as a feature extractor

While both of these methods use a pre-trained model on typically ImageNet, how 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 Convolutional Neural Network 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 only update 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 (e.g. "output layer") and consider the rest of the Convolutional Network as a "feature extractor" for the smaller dataset. This is also relevant as a pre-train dataset such as ImageNet would have 1000 different class outputs in the output layer and would possibly be not as relevant for a domain-specific task.

14 http://arxiv.org/abs/1411.1792

Training ImageNet from Scratch

Training a Convolutional Neural Network can take up to 2-3 weeks to train across multiple GPUs on ImageNet. It's common for practitioners to release their model for others to use as pre-training starting points to continue training their own models. The Caffe project has a model zoo[15] where we can download many pre-trained models as well.

When to Consider Trying Transfer Learning

We'd consider trying transfer learning when we see the below (or combinations of) situations:

- Our training dataset is small
- Our training dataset shares visual features with the base dataset

In other cases not covered above the practitioner may 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 pre-trained weights are already probably good.

Recurrent Neural Networks

Recurrent neural networks share some similarities with the other architectures such as Convolutional Neural Networks, but they have their own set of challenges and hyperparameters. There is no sure way, as with most neural networks, 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 the practitioner that for all networks its always going to be a game of trial and error.

15 https://github.com/BVLC/caffe/wiki/Model-Zoo

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: GRU (Gated Recurrent Unit) 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 timeseries data. In DL4J we represent this input with the parameters as seen in the following diagram:

- Number of examples
- Input size (number of columns)
- Timeseries Length

Figure 7-2. Representing 3D Input for Recurrent Neural Networks

Constructing volumetric input data for Recurrent Neural Networks takes more effort than traditional networks such as Multi-Layer Perceptrons. As we discussed previously in the book (and in the diagram above), there are 3 dimensions for input:

1. Mini-Batch Size

2. Number of Columns in our Vector per Time Step
3. Timeseries length

To understand more about how to do this easily check out the appendix chapters on "Using DataVec" and the "ND4J API". We'll also look at general patterns of timeseries vectorization in the next chapter.

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[16].

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)
- Timeseries 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):

Variable	Description
i	index of example in mini-batch
j	index of column per timestep
k	timestep index

In DL4J we use the RnnOutputLayer for the final layer with many recurrent neural networks for the tasks of:

- regression

16 Graves 2012, Supervised Sequence Labelling with Recurrent Neural Networks (PhD Thesis), http://www.cs.toronto.edu/~graves/phd.pdf

- classification

RnnOutputLayer handles functions such as:

- score calculation
- error calculation (prediction vs actual) given a loss function

We can see RnnOutputLayer is functionally similar to the standard OutputLayer in DL4J for canonical feed-forward networks yet it can deal with 3-dimensional output. Configuration for the RnnOutputLayer follows the same design other layers. For classification we set the last layer in a MultiLayerNetwork to a RnnOutputLayer:

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 where we classified sensor readings.

Training the Network

Recurrent neural network training can be computationally expensive. In general settings we see stochastic gradient descent 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[17].

Initializing Weights

Initializing weights is important in training recurrent neural networks. Good initialization of the weights has shown to create hidden units that can convey information over long enough distances to model long-term dependency tasks[18].

For most Recurrent Neural Network layers in DL4J we recommend to use the Xavier initialization[19] as demonstrated below in the code snippet.

17 Graves 2012, Supervised Sequence Labelling with Recurrent Neural Networks (PhD Thesis), http://www.cs.toronto.edu/~graves/phd.pdf

18 Sutskever 2013, Training Recurrent Neural Networks (PhD Thesis), http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf

19 Orthogonal weight initialization for recurrent neural networks has shown promise as well --- http://smerity.com/articles/2016/orthogonal_init.html

```
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)
```

Back Propagration Through Time

When our recurrent network is dealing with long sequence with many time steps, we find that truncated back-propagation through time (BPTT) works well. Backpropagation through time is an extension of the standard backprop algorithm for recurrent neural networks (as introduced in chapter 4). Truncated BPTT reduces the computational complexity of each parameter update in a recurrent neural network. In the code example below 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 shown above. 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 ways to use regularization with Recurrent Neural Networks. In Recurrent Neural Networks dropout is applied to only a subset of connections in LSTMs (non-recurrent connections). This variant of dropout has been shown to help[20]. Dropout as implemented for RNNs in DL4J is done on the input connections/activations only, not the recurrent connections.

20 Zaremba, Sutskever, Vinyals 2014, Recurrent Neural Network Regularization, http://arxiv.org/pdf/1409.2329v4.pdf

Standard Dropout and Recurrent Neural Networks

Standard dropout has been seen to not work well with recurrent neural networks as the recurrence amplifies noise and this hurts learning.

Recurrent neural networks are more sensitive to learning rate and momentum settings than multi-layer perceptron networks[21].

Debugging Common Issues with LSTMs

While it is fair to say that neural network tuning involves a lot of trial and error after a while we being 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 (e.g. called "exploding gradients", and are visible on 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 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 get too small and make it difficult to model long-range dependencies (10 timesteps or more) in the structure of the input dataset.

The most effective way to get around the vanishing gradient issue[22] in recurrent neural networks is to use the Long Short-Term Memory (LSTM) variant of recurrent neural networks which DL4J supports. Other methods to deal with this issue include:

- Combining short and long paths in unfolded flow graph (time-delay neural networks)
- Leaky units and a hierarch of different time units
- Gated Recurrent Units (GRUs)
- Better optimization methods
- Gradient clipping (for exploding gradients)

21 Sutskever 2013, Training Recurrent Neural Networks (PhD Thesis), http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf

22 LSTMs help solve the vanishing gradient issue, but do not inherently solve the exploding gradient issue.

- Regularization to encourage information flow
- Using L1 / L2 penalties

Padding and Masking

Padding involves adding zeros ("pad") to the end of any time series that are shorter than the longest timeseries 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).

In the diagram below we can see visually examples of each of these.

Figure 7-3. Different Variations of RNN Training

Padding and masking helps 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 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 1's it has the same effect as no masking array at all. In the case of the many-to-one case we'd have a single masking array for the output only. In DL4J these masking arrays are created during the data import stage (example: SequenceRecordReaderDatasetIterator) 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 (ie, when 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 in the code sample below:

Example 7-3. Evaluation setup

```
Evaluation.evalTimeSeries(INDArray labels, INDArray predicted, INDArray outputMask)
```

In the above example we have:

Table 7-3. 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 as 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 a different from the use of the above. We are calculating the value of the loss function value, not metrics such as accuracy or F1. However, in both cases: we only want to score/evaluate those time steps that are actually present, not those that are simply padded time steps. We can score timeseries data with 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 where the user may want to combine different types of architecture layers to model different mechanics of data such as video data. In some cases we may want to combine LSTM and Convolutional Neural Network layers to model a sequence of images (e.g. video frames). DL4J provides special pre-processor classes to help in these architecture variants such as CnnToRnnPreProcessor and FeedForwardToRnnPreprocessor.

A Note About Using Preprocessors

Recurrent Neural Network preprocesors can be added manually, but in many cases they will be added automatically by the network.

Restricted Boltzmann Machines

Restricted Boltzmann Machines are a type of neural network used to learn features from data sets in an unsupervised way, 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 to autoencoders.

Typically we're focused on using Restricted Boltzmann Machines for use cases such as:

- unsupervised feature learning in Deep Belief Networks

- 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 2/3 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 setup 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 only hold at most 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 leverage this formation to more effectively extract features and/or allow for more effective discriminative classification later in the pipeline (when used as part of Deep Belief Networks). What we want to look at is how much information is available to us to leverage. How much information is in your training record impacts 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 Restricted Boltzmann Machine networks that are not globally connected or use weight-sharing schemes we can use more units. So now let's look at a heuristic for choosing a specific number of hidden units for Restricted Boltzmann Machines.

Choosing the Number of Hidden Units

When we are trying to select the number of hidden units for Restricted Boltzmann Machines our main concern is avoiding over-fitting. 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 may get away with using more hidden units. In the cases where the input dataset contains a large number of input records that are very similar then we can use fewer parameters as well.

Leveraging Different Units

Binary (or logistic) units in Restricted Boltzmann Machines are typically where we start with Restricted Boltzmann Machines and Deep Belief Networks. Depending on the use case training setup where the data may not be well-modeled by these units we have other options to use for Restricted Boltzmann Machine hidden and visible units. For alternative situations in data modeling we list more unit types below:

- Multinomial Visible Units
- Gaussian Visible Units
- Binomial Units
- Rectified Linear Units

Multinomial units are for topic modeling, building recommender systems, and building a classifier with a Restricted Boltzmann Machine (where, to build the classifier we'd throw an output layer on top for classification after unsupervised pre-training). 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, as using Gaussian units for both will result in worse instability in training.

We see Gaussian units used in image and speech applications as 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'd use rectified linear units. 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[23]. Below in Table 7-3 we have a quick reference for the reader to refer back to when they want to look up unit types indexed by input data.

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 as 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 Restricted Boltzmann Machines 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 Restricted Boltzmann Machines 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 for overfitting. In some cases we'd like the biases to grow large so applying a weight-cost factor would make less sense here. For Restricted Boltzmann Machine's weight cost a great rule of thumb is to start with a value of 0.0001. Small differences in the weight-cost probably will not change how

23 https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf

your training process performs. It has also been shown[24] where dropout has been applied effectively to RBMs.

Deep Belief Networks

Deep belief networks involve a two phase training process: pre-train and fine-tune. Pre-training 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 deep belief network uses stacks of restricted boltzmann machines to learn features during the pre-train phase. In chapter 4 we gave an extended description of the interplay between deep belief networks and restricted boltzmann machines, refer back to that passage for more detail.

We want to initialize the weights in a deep belief network typically with small random values from a zero-mean Gaussian with a standard deviation of about 0.01. As the initial values get larger we may see faster initial learning but can have the side effect of a worse final model[25]. Hidden bias values of 0 are usually fine.

In training neural networks and more specifically Deep Belief Networks 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 deep belief networks 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 may 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.

24 Srivastava et al. 2014: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

25 https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf

We can tune momentum for contrastive divergence used for training Restricted Boltzmann Machines during pre-training. We like to start with a momentum setting of 0.5 and increase it to around 0.9 once the reconstruction error has settled down. If this creates too much of a shock in reconstruction error reduce the learning rate by factors of 2 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 towards 0.9. As the reconstruction error rises or we near a terminating condition the momentum will begin to fall towards 0.0.

Using Regularization

In the section above on regularization for Restricted Boltzmann Machines we've already covered the major reasons for using regularization (and this extends to pre-train in Deep Belief Networks).

Unsupervised pre-training with Restricted Boltzmann Machines can be seen as a form of regularization. Unsupervised pre-training amounts of a constraint on the region in parameter space where our algorithm can scan. Evidence points to unsupervised pre-training as a data-dependent regularizer when there is enough neurons or parameters. However, unsupervised pre-train has also been shown to hurt generalization in some cases. We see an example of this when the training data set size is relatively small (where small is under 100k rows of data).

Sparsity and Deep Belief Networks

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 pre-training[26] we want to use a small learning rate with no weight constraints to avoid losing the feature detectors discovered by the pre-training technique. Dropout is also effective at the fine-tune phase of Deep Belief Networks.

> **MNIST and Dropout**
>
> In a published account of using dropout on the MNIST dataset a 50% dropout setting in the hidden layers gave performance improvement over using no dropout. Performing an additional 20% dropout on the input units gave further improvement. The settings of 20% on the input units and 50% on the hidden units was often found to be optimal for the dataset.

Determining Hidden Unit Count

The main question we face we considering the hidden unit count for Deep Belief Networks is

> how many bits it would take to describe each input training vector? (to produce a good model)

Under this context our main issue we face is overfitting, which we've discussed previously around ways to 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.

26 Hinton et al. 2012, Improving neural networks by preventing co-adaption of feature detectors, http://arxiv.org/pdf/1207.0580v1.pdf

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. The reader may 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 where 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 CSV format yet still need to be transformed into vectors. We'd 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 may not even have a good way to attempt the text vectorization without extensive programming extraneous to the actual statistical modeling itself.

Raw Data and Automated Feature Learning

If the reader will recall in chapter 3 we call out a key aspect of Deep Learning as moving from "feature engineering" towards "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 our tools can operate on. Techniques of raw data vectorization allow us to do just this.

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 the reader a solid treatment of vectorization in a way that supported the Deep Learning modeling process and didn't distract with unwieldy side programming exercise for creating vectors. The other aspect is that as we move from the theory side of the book towards the practical side we want to setup 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 the reader 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 consume from hours to days, depending on the user's 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 multi-dimensional array of numbers). Neural networks still need to represent the input data as vector and matrices as they cannot work directly on text, graph, and other non-vector / matrix representation.

There are many different ways to handle the vectorization and many pre-processing steps can be applied giving us different grades of effectiveness on the output models. Many times 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 example input data below we see an example of columnar CSV data in the raw iris dataset[1] from the UCI 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 serialized sparse vector format below:

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468 4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0.5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0.5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149 4:0.5217391304347826
0.0 2:0.7500000000000002
```

Converting raw data into machine learning ready vectors can be broken down into two phases:

1. Vectorization
2. Normalization

1 https://archive.ics.uci.edu/ml/datasets/Iris

We cover each phase in the sections below. We'll also look at the concepts of data vectorization and normalization in three key areas of Deep Learning:

1. Sequential data
2. Image data
3. 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 are 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 so we'll also look at the sub-topic of image vectorization. We'll close with a review of text vectorization methods and a look at how word2vec is used in vectorization.

Deep Learning and the Changing Nature of Data Preparation

Deep Learning has shown that we can do less of these specific feature engineering pre-processing 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 experts of the past decade have preferred to hand craft 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 on in the chapter we'll see how a dataset with a small number of attributes may 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, TF-IDF, and word2vec.

Major things to consider when building models from raw data:

1. What kind of source data are we dealing with?
2. What type of model do we want to train with this data?
3. 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, how do we handle it?

Based on the source data type we'll have 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 multi-variate timeseries 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 start out with a review on how to deal with columnar data in the context of vectorization.

ETL (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 where 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 the practitioner needs to take into account what actions will be needed to be performed to build out their pipeline.

Strategies For Dealing with Columnar Raw Data Attributes

Columnar source data (e.g., an exported table from a 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. Since we're working in the world of the data scientist, we'll use the definitions for columns or attributes of data based in that work. Statistical textbooks typically define four types of attributes in data:

1. Nominal
2. Ordinal
3. Interval
4. Ratio

Below we take a quick look at each type.

Nominal

Ex: "sunny", "overcast", and "rainy"

Nominal attributes are sometimes referred to as "enumerated", "categorical", and/or "discrete". The term categorical means 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. 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	...

The table above shows example representations for 2 records where the first record had a value of "sunny", and the second record had a value of "rainy".

Ordinal

Ordinal values are like nominal values yet they have order. Ordinal values have rank giving us a notion of ordering yet 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.

An example of ordinal values would be "hot", "mild", "cool" (which direction these are ordered doesn't matter, as long as the convention is consistent). Another example of ordinal values would be "low", "medium", and "high". These values end up being converted into either integer values (e.g., "cool" = 0, "mild" = 1, etc) yet represented by floating point values at the code level in vectorization.

The differences between ordinal and nominal are not always obvious as there are subtle differences between them. Some of the older data mining systems tended to only cover nominal and ordinal types. If the vector representation is for the output vector then we'd want to use a one-hot representation. For input vectors, we may 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 may 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 won't require a transformation. We may choose to apply a method of normalization, however, as we'll cover later 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 forrests, etc.

In the classical variant of vectorization we create a fixed size n-length vector (were 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 came up with to represent our data. This is generally appropriate for a network such as multi-layer perceptron (where 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 then 4-dimensional tensor input for convolutional neural networks. For the purposes of this section we're going to focus on how we engineer individual feature values in these input tensors and then how we'd apply methods of normalization to these values.

The process of vectorization is based around selecting attributes and finding a dimension in a feature vector to assign the feature to. This is done in different ways depending on if we're working with CSV, text, image, or timeseries data. We still have to select which parts of the input data we want to work with and collect that in the input vector. The process of taking n attributes and turning them into m features in our output vector can be done in a number of ways and is referred to as "feature engineering". Raw source data typically needs transformation before we're ready for modeling. Raw source data can take on a number of manifestations ranging from:

- Raw text such as a document in a text file
- A file containing a tweet string per line
- Binary timeseries data in a custom file format
- Pre-processed 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 representation. Most cases the values of attributes are numeric and continuous. These attributes measure numbers either real or integer value. 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 place holder values depending on each attribute's dynamics. 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 for 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. Once again we see here that studying the statistics of the dataset and graphing attributes to be a great practice.

Techniques of feature engineering can be applied in the hand-coded vectorization methods of vectorizing columnar data or when we're working with more complex data such as timeseries or image data. These techniques include:

- 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 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 transform on the attribute.

Dealing with Missing Values

Many times with data we'll encounter missing values in our raw input data and 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 start 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 (example: year attributes that have a 0 value).

Some basic methods to deal with missing values are:

- filter out records with missing values
  - (caveat: may 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 values)

Normalization

Normalization methods scale the input training data to be in a range (e.g., [0, 1], [-1, 1], etc) after we've converted the raw data into a vector representation. We care about normalization as it can impact the activations in a neural network. If the inputs to a network are too large we see activations which are too large, potentially adversely affecting training. Conversely, if our inputs are too small, our activations may end up being too small as well. Calculated gradients in the network can be impacted in a similar fashion.

Some Hyperparameters are Assuming 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 to be more consistent in some way. There are two basic tools[2] we can combine in different ways to perform normalization:

1. Centering
2. 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 around the origin, or in most cases, the mean. Scaling is a method where we rescale the feature such that the feature has a variance of 1 across the entire dataset or each feature has a max absolute value of 1 across the training dataset.

Specific methods of normalization we'll cover in this chapter that leverage the above core techniques are:

- Standardization
- Min-Max Scaling
- Whitening
- Principal Component Analysis (PCA)

We see both centering and scaling applied in the common normalization practice standardization. Standardization will 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 non-outlier data to a small interval and this is something the practitioner should watch out for.

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 $x_1, x_2, \ldots x_N$ we calculate the mean as:

2 http://ciml.info/dl/v0_99/ciml-v0_99-ch05.pdf

$\mu = \frac{1}{N}\Sigma_n x_n$

The variance of a feature measures how much the values for a feature vary around their mean.

$\sigma^2 = \frac{1}{N-1}\Sigma_n (x_n - \mu)^2$

where μ is the sample mean, as defined above.

Standardization and Zero Mean, Unit Variance. When we "standardize"[3] 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:

$\mu = 0$ (the mean equals 0)

$\sigma = 1$ (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 in the equation below:

$z = \frac{x - \mu}{\sigma}$

This is also referred to as zero mean, unit variance. This equation will center the feature value around 0 with a standard deviation of 1.0, producing values on the range of [-1, 1].

Most Common Method of Standardization

The method we see most commonly in standardization is zero mean, unit variance. We typically see zero mean, unit variance used with dense vectors in neural networks.

Optimization methods, such as stochastic gradient descent, can greatly benefit from methods such as standardization. Many times when we see features on different scales we'll also see parameters being updated faster than others due to how the input features are on different scales. Standardization also helps methods that compare similarities between features based on a distance metric.

3 In some literature standardization is referred ot as "z-score normalization".

As we transform data fields from attributes to features in our vector we also need to consider how sparse (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 pre-processing step to zero the mean and unit variance before we complete the step to scale resulting values (now on the range of [-1, 1]).

The primary difference to illustrate here between sparse and dense data pre-processing is with dense data, we first do "zero unit / mean operations" and then do scaling whereas with sparse, we just do scaling. Obviously in the cases where our algorithm needs the data on the range of (0, 1) then we'd want to scale it to that range (*example: Restricted Boltzmann Machines).*

Standardize with Caution

Naively applying standardization to a sparse vector may 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 more bits of information in the floating point representation.

Min-Max Scaling. Another way to normalize data is to use min-max scaling where 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 tradeoff we're making with scaling compared with standardization is we end up with smaller standard deviations. We see that min-max normalization is more sensitive to outliers, as a single outlier (small or large) impacts 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 where pixel intensities need to be normalize 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 will term min-max scaling simply as "normalization" and then "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 whitening"[4] (whitening, for short). This is where we transform data such that is has an identity covariance matrix[5]. Whitening also has the property of decorrelating the data, helping to find an optimal way to represent the data in an efficient way. This transform is named due to how it changes the input data into a white noise vector. A known issue[6] with using whitening is that it can exaggerate the noise in our data in certain situations.

Principal component analysis[7] is a method to convert data in a dataset (which are possibly correlated) into a dataset of linearly uncorrelated variables. The 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 where 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 is that the pre-train phase learns representations of the data by training layers of Restricted Boltzmann Machines. 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.

4 https://arxiv.org/abs/1512.00809

5 https://en.wikipedia.org/wiki/Covariance_matrix

6 http://cs231n.github.io/neural-networks-2/

7 "On Lines and Planes of Closest Fit to Systems of Points in Space", Pearson - http://stat.smmu.edu.cn/history/pearson1901.pdf

Applying Normalization in RNNs and CNNs. When our model assumes some structure in the input, our normalization should as well. When we're working with recurrent network variants (e.g., "LSTMs") we want to consider all timesteps 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 convolutional networks, 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. While 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).

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 loss function: if we have values in the range 0 to 1 million, then we end up with large errors and much too large gradients.

However, after normalizing the labels (output values), we aren't predicting the original thing we care about any more so we need to undo the normalization for the network predictions. We can perform this reverse normalization process this with min-max and standardization as 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

We have other situations in building features from attributes where the modeling algorithm may wish the data is distributed according to a multi-variate 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.

Leveraging 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 the 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 have to be Big to Need More Speed

Just because we don't have terabytes of input data doesn't mean we won't need to consider a scalable solution for the ETL phase of our 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 chapter 5's examples we saw many examples where 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 base case for vectorization in machine learning (and deep learning) is to be able to handle the columnar (e.g. "csv") data type. To do this in local mode with DataVec we'd simply write:

```
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, numCl
```

We saw this code initially in chapter 5 when we built the multi-layer 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:

- 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 on DataVec

Check out the appendix chapter on DataVec 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 represent an image stored inside 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 images visualized as rows and columns.

	Col-1	Col-2	…	Col-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 dictate how many colors we can show per pixel. A 1-bit pixel can only show binary images (typically black and white pixels). 8-bit pixels are more common and show 256 colors (or greyscale) per pixel. For greyscale 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 jpeg, png, and gif.

Image Formats

Each format can store the image data different and perform varying levels of compression on the raw data. Image 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 timeseries 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 (eg: single frame vs 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 convolutional neural networks. In DL4J we want to convert each image into a 3-dimensional tensor representation in an NDArray object (see the appendix on ND4J for more on this topic) as seen below in Figure 8-1.

Figure 8-1. 3-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 4-dimensional tensors that have the 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

This 4D format is the standard in DL4J for working with convolutional neural networks and images.

In the event we want to vectorize data for a multi-layer perceptron network then we need to flatten it into 2-dimensions (e.g., minibatchSize, depth x height x width). To understand this variant of image vectorization, let's continue with our earlier visual

grid explanation from above. With an image the first thing we need to do is extract the pixel intensities from each location in the array of pixels. With images the basic concept is to take the concept of an M x N image and "flatten" this rectangle into a 1 x (m x n) array.

Table 8-3. Flattened Image Data

	Col-1	Col-2	…	Col-m-x-n
Row-1	pixel1	pixel2	…	pixel-m-x-n

The image is composed of a sequence of floating point values that indicate hue intensity physically but we tend to think of them as their logical form of the M x 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.

Table 8-4. Flattened Mini-Batches of Images in 2d Array

	Col1	Col2	…	Col-m-x-n
Image1	pixel1	pixel2	…	pixel-m-x-n
Image2	pixel1	pixel2	…	pixel-m-x-n
Image3	pixel1	pixel2	…	pixel-m-x-n

We need to "flatten" this rectangle into a 1 x (N x M) array per image record in our matrix as seen in the table above. We perform this flattening routine using CnnToFeedForwardPreprocessor and this is done automatically by doing .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'll add the preprocessor to flatten it for us.

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 translate them directly into a vector with some transforms per pixel. In more advanced meth-

ods of image processing sections of the image is parsed out and used as a vector by itself.

DataVec's ImageRecordReader is the record reader to use with image data, and perform common image operations such as cropping. ImageRecordReader supports a wide variety of image formats including

- jpg
- gif
- png
- tiff
- bmp

with efficient loading and image operations provided by JavaCV and OpenCV. Below we can see a short snippet of code where DataVec's ImageRecordReader is being setup to build a DataSetIterator which allows DL4J to train directly on image data.

```
ImageRecordReader reader = new ImageRecordReader( outputHeight, outputWidth, inputNumChannels, label
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, numCl
```

Functionality such as this makes DL4J a much easier platform to work with as the practitioner can focus on the overall ETL and vectorization pipeline flow and less on the minute details such as extracting pixel values from file formats. The ImageRecordReader is covered in further detail in the appendix on using DataVec, 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 may 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 termed as "timeseries data". Timeseries data is defined as a sequence of data points measured typically at successive time intervals. In the world of finance we see timeseries data[8] in the form of the daily-adjusted close price of a stock at the NYSE. Another example

8 For a more detailed look at timeseries in the wild and in practice take a look at the following blog post: http://www.cloudera.com/blog/2011/03/simple-moving-average-secondary-sort-and-mapreduce-part-1/

would be sensor readings from phasor measurement units devices (PMUs) on the smart grid[9] measuring "phase angle" and voltage 30 times a second.

One of the most common applications the practitioner can run into is log processing. Systems such as web servers, cell phones, and credit card readers are all generating log entries every time an action is performed in response to a request. These devices are a tremendous source of sequential and timeseries data. This data doesn't always fit that well in a traditional RDBMS system so we often see it stored on SAN drives or Apache Hadoop HDFS filesystems. The much-ballyhooed "Internet of Things" (IoT) largely consists of these use cases where we are ingesting and processing timeseries data from sensors (webservers, PMUs, cell phones, etc). Other sources of sequential data we might not think of directly are genomic data and then regular character sequences (e.g., sentences).

Many machine learning tools today have poor support for sequential data and the practitioner ends up having to hand-code complex ETL pipelines for handling this class of data. We also have a lot of sequential values that can be very noisy and we need the application of normalization and standardization. In the rest of this section we'll look at some practical ways to deal with the ETL and vectorization needs to 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. Three major ways we often see sequential data are:

1. Raw Log Lines
2. Single CSV File of Timeseries Data
3. Multiple CSV Files of Timeseries Data

Inside of these source files we may see different arrangements of data. We have entities (e.g., a "user", or other agent where 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 these data are stored on disk include:

1. All sequences in a single file
   - each line is a CSV set of readings (e.g., multiple columns) from a single source
2. Sequences spread across multiple files

9 https://openpdc.codeplex.com/

- each file in this variant tends to represent readings from a single source

In the first, 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 timeseries at each step we'll often see the data arranged in CSV files where each line is a timestep. Each file represents a single source of the multi-variate timeseries data. Each row will typically have one or more columns to represent the multiple values per timestep for the source.

We should also consider how many columns are measured at each step in the sequence of timeseries data. An example would be where we're measuring only "voltage" on the power grid, we'd only be measuring a single column of data per step in the sequence. If for every step (or timestep) we're measure both "voltage" and "temperature", then we'd need a file format with the ability to represent at least 2 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 disk (local, or from another location such as HDFS). This involves such tasks as:

- reading the file format
- process any joins or transformations of the data
- matching up the labels (if any) to the features for the relevant sources of sequences
- perform any normalization or standardization needed
- finally produce 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 leveraging that execution environment). Ultimately we want to construct a mini-batched set of sequential data with 1 or many columns in a tensor structure represented by an NDArray in ND4J. We want a 3-dimensional tensor structure as seen in the right side of Figure 8-2 below:

Figure 8-2. Representing Sequential Data for Recurrent Neural Networks

We want to create this 3D tensor (e.g., "3D matrix") where:

- mini-batch size is the first dimension
- feature columns are the second dimension
- timestep values for each column are the third dimension

It can get tricky to get perform all of the aforementioned ETL work and then align the data values in the NDArray tensor by hand. From that perspective, we recommend the practitioner leverage record readers and formats already in DataVec where possible.

Custom File Formats

If we do not have a record reader in DataVec for the particular type of input data then the practitioner will have to write some custom code for tensor construction. This can be a non-trivial exercise depending on the complexity of the data.

Let's take a look at some older variants of timeseries vectorization before we move on to how we do a few patterns of sequence conversion into tensors with DataVec.

Converting Timeseries 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 would represent a mini-batch of these vectors to train on, as we see in the left hand side of Figure 8-2 above.

The major drawback is how we're 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. Most of the time the practitioner was tasked with performing exotic methods of vector creation through hand engineering. While in some cases this was 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 (SAX)

There are a few ways to handle timeseries data in the world of flat vector representation. One method to deal with noisy timeseries data is to run a low pass filter over it as a pre-processing step. Symbolic Aggregated Approximation (SAX)[10] pre-processing is a great way to reduce noise in timeseries data. Dr. Eamonn Keogh's team at UC Riverside developed SAX. SAX is a symbolic representation of times series with some unique properties. Symbolic Aggregate approximation (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 timeseries data which helps learning algorithms.

The successor to SAX is indexable Symbolic Aggregate approXimation (iSAX) which added some additional capabilities. iSAX modifies SAX to allow "extensible hashing" and a multi-resolution representation and allows for both fast exact search and ultra fast 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 timeseries 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 3-dimensional tensor represented by a DataSet object. To illustrate this concept, let's consider the case where we were modeling data that had each entity's sequence data separated into a distinct file. There was also a corresponding file for the label for each entity as well. For example, we'd have:

- the file for the sequence data for entity 0:
  - train/features/0.csv
- label for sequence data entity 0:
  - train/labels/0.csv

10 http://www.cs.ucr.edu/~eamonn/SAX.htm

This data has only one column of data per measured step so we'd consider it "univariate". With DataVec we can leverage the CSVSequenceRecordReader as our record reader and then the NumberedFileInputSplit class will handled the file and directory naming schemes for us. We'd use the SequenceRecordReaderDataSetIterator class to produce our DataSet objects, which will leverage the record reader and input split objects as we can see below in the code snippet.

```
//Note that we have 450 training files for features: train/features/0.csv through train/features/449

SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain.getAbsolutePath() + "/%d.csv",

SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain.getAbsolutePath() + "/%d.csv", 0, 4

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'
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. Towards the bottom of the code we can also see how normalization and standardization might be applied to the data while in-flight in the DataVec pipeline.

Other variants of the record reader in DataVec include:

- CSVNLinesSequenceRecordReader
  - another version of CSV sequence reader from above
  - each sequence is exactly N lines long, all in one file, one after the other
- RegexSequenceRecordReader
  - Works well for log data where we need regular expressions to parse out the columns

In terms of dealing with file formats, above we've demonstrated the use of the NumberedFileInputSplit that understands how to read numbered files split out in directo-

ries. Obviously there are many other ways files can be laid out on disk and we'd want to subclass the FileSplit class to further deal with those scenarios.

Other topics to consider here is building custom DataSets from sequential data where 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'll cover vectorization of sequential data on Apache Spark in the next chapter.

Building Custom DataSets from Sequential Data

The easiest way to think about how to create the needed data structures for training a LSTM in DL4J is to realize we only need:

- 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 above line. An example of the might look like the code snippet below:

```
//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
    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. The main things to note about the code above are:

- we need to use ND4J to construct our input matrices
- the two loop pattern is generally a coherent way to walk through the 3-d data structure and setup our data
- we then mostly make calls to INDArray.setScalar(...) to set values

The trick we'll note here is how the label matrix data is laid out. If we're doing character prediction as in the Shakespeare example from chapter 5, we'd being putting the next character value at the current timestep in the labels matrix. If we're doing a sequential data classification model (e.g., classifying an anomaly in log data, for example) then we may put the class at every single timestep 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 timesteps. The strategy for aligning timesteps is a decision we leave to the discretion of the reader.

Using Masking in Tensors

Masks are used for both training data and labels in DL4J.

We set the mask timestep value to 1.0 for every timestep containing training data in the training data mask. We set the mask timestep value to 0.0 for all other timesteps. We normally use the same mask for both the input data structure and the label data structure.

Typically we'll see this in code where we're looping over the tensor data structure and we'll set the mask entry for each timestep as we go along. This might look like the code below:

```
INDArray mask = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
...
for (...) {
   ...
   mask.putScalar(new int[]{ miniBatchIndex, timeStep }, 1.0);
```

Notice we're only setting a mini-batch index and then a timestep index in the mask data structure. We don't need to specift which columns have data, just that any column in this specific timestep could have data.

Working with Text in Vectorization

Text can be unwieldy on 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, our simplest method of feature engineering (simply copying the attribute value over) will not work. We need methods to convert a variable number of attributes into a consistent number of features. As we'll see below, there are a number of ways to do this. In this section we'll review some of the older methods text vectorization and then compare those to more modern methods such as word2vec. Let's start out by looking at the fundamentals of free text and the vector space model.

The vector space model (VSM)[11] 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 then 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 - but we can skip straight to tokenization depending on use case
2. Tokenization – find individual words
3. Stemming[12] - finding the base or stem of words (optional)
4. Lemmatization[13] - collapses the different inflectional variants of a lemma (optional)
5. Removing stop words (optional)
6. Vectorization – we take the output of the process and make an array of floating point values

With the vector space model 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, yet for text this is actually not true as 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

11 http://nlp.stanford.edu/IR-book/html/htmledition/the-vector-space-model-for-scoring-1.html

12 http://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

13 In some languages stemming is "light" version of lemmatization.

a few of the most common 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 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 often used in natural language processing (NLP) and information retrieval (IR).

A group of words or a document is represented as a bag or "multi-set" 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 information retrieval domains[14].

	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}

In the table above, our vector has an index for every distinct word in the document. If the word "apple" corresponds to the 0 index, then for every time the word "apple"

14 One of the earliest references of the bag of words model can be found in Zellig Harris's article on Distributional Structure in 1954 (Orderless document representation: frequencies of words from a dictionary by Salton & McGill also appears in 1983)

occurs in the document we increment the value at this index. This ends up becoming a "word counting" exercise where we define a distinct set of words and then count how many times they occur. Once 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 yet we lose the word ordering information. We can also consider this representation to be equivalent to the histogram representation.

The bag of words model is commonly called a "term frequency" vector. This term frequency can be used in more elaborate vectorization schemes such as TF-IDF (defined later in this chapter). 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 only use 0 or 1 as the vector entry value to indicate that the word occurred or not. 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 pre-processing, also does not account for potential misspellings or word variations that we might want to combine into a single word. N-Grams and other pre-processing techniques alleviate both of these concerns[15].

Term Frequency Inverse Document Frequency (TF-IDF)

Term Frequency inverse document frequency (TF-IDF)[16] 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 nor in the same document. TF-IDF is used to measure that relative weight as how frequent 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 informational retrieval (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 may 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 leverage the information about how often a word occurs in a document while considering the frequency of the word in

15 Another vectorization variation on the bag of words model is the hashing trick where we use a hashing function to map each word to an index in a fixed sized vector.

16 http://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html

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 the inverse document frequency (IDF), where the former (TF) is divided by the later (IDF). To produce our TF-IDF number we start out 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 will get 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 classical version of TF-IDF we see that stop words get a small weight and the terms that are rare get a much larger weight. This allows us to "see" the words of relative high value such as the topic of a document. Important words will have both large term frequencies and large inverse document frequencies which produces a relatively high TF-IDF score for them.

Besides search engines, TF-IDF is also 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 have 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 pre-processing passes.

Term Frequency

Term frequency is defined in its simplest form as the number of time a word appears in a group of documents. Much like bag of words gives us an indication that a word occurred in the corpus, term frequency 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 so 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

$$tf_{t,d} = \frac{count(t)}{count(alltermsind)}$$

Inverse Document Frequency

The next step for calculating TF-IDF is to measure how much information a word provides by calculating the word's "inverse document frequency" (IDF)[17]. 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 get a higher value in the vector than terms that occur in many documents across the corpus[18].

To get 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 "inverse document frequency" we then take the logarithm of this derived fraction.

Table 8-5. Components of Inverse Document Frequency

IDF Variable	Description
df_t	document frequency; number of documents in the corpus that contain a term *t*
N	total number of documents in the input corpus

The IDF value in the above form is still not ideal, as 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:

$$idf_t = log\left(\frac{N}{df_t}\right)$$

Computing the Full TF-IDF Score

Thus the TF-IDF weight for a word wi becomes:

$$tfidf_{t,d} = tf_{t,d} \times idf_t$$

A high weight for a TF-IDF calculation is driven by words that have a relatively high term frequency and a relatively low document frequency with respect to the entire document corpus. Common terms tend to get filtered out as they will have low term frequencies 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.

17 http://nlp.stanford.edu/IR-book/html/htmledition/inverse-document-frequency-1.html

18 The inverse document frequency was original 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 see 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 pre-processing phases we can gain even more effectiveness though a technique called "n-grams".

DL4J includes an implementation[19] 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 below.

```
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();
```

19 Apache Lucene has a quality implementation of TF-IDF that is used in many places in the JVM landscape as well. --- https://github.com/apache/lucene-solr/blob/master/lucene/core/src/java/org/apache/lucene/search/similarities/TFIDFSimilarity.java

Removing Stop Words

A common pre-processing 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 get a more accurate measure between two document vectors. Examples of stop words include:

- "a"
- "an"
- "who"
- "the"
- "what"

Removing the words improves our TF-IDF measure considerably as we see initial distance values between two documents is dominated by the weights of these stop words.

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, which can miss dependent information in identifies such as "Wall Street" or "Coca Cola".

N-grams are used in tasks where 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 leverage 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 natural language processing 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 where we're trying to discover which language a text is composed in. When we parse a document or body of text we model words such that each n-gram is composed to n words in a sequence.

It's worth a note of caution that with n-grams when n is high enough it becomes computationally expensive to run the algorithm. The vocabulary growth is overwhelming even when n = 5, so n-grams should be used with caution and a lower n value.

Kernel Hashing

Another method we 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 hard when we deal with a lot of data with respect to having to touch a lot of hardware while trying to get good throughput. Kernel hashing is used when we want to vectorize the data in a single pass making it a "just in time" vectorizer. The advantage to using kernel hashing is that we don't need the pre-cursor pass like we do with TF-IDF but we run the risk of having collisions between words. It can be used when we want to vectorize text right before we feed it to our learning algorithm.

Word2vec and Vector Space Model Comparison

As we discussed previously 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, features such as the context of individual words. Specifically word2vec produces a list of word embeddings (vectors) where 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 encoding spatially and have useful characteristics such as vector arithmetic.

Vector space models (e.g., tf-idf), in contrast, are built using only cooccurrence 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 only leveraging cooccurrence information in the corpus).

Word2vec's vector representations have allowed for novel approaches to raw text understanding[20], recommendation[21], and graph modeling[22] as previously mentioned in this book. For code samples on how to construct word2vec (and paragraph vectors), the reader can refer back to the chapter 5 examples.

Working with Graphs

Vectorizing graph structures is a difficult problem in engineering data science pipelines. While we won't exhaustively cover the topic of graph vectorization here, we'll note the node2vec variant that we previously mentioned in chapter 5.

With node2vec[23] we can generate a vector for each vertex (and, by extension, each edge) in a graph. These vectors can be used just like word vectors (mentioned in the previous section) for a variety of tasks. Examples include:

1. node classification
2. inferring whether an edge (connection) should or should not exist

Node2vec is a generalization of another graph method called DeepWalk[24]. 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 sentences. Lastly, the authors would like to note the work on graphs[25] with graph convolutional neural networks[26] as something the reader should be aware of yet is not currently supported in the DL4J framework.

Adjacency Matrices

Vectorizing graphs as an adjacency matrix was a common method[27] 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 fil-

20 http://aclweb.org/anthology/W16-5607

21 https://arxiv.org/abs/1603.04259

22 https://arxiv.org/abs/1607.00653

23 https://arxiv.org/abs/1607.00653

24 https://arxiv.org/abs/1403.6652

25 https://arxiv.org/abs/1605.05273

26 https://tkipf.github.io/graph-convolutional-networks/

27 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

led with zeros (a node cannot have an edge connected to itself in this type of representation).

In this book we direct the reader more towards methods that are able to operate directly on the graph structure as opposed to custom variants of building adjacency matrices.

CHAPTER 9

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?

--- Waylon Jennings

Introduction to Using DL4J with Spark and Hadoop

Two key datacenter technologies that have emerged in the past decade are both 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 executing 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 AWS, Google Cloud, and Microsoft Azure can 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 the practitioner flexibility in how and where they run their deep learning workflows.

1 Currently DataBrick's cloud Spark offering is not supported, but that can may in the future.

Spark[2] is a general parallel processing engine that can execute on its own, on an Apache Mesos[3] cluster, or on a Hadoop cluster via the Hadoop YARN framework. It can work with data in Hadoop's distributed filesystem (HDFS) by leverage input formats included with Hadoop. Spark leverage some techniques around caching frequently used data in memory with Resilient Distributed Datasets (RDDs, covered below). Spark also allows the programmer to abstract away parallel processing at 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 in DL4J.

Below we list the key components of a spark job the practitioner should be aware of.

- 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. We only have a single driver per spark application.
- Spark Application Master
  - When we run spark on Hadoop via YARN we have a spark application master to negotiate 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 JVM instance on the local host the executor is running on. The Spark driver tells the long-lived executors what tasks they need to run. A single host may have multiple spark executors locally on it. A cluster may end up having 100's or even 1000's 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 a "resilient distributed dataset").
- Resilient Distributed Dataset (RDD)

2 http://spark.apache.org/

3 http://mesos.apache.org/

— fault-tolerant collection of elements that can be processed in parallel

Resilient Distributed Dataset (RDD)

Resilient Distributed Datasets 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.

Two common ways we can create a RDD in a Spark program are:

1. parallelize an existing collection in our Spark application
2. referencing a dataset in an external storage system

The external storage systems include HDFS, HBase[4], Cassandra[5], 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 was based on the published design of Google's MapReduce[6] and Google File System (GFS) papers. Hadoop was originally constructed to parallelize inverted index construction for the Apache Lucene[7] search engine as part of the Apache Nutch[8] project. The Hadoop project was started by Doug Cutting[9] and Mike Cafarella and ultimately kicked off the democratization of search engine infrastructure, later revolutionizing how data warehouses were 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 the sorting a Terabyte of data (on a 910 node Hadoop cluster, taking only 209 seconds)[10]. By 2009 many companies such as last.fm,

4 https://hbase.apache.org/

5 http://cassandra.apache.org/

6 https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

7 https://lucene.apache.org/

8 https://nutch.apache.org/

9 https://en.wikipedia.org/wiki/Doug_Cutting

10 http://sortbenchmark.org/YahooHadoop.pdf

Facebook, The New York Times and even the Tennessee Valley Authority[11][12] saw Hadoop as a practical way to use commodity hardware to parallelize processing on large amounts of data[13]. Today, multiple Hadoop distributions vendors have taken Hadoop to the Fortune 500 and beyond. Apache Hadoop has become the modern data warehouse infrastructure and is a defacto standard as an enterprise execution environment.

DL4J, Spark, and Hadoop

DL4J was designed from day 1 to be a first-class citizen in the Hadoop environment. DL4J is built to operate on stand-alone Spark, Spark on Mesos, or with Spark running via YARN on a Hadoop cluster. The practitioners who experiment with DL4J may be only using DL4J on a spark-only cluster, yet 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

Spark jobs are typically run from the command-line and include a number of parameters. Below 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[14], HDP[15]) with spark (via YARN) below.

```
spark-submit --class [class name] --master yarn [jar name]
    [job options]
```

- class name
  - this is the fully-qualified class name
- jar name
  - this is the path and jar file name

11 https://www.cnet.com/news/open-source-hadoop-powers-tennessee-smart-grid/

12 https://gigaom.com/2009/06/02/how-to-use-open-source-hadoop-for-the-smart-grid/

13 https://gigaom.com/2009/11/10/the-google-android-of-the-smart-grid-openpdc/

14 https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_running_spark_apps.html

15 http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.5.3/bk_spark-component-guide/content/ch_tuning-spark.html

  - this is also known as an "uber jar", including everything we need to run the job
- job options
  - various options the spark job takes

An example of this:

```
spark-submit --class io.skymind.spark.SparkJob --master yarn
    /tmp/Skymind-SNAPSHOT.jar /user/skymind/data/iris/iris.txt
```

In the job above 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 the user may want change certain job configuration properties from the command-line at execution time. We can do this either by putting these extra flags on the command-line or by specifying them in a special configuration file.

```
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 the one above 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 using cleartext to send credentials across the network. (if you aren't interested in reading about security, continue down to the "tuning spark" section below)

Major Hadoop distributions such as CDH and HDP have Kerberos authentication support.

Certified to Run on Hadoop Distributions

Skymind (commercial support for DL4J) makes sure new DL4J releases are certified on new releases of both CDH and HDP.

Kerberos credentials are optionally stored in 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 in:

```
/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[16].

Kerberos and Spark Note

On Kerberized clusters, if we try and 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. First we need to type:

```
kinit [user-name]
```

and this prompts us for a password. Once we have Kerberos initialized we can confirm that we have our Kerberos ticket by typing:

```
klist
```

This will show something similar to the below output and confirm that our Kerberos ticket is active and valid.

```
[skymind@sandbox ~]$ klist

Ticket cache: FILE:/tmp/krb5cc_1025

Default principal: skymind@HORTONWORKS.COM
```

16 http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.5.3/bk_spark-component-guide/content/run-sample-apps.html

```
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 Hortonwork's HDP distributions and then users of the Apache Mesos cluster management system.

Spark can execute in different ways based on if the execution is distributed or not and where the spark driver process for the job is executing while the job runs. Understanding some basic concepts around how this works helps the practitioner 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 be run in the foreground (client mode) or the background (cluster mode). Killing a foreground client, for instance, kills the job as we're killing the local process that is controlling the job on the cluster. We can quit the client session for a background client and allow the Spark job to continue to run as 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 it was executed on. In cluster mode, the spark driver process is run on a remote machine in the cluster itself.

Running Spark on Mesos

Some users may prefer to operate Spark in distributed mode without a Hadoop distribution with the supported Mesos[17] cluster manager. Apache Mesos is a way for a user abstract CPU, memory, storage, and other compute resources away from the concept of the physical machines themselves. It allows us to treat the cluster of machines as a single logical pool of resources, enabling fault-tolerance, multi-tenant 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.

17 http://mesos.apache.org/

There will be many short lived tasks on the Mesos cluster and Mesos is able to direct these tasks in such as way that it allows for multi-tenancy with other frameworks and tasks on the cluster.

There are two main modes to use spark on Mesos:

1. Client mode
2. 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 on Spark with Mesos in client mode[18], we need to:

- setup certain Mesos-related environment variables in spark-env.sh
- Pass in the correct Mesos cluster url to the SparkContext

In cluster mode[19] the 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. An example can be seen below:

```
./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
```

More Details on How to Leverage Mesos with Spark

Check out the page[20] for "running on mesos" on the Apache site for Spark. The page has a complete listing of key configuration options to note for configuring Mesos jobs.

18 https://spark.apache.org/docs/latest/running-on-mesos.html#client-mode

19 https://spark.apache.org/docs/latest/running-on-mesos.html#cluster-mode

20 https://spark.apache.org/docs/latest/running-on-mesos.html

There is also the scenario where the reader may 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 filesystem.

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 have 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. This section is not trying to confuse the reader with more distributed systems jargon yet 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 the reader to have an awareness of these processes to better understand how, where, and when they just run their Spark jobs.

Each spark executor runs as a YARN container[21]. Spark hosts multiple tasks in the same container allowing for orders of magnitude quicker task startup time.

yarn-client

In this mode the Spark Driver is executing on the machine where the job was submitted from. 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 inside YARN Containers.

21 Not to be confused with the concept of a Linux container, as this is a different concept.

yarn-cluster

The spark driver process runs remotely on an ApplicationMaster in a YARN-based Hadoop cluster (HDP[22], CDH[23]). The YARN Application master, from which YARN expects inputs to coordinate the job, also runs the Spark driver process. A Spark job run in this manner has the ability for the user to shut off their client or terminal session and still have the job complete.

Summary Table Comparing Spark Execution Modes

Below is a quick summary table to understand the effects of running Spark in different modes under YARN. This table was originally created by Sandy Ryza for Cloudera's Engineering Blog[24].

Table 9-1. Understanding the different Spark Execution Modes under YARN

	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark worker
Persistent services	YARN ResourceManager an NodeManagers	YARN ResourceManager an 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 many advantages including:

22 https://community.hortonworks.com/articles/28070/starting-spark-jobs-directly-via-yarn-rest-api.html and then launching remote spark jobs from NIFI: https://community.hortonworks.com/articles/73828/submitting-spark-jobs-from-apache-nifi-using-livy.html

23 https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_running_spark_on_yarn.html

24 http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

- the ability to dynamically share and centrally configure the same pool of cluster resources between different concurrent applications and frameworks
- the ability to take advantage of the yarn scheduler capabilities for better concurrency between applications
- the ability dynamically allocate and control how many executors are being used at any given time on the cluster for a single Spark application
- Kerberos support[25]

Being able to dynamically share the cluster between multiple applications is an important property of an enterprise production system. This means 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 gets. Being able to leverage the YARN or Mesos scheduler allows us to concurrently run ad-hoc queries on the cluster while also running a scheduled production job and making sure the production job gets enough resources to complete in the required SLA. This is because YARN (and Mesos, too) understands how to manage multiple frameworks concurrently in a way that governs resources in a predictable and manageable fashion.

In a multi-tenant 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 tuning 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 GPUs, whereas with YARN currently we do not.

Setting Number of Executors

We can set the number of executors we request for our application with a command-line flag or configuration file property. The command-line flag is set with:

```
--num-executor
```

The configuration file property is set with the property key:

25 Mesos supports most of these, except for the Kerberos support

```
spark.executor.instances
```

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 and is set on the command-line or in a configuration file property. The command-line flag is set with:

```
--executor-cores
```

The configuration file property is set with the property key:

```
spark.executor.cores
```

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 that 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. The command-line flag is:

```
--executor-memory
```

and we list the size in either megabytes (m) or gigabytes (g) of ram allocated per executor. We can also set this property from a configuration file with the property:

```
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 it's memory between transforming data and persisting data.

spark.shuffle.memoryFraction. Controls how much Java heap memory is used for aggregation and cogroups 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 and controls the fraction of Java heap memory to use for Spark's memory cache. This makes sure the executor doesnt exceed RDD heap space volume multiplied by this value.
The Spark executor can also have overhead which we manage with the setting spark.yarn.executor.memoryOverhead.

spark.yarn.executor.memoryOverhead. Controls the amount of off-heap memory to be allocated for each executor. This memory is used for:

- VM overhead
- interned strings
- other native overhead

The default value is (executorMemory * 0.10) with a minimum value of 384M.

Spark and YARN Container Resource Allocation

Working with Spark on YARN we have to consider about resources in the context of what resources YARN has available[26]. The major YARN properties to consider are:

- 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.memoryOverhaul together to get the full YARN memory request.

26 It's also worth noting that YARN trumps Spark in allocation because the Spark process runs within a constrained YARN container.

Figure 9-1. Hierarchy of Memory Configurations of Spark on YARN

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 yarn.nodemanager.resource.memory-mb setting in YARN then the job will not start as it will not get 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. We calculate the amount of memory available to each executor task as:

```
( spark.executor.memory * spark.shuffle.memoryFraction
 * spark.shuffle.safetyFraction ) / spark.executor.cores
```

Hopefully we've given enough overview of dealing with Spark and YARN tuning that they can get started. The reader should be aware that there are many more settings to tune in Spark and YARN beyond the scope of this book.

> **Further Reading**
>
> Check out Sandy Ryza's blog entries[27] on Cloudera's Engineering Blog for more Spark tuning tricks[28].

Understanding Spark, the JVM, and Garbage Collection

The topic of JVM tuning under Spark is large of complex. For the purposes of this book we'll summarize a few key topics and let the reader review the topic further as needed on their own.

Dealing with Slowing GC Efficiency or GC 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 this is leaving the executor less space to work with and we'll see performance degradation.

When we 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 RDD's after they are no longer needed.

Selecting a Garbage Collector for the JVM and Spark. It's generally recommended to use the G1 Garbage Collector[29] 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 the how tuning a DL4J job works on Spark. Just like with general Spark tuning, we can control 3 major factors in our DL4J jobs:

1. Number of workers (executors)
2. Amount of memory each worker has to use
3. Number of cores each worker has to use

We discuss how each affect training a Deep Learning model below.

27 http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

28 http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

29 https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

Tuning 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 on More Executors

Adding more executors (beyond a certain point) will have diminishing return on investment, in terms of accuracy obtainable per unit time for training. Effectively, once we've subdividing the workers' data split down to the mini-batch size and we have the same number of workers as data splits, we can go no lower. Adding more workers would be useless as the workers would have no data to process.

Tuning Amount of Memory for Executors

A single mini-batch of training records is sent to ND4J as a single matrices 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 as it not only affects learning but also how much memory the executor will need[30].

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.

Rules of Thumb for Memory Usage

For float data (most common): 4 bytes per value, plus some overhead.

Examples:

MNIST data (small): 28x28 = 784 → ~3kB (plus labels – 10 values) per example.

Time series with 256 inputs, 1000 time steps → 1MB 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

30 https://arxiv.org/abs/1508.02788

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 an 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 starting out by increasing the number of training epochs by 10% to account for the extra regularization. While 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 POM for Spark and DL4J

Setting up a Maven POM file is a key part of building a DL4J project, Hadoop job or Spark job. Apache Maven allows the practitioner to package all of the needed files and dependencies into a single JAR artifact[31]. In this section we'll look at some best practices when building DL4J POM files. The major dependencies for a DL4J Spark application are:

- DL4J
- ND4J
- DataVec
- DL4J-Spark

In the following sections we'll look at how to get these going in a Maven pom.xml file. Depending on the version or Spark we'll need to set the pom.xml file up 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 that this Maven

31 https://maven.apache.org/plugins/maven-assembly-plugin/

variable can take on check out the Hadoop vendor's websites for Maven artifacts (CDH[32], HDP[33])[34].

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

Note that the `_${scala.binary.version}` is a Maven property and 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. Summary of 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

The nd4j-native-platform Dependency

If building on one platform, deploying on another: use "nd4j-native-platform" dependency instead. Setting this property includes native binaries for all platforms.

An example would be if we are building our project in Maven on a Macbook Pro and we intend to run the job on a Spark cluster running Redhat on the nodes, we need to set this property to target Redhat.

32 https://www.cloudera.com/documentation/enterprise/release-notes/topics/cdh_vd_cdh5_maven_repo_57x.html

33 http://repo.hortonworks.com/index.html#welcome

34 For more on how semantic versioning works with Hadoop, check out the Apache Hadoop website: https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/Compatibility.html

A Pom.xml File Dependency Template

In this section we show how to setup 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 setup depending on the specific Hadoop distribution.

```
<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 not including these jars we can sometimes reduce jar size by 50-80% making compiling, moving, and executing

jobs easier. We can control which dependencies are included in our jar with the "<scope>" tag as shown in the example below:

```
<scope>provided</scope>
```

An example of a dependency with the scope set in this manner is shown below:

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>provided</scope>
</dependency>
```

A Note About Serialization

Spark still uses Java serialization by default which is generally considered a "bad idea". We see this mitigated by the inclusion of the Kryo dependency above with the setting:

```
<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 setup 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. Below we can see the properties for the 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>
```

Matching DL4J Component Versions

The version numbers for:

- nd4j.version
- dl4j.version
- datave.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 setup a pom.xml file for a HDP 2.4 Spark job. We assume the reader can figure out how to create the pom.xml boilerplate. These settings will fill out the key dependencies and versions of components for the job.

In the following section we show the properties section of the Maven pom.xml template for a HDP Spark job. These variables will match up to the dependencies in the following section.

```
<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 these properties in the above example pom.xml file 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'll list a few of the common issues the practitioner will hit periodically when using Spark on Hadoop.

Sometimes if the executor requests more RAM per container than is available the spark driver will 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 this can be alleviated by lowering the amount of ram or cores requested by the job.

Where to Look to Debug Spark

Below are 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

Below we list a few common issues encountered when using ND4J on Spark and their resolutions.

ND4J and Kyro 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 short-hand for serialization and deserialization of data in systems engineering parlance. Serialization on spark involves data and functions. Spark only cares 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[35]. To use Kryo serialization with ND4J on Apache Spark, it is necessary to set up some extra configuration for Spark. If Kryo is not correctly configured, it is 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:

```
SparkConf conf = new SparkConf();
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.kryo.registrator", "org.nd4j.Nd4jRegistrator");
```

A Warning About Incorrect Kryo Usage

Note that when using Deeplearning4j's SparkDl4jMultiLayer or SparkComputationGraph classes, a warning will be logged if the Kryo configuration is incorrect.

jnind4j and java.library.path

A common error is:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no jnind4j in java.library.path
```

If building on one platform and deploying on another (e.g. "build on osx, run on linux") use the "nd4j-native-platform" dependency instead. It includes native binaries for all platforms.

DL4J Parallel Execution on Spark

Deeplearning4j supports training neural networks on a Spark cluster, in order to accelerate network training.

35 This isn't so much a ND4J issue as it's really a Spark issue, but it's worth noting here from a practical standpoint.

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:
   - 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 of loading data, with different tradeoffs; 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 overheads of Spark). Instead, consider the following:

- For single CPU/GPU systems, use standard MultiLayerNetwork or ComputationGraph training
- For multi-CPU/GPU systems, use ParallelWrapper[36]. This is functionally equivalent to running Spark in local mode, though 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 may 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) starts 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, updater 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

Below we show the basic concepts we'll work with later in the Chapter to build out our full Spark examples.

```
JavaSparkContent sc = ...;
    JavaRDD<DataSet> trainingData = ...;
    MultiLayerConfiguration networkConfig = ...;
```

36 https://github.com/deeplearning4j/deeplearning4j/blob/master/deeplearning4j-core/src/main/java/org/deeplearning4j/parallelism/ParallelWrapper.java

```
//Create the TrainingMaster instance
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster = new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetO
        .(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 image above. 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. We explore these options below.

dataSetObjectSize

Required option. This is specified in the builder constructor. This value specifies how many examples are in each DataSet object. As a general rule,

- If you are training with pre-processed DataSet objects, this will be the size of those preprocessed DataSets
- If you are training directly from Strings (for example, CSV data to a `RDD<DataSet>` though a number of steps) then this will usually be 1

batchSizePerWorker

This controls the minibatch size for each worker. This is analagous to the minibatch 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 number of minibatches of size batchSizePerWorker. As a general rule:

- Low averaging periods (for example, averagingFrequency=1) may be inefficient (too much network communication and initialization overhead, relative to computation)
- Large averaging periods (for example, averagingFrequency=200) may result in poor performance (parameters in each worker instance may diverge significantly)
- Averaging periods in the range of 5-10 minibatches is usually a safe default choice.

workerPrefetchNumBatches

Spark workers are capable of asynchorously prefetching a number of minibatches (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 will also be distributed from the master to the workers. This adds extra time and network traffic, but may 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

Configuration setting for when data should be repartitioned. The ParameterAveragingTrainingMaster does a mapParititons 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, as some data necessarily has to 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 may be.

- NumPartitionsWorkersDiffers: Repartition only if the number of partitions and the number of workers (total number of cores) differs. Note however 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

Strategy by which repartitioning should be done

- 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 may 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 minibatch), the benefit may not outweigh additional overhead of executing the better repartitioning.

DL4J API Best Practices for Spark

Some key points for getting the most out of your Spark+Hadoop cluster include:

1. build a slimmed down jar
2. use a well-tuned cluster
3. optimize the ETL/vectorization pipeline
4. make sure 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.

A Note About What We Mean by "Good Distributed Filesystem"

There are several options in the land of distributed filesystems. The practitioner 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 not to setup nor maintain distributed systems infrastructure. A machine learning project that gets distracted with infrastructure complexities is a *deadringer*[37] for a project that will likely fail. From that perspective, we recommend to go for a distributed filesystem that includes the following properties:

- Is known to be reliable
- Supports kerberos integration
- Is well-tested
- Scalable
- Works with the bulk of the other components in our 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 GC pauses.

Multi-Layer Perceptron Spark Example

In our Multi-Layer Perceptron from Chapter 5 we modeled the non-linear synthetic Saturn dataset. In this example we'll change things up and revist the MNIST dataset

37 It's been a long road to get to this point in the book. The authors suggest the reader take a break and listen to the classic RJD2 album "Deadringer" - https://en.wikipedia.org/wiki/Deadringer_(album). Just imagine it as the soundtrack for the last half of this chapter.

(which was used in the chapter 5 CNN example) but apply the multi-layer perceptron network this time.

The major thing that has changed in this chapter's examples compared to chapter 5's examples is how we're approaching parallel training on Spark. Most everything else for the examples in this chapter will remain similar as compared to the chapter 5 examples.

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, as we see below, to do this 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 its less of a concern. We note this as a consideration to the reader for other projects involving CSV and text based data that may need extensive pre-processing.

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 patters in the examples[38] of the github repository to show how we can build effective parallel ETL pipelines in Spark for DL4J workflows.

We list the code for the Saturn example below (adjusted to execute on Spark).

Example 9-1. MLP Spark Example for Saturn Dataset

```
public class MnistMLPExample {
    private static final Logger log = LoggerFactory.getLogger(MnistMLPExample.class);

    @Parameter(names = "-useSparkLocal", description = "Use spark local (helper for testing/running with
    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);
```

38 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/tree/master/datavec-examples

```
}

protected void entryPoint(String[] args) throws Exception {
    //Handle command line arguments
    JCommander jcmdr = new JCommander(this);
    try {
        jcmdr.parse(args);
    } catch (ParameterException e) {
        //User provides invalid input -> print the usage info
        jcmdr.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
    TrainingMaster tm = new ParameterAveragingTrainingMaster.Builder(batchSizePerWorker)    //Each D
        .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'll 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 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 the code above we can see our network architecture is similar to some of the networks we saw back in Chapter 5. We highlight network architecture configuration code below.

```
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 stochastic gradient descent to optimize our parameters for the network. The multi-layer perceptron example in chapter 5 only had a single hidden layer, however, and this example has an additional hidden layer. We also see a switch to the leaky ReLU variant of the ReLU activation function, which was used in the chapter 5 multi-layer perceptron example. 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 Architectures 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

The two key differences we want to highlight in this code (compared to the single process examples in chapter 5) are:

1. Introduction of the ParameterAveragingTrainingMaster into this example
2. 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 hard. Below we see the code snippet from the above code listing that introduces the TrainingMaster as explained earlier in this chapter. This controls how we perform parameter averaging in Spark.

```
//Configuration for Spark training: see http://deeplearning4j.org/spark for explanation of t
TrainingMaster tm = new ParameterAveragingTrainingMaster.Builder(batchSizePerWorker)    //Ea
    .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 below.

```
//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 above, the SparkDl4jMultiLayer class takes 3 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 as in the code snippet below.

```
//Perform evaluation (distributed)
Evaluation evaluation = sparkNet.evaluate(testData);
log.info("***** Evaluation *****");
log.info(evaluation.stats());
```

This example is a great exemplar for how the basics of Spark training work with DL4J in a clean, easy fashion. The practitioner has to do little work 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

Change the current directory to the base directory of the example. Then we'll use Maven to build the job jar with the command:

```
maven package
```

This will produce a full job jar in the ./target/ subdirectory.

Once we've copied the job jar to our intended machine to execute on Spark, we type the command:

```
spark-submit --class org.deeplearning4j.examples.feedforard.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 the 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 we don't have to type them out every time. The other property of note is setting the number of workers (or "executors") in Spark with the --num-executors flag, where here we are setting it to 3.

Generating Shakespeare Text with Spark and LSTMs

Let's revisit the LSTM example from chapter 5 and see how we can modify the code and construct the same model on Spark[39]. In the explanations below we'll not go through the code for data loading as we want to focus on how the network architecture is slightly different and the training process under spark. We see the core training method in the example code below.

Example 9-2. Core Training Method for LSTM Shakespeare Example in Java

```
protected void entryPoint(String[] args) throws Exception {
    //Handle command line arguments
    JCommander jcmdr = new JCommander(this);
    try {
        jcmdr.parse(args);
    } catch (ParameterException e) {
        //User provides invalid input -> print the usage info
        jcmdr.usage();
        try {
            Thread.sleep(500);
        } catch (Exception e2) {
        }
```

39 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-spark-examples/dl4j-spark/src/main/java/org/deeplearning4j/rnn/SparkLSTMCharacterExample.java

```
    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
int nSamplesToGenerate = 4;                    //Number of samples to generate after each training e
int nCharactersToSample = 300;                //Length of each sample to generate
String generationInitialization = null;        //Optional character initialization; a random charact
// 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(Activa
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize).activation(Activation.
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT).activation(Activation.SOFTMAX)
        .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength).tBPTTBackwardLength(tb
    .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 infrequentl
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(examplesPerDataSe
        .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
        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 previously mentioned, we'll let the reader explore the rest of the code on their own and we are not going to list the data loading code here for the sake of brevity.

The two key differences we want to highlight in this code are:

1. Introduction of the ParameterAveragingTrainingMaster into this example
2. 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 changed in the previous multi-layer 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 hard. 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 below (from the section above) is that it is identical to the single process code example version back 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(Ac
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize).activation(Activat
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT).activation(Activation.SOFTMAX)
        .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength).tBPTTBackwardLengt
    .pretrain(false).backprop(true)
    .build();
```

We take the space to note this in the book as from an operational standpoint this is a huge advantage. We are able to take network configurations we've developed locally and run them on a Hadoop or Mesos cluster with Spark easily.

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 (RnnOutputLayer) with the MCXENT loss function coupled with the softmax activation function. As we recall from chapter 6, we use the softmax activation function anytime we want to predict 1 of many classes and here we're predicting the next character (1 of many). Let's now take a look at how this code slightly differs 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(examplesPerDataSetO
        .(other configuration options)
        .build();

//Create the SparkDl4jMultiLayer instance
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, networkConfig, trainingMaster);
```

Below we can see how this actually looks in the new modified LSTM example where we're 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 options available to the practitioner 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 Sp
//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(examplesPerDa
    .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 ScoreIterationListene

//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 tra
    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_CH
        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 example from chapter 5 and update it to run on Spark. In the example below we can see the code for the building a Convolutional Neural Network model with Spark. This example is similar to the earlier MNIST modeling example in this chapter but this time we're chang-

ing the network architecture over to a network more appropriate for image data: the convolutional neural network architecture.

Example 9-3. Spark MNIST Example with a Convolutional Neural Network

```
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 filter
                .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(examplesPerDa
        .workerPrefetchNumBatches(0)
        .saveUpdater(true)
        .averagingFrequency(5)                          //Do 5 minibatch fit operations per wo
        .batchSizePerWorker(examplesPerDataSetObject)     //Number of examples that each worker
        .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 below 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 setup the job to run in local mode or on a Spark cluster[40].

When we work work with Spark our data generally needs to be stored in an RDD structure. In the code snippet below we can see how we take the raw MNIST data from the iterator we previously used in chapter 5 and now are converting the data into 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
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 Convolutional network architecture in chapter 5, we can see the same architecture being setup in the code above. For simplicity we highlight this section of code below.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations) // Training iterations as above
.regularization(true).l2(0.0005)
.learningRate(.01)//.biasLearningRate(0.02)
```

40 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 only be used for development/testing. For data parallel training on a single machine (for example, multi-GPU systems) instead use ParallelWrapper (which is faster than using Spark for training on a single machine).

```
.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 fi
        .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 as 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. Let's take a look at how the training section of our example code is different from the similar example in Chapter 5.

```
//Create Spark multi layer network from configuration
ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster.Builder(examplesP
        .workerPrefetchNumBatches(0)
        .saveUpdater(true)
        .averagingFrequency(5)                          //Do 5 minibatch fit operations pe
        .batchSizePerWorker(examplesPerDataSetObject)     //Number of examples that each wor
        .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 we can see above are (as we've seen in the previous two examples):

1. Use of the ParameterAveragingTrainingMaster class
2. Use of the SparkDl4jMultiLayer class

Other than those two changes, the code is not terribly different.

Spark Code vs Local Code

A useful aspect to DL4J is in how creating a model on your local machine and creating a model on Spark is not significantly different. The major differences as listed above 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 is a discipline as old as the study of philosophy itself that 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 Artificial Intelligence was by author Pamela McCorduck when she wrote[1] that AI began with

> an ancient wish to forge the gods

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 Artificial Intelligence regularly and it becomes difficult to have a directed conversation on the topic.

We added this appendix because the practitioner commonly has to have a grounded conversation with customers, executives, and managers about what Deep Learning can do for them and how it fits into the Artificial Intelligence landscape. The themes

1 McCorduck, Pamela (2004), Machines Who Think (2nd ed.)

in this appendix are a mixture of a history of the discipline of Artificial Intelligence (for context) and discussions the authors have had with customers and industry peers. We seek to allow the practitioner to reset the narrative around Deep Learning, communicate realistic expectations 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 Artificial Intelligence has become over-hyped 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 practitioners such that we can still dream but with our feet on the ground. We'll work through some basic definitions, a short history of the topic of Artificial Intelligence, and then look forward to where this all could go. Hopefully we'll help folks avoid the pitfalls of previous cycles of Artificial Intelligence interest and better support their Deep Learning projects to be be more successful through responsibly setting goals and expectations.

The Story So Far

The main topic of this book, Deep Learning, constantly is 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 with past themes being:

- smart grid
- cloud
- big data

When we work in a domain such as the above or Deep Learning, as practitioners we have to delineate what is real from what is marketing hype. That begins with understanding the history of a subject and having solid definitions to build from. Let's start by reviewing what we consider to be Deep Learning and then we'll dive into the topic of defining Artificial Intelligence.

Defining Deep Learning

In chapters 1 and 3 we lay out a working definition for Deep Learning that is described as neural networks with the following properties:

1. more neurons than previous neural networks
2. more complex ways of connecting layers
3. cambrian explosion of computing power to train
4. 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

- generative modeling
  - generating art
  - generating 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 engineer features) from data in 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 exhibits no higher-level functions such as "automatically understanding the most interesting question to ask a dataset", let alone any type of sentient operation. Let's now come back to defining Artificial Intelligence.

Defining Artificial Intelligence

The history of Artificial Intelligence is fraught with myths, stories, and over-zealous marketing departments trying to tap into the latest technology narrative. To define Artificial Intelligence 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 a starting place 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 (who?) yet is at least 2000 years old. The field is based on understanding intelligent entities, studying topics:

- 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. Below we see a few of the building blocks of intelligent study over time.

Philosophy (400 B.C.). Philosophers began to suggest the mind as a mechanical machine where knowledge is encoded 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. Built on the ideas that animals and humans have a brain that can process information.

Computer Science. Came up with hardware, data structures, and algorithms to support reverse engineering basic components of the brain.

The study and application of Artificial Intelligence techniques we see today are based on the fundamentals over these past 2000 years. We typically see the study of Artificial Intelligence broken into a focus on either behaving of thinking in simulated intelligent systems. We see these applied in examples of:

- machine learning applications
- basic knowledge systems
- game playing (e.g. Chess, Go)

However, there are limitations to our implementations of intelligent study. There are still no good model of higher-order brain functions such as consciousness. Science has also yet to determine where consciousness resides in the brain. 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 Artificial Intelligence (if not the best) is Stuart Russell and Peter Norvig's "Artificial Intelligence: A Modern Approach"[2]. We can't recommend this book enough for the reader to get a more complete idea of the depth and history of Artificial Intelligence.

Cognitive Dissonance and Modern Definitions

When we're dealing with a topic such as Artificial Intelligence that ties into so many core definitions society depends on we naturally find dissonance around setting ground truth to work from. Beau Cronin writes[3]:

> 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,

2 http://aima.cs.berkeley.edu/

3 https://www.oreilly.com/ideas/ais-dueling-definitions

> 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 negotiations for 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 we don't understand natural intelligence, therefore, makes it hard to define the artificial variant.

Dr. Jason Baldridge wrote[4] on the topic of Artificial Intelligence 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 non-biological 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 to 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 a difference between Deep Learning and then 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 towards a better definition by segmenting the topic and breaking these segments down into simpler topics.

Francois Chollet recently made the salient comment on twitter[5]:

> artificial 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[6]:

4 https://www.peoplepattern.com/post.html#!/ai-not-yet-but-machine-learning-is-here-and-now

5 https://twitter.com/fchollet/status/734919468908875776

6 https://twitter.com/fchollet/status/734919888876097536

> Part of the problem is that some companies & 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[7]:

> 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 Artificial Intelligence is Not. Folks who claim machine learning to be artificial intelligence do the whole 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 above, its just best (for now) to avoid brain analogies.

Many times Artificial Intelligence is marketing as an application that is going to give the reader 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 the 2016 article[8] 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 Chapter 4's renders of Convolutional Neural Network filter renders. Dr. Epstein's central argument of 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 Artificial Intelligence:

> The intellectual establishment, by and large, preferred to believe that a 'machine can never do X'

They go on to demonstrate examples of where Artificial Intelligence researchers have systematically responded by demonstrating one X after another. Thus, the study of Artificial Intelligence 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".

---

7 https://twitter.com/fchollet/status/734920203478274048

8 https://aeon.co/essays/your-brain-does-not-process-information-and-it-is-not-a-computer

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[9] by Beau Cronin uses the following 4 major segmented definitions of Artificial Intelligence:

- 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 technology to determine what the user wishes to accomplish. AI as the interlocutor has limited reasoning capabilities as it typically relies on a separate system to send the voice-to-text and NLP processed results to as input. This separate system is many times as basic as a classic rule-base system or "expert system".

While the user may be initially entertained by conversation by the system and even fooled by its "intelligence", the limitations of the interactions are quickly met. 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 sub-systems just as the Interlocutor is.

9 https://www.oreilly.com/ideas/ais-dueling-definitions

AI as the Reasoner is a classical implementation of Artificial Intelligence 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 product value, such as 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 Artificial Intelligence. It is, however, a useful subsystem of an intelligent system.

Showing Restraint with Marketing Deep Learning

The practitioner should show restraint in labeling a machine learning model, deep learning model or not, as "Artificial Intelligence". Overselling capabilities early may get funding but will hamper the practitioner's project longer term.

A Fifth Aspirational Definition of Artificial Intelligence. 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 Artificial Intelligence is what brings on the unrealistic expectations that always crash down on the industry no matter how much tangible progress was made.

The Artificial Intelligence Winters

The Artificial Intelligence 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 over-hyped followed by a cycle of predictably underwhelming results. This down period is referred to as an "Artificial Winter" and involves cuts in academic research funding, reduced venture capital interest, and a stigma in the marketing realm around anything connected to the term "artificial intelligence".

The result of these cycles is the good technical advances (e.g. "voice recognition" or "optical character recognition") rebrands and is integrated into other products.

AI Winter I: (1974-1980). The lead up to the first Artificial Intelligence winter saw machine translation fail to live up to the hype. Connectionism (neural networks) interest waned in the 1970's, and speech understanding research over promised and under delivered.

In 1973 DARPA cuts academic research in the Artificial Intelligence domain. The Lighthill report in the UK harshly criticizes the field and research funding is further curtailed. In the background we see Webos continue work on backpropagation with limited resources.

AI Winter II: Late 1980s. In the late 1980's and early 1990's we see an over-promotion of technologies such as expert systems and LISP machines, both of which failed to live up to expectations. The Strategic Computing Initiative cancels new spending at the end of this cycle. The fifth generation computer fails its goals.

The Common Patterns of Artificial Intelligence Winters

The common patterns of these winters see the industry get over-hyped by a series of promising successes. Once conditions percolate enough we see the hype reach a nadir and the money for academic and industrial research pour into the Artificial Intelligence 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 unfullfilled, however, and the Trough of dissillusionment 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:

- informatics
- machine learning
- knowledge based systems
- business rules management
- cognitive systems
- intelligent systems
- computational intelligence

The name change may 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 false promises attached to the name "artificial intelligence."

An interesting note from an Artificial Intelligence meeting in the 1980's:

> *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 Artificial Intelligence?

Three major contributors are driving today's Artificial Intelligence interest:

1. The big jump in Computer Vision in the late 2000's
2. The Big Data wave of the early 2010's
3. Advancements in applications of Deep Learning by top technology firms

In 2006 Geoff Hinton and team at the University of Toronto published a key paper on Deep Belief Networks[10]. This provided the industry with a spark of creativity on what could be possible in improving on 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. "better face detection" or "Alexa at Amazon") were widely recognized in the technology media.

In the mid-2000's 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 the west coast web firms (e.g. 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 leverage these new and larger datasets.

Traditional Fortune 500 enterprise companies were bring online large distributed systems in the early 2010's to hold their growing transactional datasets. These enterprise companies tend to follow what the west coast web firms do by about 5 - 10 years. This has given rise to an interest in Deep Learning in the Fortune 500 and to systems that would allow the traditional enterprise to better leverage their investments in Big Data.

If we combine the overlapping above 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 where enthusiasm outstrips the reality of the road ahead.

10 https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf

It's high tide in coverage and enthusiasm around Artificial Intelligence. 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. Some of these applications include:

- 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 above use cases (and more) in this book. When the practitioner promotes Deep Learning and "Artificial Intelligence" we recommend finding real use cases such as these and standing on "solid ground". We mention solid ground here metaphorically as 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 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 5th aspirational definition of Artificial Intelligence 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[11].

Marketing departments are setting up similar scenarios as in the previous two Artificial Intelligence winters, unfortunately. The coals of the 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.

11 Jeopardy does seem to be a "solved problem", however.

APPENDIX B

RL4J and Reinforcement Learning

Author: Ruben Fiszel[1]

Preliminaries

We begin this appendix by an introduction to reinforcement learning and is then followed by a detailed explanation of DQN for pixel inputs and concluded by an RL4J example. Let's start 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 else than the combination (5-tuple):

- A set of states SS (eg: in chess, a state is the board configuration)
- A set of possible action AA (In chess, all the move that could be possible in every configuration possible, eg: e4-e5)
- The conditional distribution P(s′|s,a)$P\left(s' \middle| s, a\right)$ of 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 other with probability 0. Nevertheless, in a sto-

1 **http://rubenfiszel.github.io/**

chastic (involving randomness, eg: a coin toss) environment, the distribution is not as simple.)

- The reward function of transitionning from state s to s': $R(s, s')$ (eg: 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[2].)

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 simply the a elements a in A such that $P(s'|s,a) > 0$.

The markov property is to be memoryless. Once you reach a state, the past history (the states visited before) should not affect the next transitions and rewards. Only the present state matters.

2 https://en.wikipedia.org/wiki/Discounted_utility

Figure B-1. Schema of a MDP

Terminology

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 state to a final state.

$s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n$

Cumulative reward: The cumulative reward is the discounted sum of reward accumulated throughout an episode:

$R = \Sigma_{t=0}^{n} \gamma^t r_{t+1}$

Policy: A Policy is the agent's strategy to choose an action at each state. It is noted by π.

Optimal policy: The optimal policy is the theoretical policy that 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 episode. This policy might be intractable.

The objective of reinforcement learning is to train an agent such that he 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[3] use that fact to learn the V-function (see below).

Some reinforcement learning algorithms can work without being given the model. Nevertheless, in order to learn the best strategy, they additionally have 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 since it is a superset of model based reinforcement learning.

Observation Setting

Instead of being given access to the state, you might being given access to a partial observation of the state only. It is the same idea behind 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 particule in the universe). Fortunately, the partial observation setting can be reduced to a fully observed setting with the use a history (the state becomes an accumulation of previous states).

Nevertheless, it is most common to not accumulate the whole history. Either only the last h observations are stacked (in a windowed fashion) or you can use a Recurrent Neural Network (RNN) to learn what to keep in memory and what to forget (that is essentially how a LSTM works).

Abusing the language slighty for consistency purposes with the existing notation, history (even truncated ones) will also be called "state" and also symbolized S_t.

3 https://webdocs.cs.ualberta.ca/~sutton/book/ebook/node108.html

Single Player and Adversarial Games

A single player game has a natural translation into a MDP. The states represent the moment where the player is in control. The observations from those states are all the information accumulated between states (eg: as many pixel frame as there are in-between frames of control). An action is all the available command at the disposal of the player (In doom, go up, right, left, shoot, etc …).

Reinforcement learning can also be applied to adversarial games by self-play: The agent plays against itself. Often in this setting, there exists a Nash equilibrium[4] such that it is always in your interest to play as if your opponent was a perfect player. This makes sense in chess by example. If given a 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 get 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 start. 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[5] (the Go agent from DeepMind that beat the World Champion). The policy was bootstrapped (initially trained) on a dataset of master moves, then it used reinforcement learning and self play to increase furthermore the level (quantified with elo). In the end, the agent got better than policy it was learning from the original dataset. After all, it beat the master above all the masters. To compute the final policy, they 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. Firstly, because the input is not as high-dimensional. The manifold[6] is a lot closer to its embedding space. Nevertheless, a convolutional layer was still used in this case to use efficiently the locality of some subgrid board patterns. Secondly, because AlphaGo is not model-free (it is deterministic). In the following of this post, I will talk exclusively about the model-free 1-player setting.

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

4 https://en.wikipedia.org/wiki/Nash_equilibrium

5 https://deepmind.com/alpha-go

6 http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

everything, in principle, everything ironed and the true problems concealed. (Erwin Schrödinger)

From Policy to Neural Network

Our goal is to learn the optimal policy π∗ that maximize:

$$E[R_0] = E\left[\Sigma_{t=0}^{n} \gamma^t r_{t+1}\right]$$

Let's introduce an auxilliary function:

$$V_\pi(s) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_n \mid s_t = s, \text{policy followed at each state is } \pi\right\}$$

which is the expected cumulative reward from a state s following the policy π. Let suppose an oracle

$$V_{\pi^*}(s)$$

The V function of the optimal policy. From it, we could retrieve the optimal policy by defining the policy that among all available actions at the current state, choose the action that maximize the expectation of $V_{\pi^*}(s)$. This is a greedy behavior. The optimal policy is the greedy policy w.r.t to V_{π^*}.

$$\pi^*(s) \text{ chooses a s.t } a = arg\ \max_a \left[E_\pi\left(r_t + \gamma V(s_{t+1}) \mid s_t = s, a_t = a\right)\right]$$

If you were very attentive, something would sound 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 gonna use another auxiliarry function, the Q-function:

$$Q_{\pi^*}(s, a) = E_\pi\left[r_t + \gamma V_{\pi^*}(s_{t+1}) \mid s_t, a_t = a\right]$$

In a greedy setting, we have the relationship:

$$V_\pi(s_t) = \max_a Q_\pi(s_t, a)$$

Now, let suppose instead of the V oracle, we have the Q oracle. We can now redefine π^*.

$$\pi^*(s) \text{ chooses a s.t } a = \max_a \left[Q_{\pi^*}(s, a)\right]$$

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. Although they can get 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 case, with the right hyperparameters, they are unreasonably powerful[7]. Using deep learning with reinforcement learning is called deep reinforcement learning.

Policy Iteration

Now machine learning knowledge and common sense tells 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 have to get our labels another way. (:<).

This is where the magic of Monte Carlo come in. Monte carlo methods are methods that rely on repeated random sampling to calculate an estimator. (A famous example is the pi calculation[8]).

If we play randomly from a given state, the better states should get better rewards on average (thank you law of large numbers[9]). So without knowing anything about the environment, you can get 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 are also based on this property (shocking isn't it?). This is a phase of exploration that lead to unsupervised learning and enable us to extract meaningful label.

More formally,

Given a policy π, a state s and an action a, in order to get an approximation of $Q(s,a)$ we sample it according to its definition:

7 http://karpathy.github.io/2015/05/21/rnn-effectiveness/

8 http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html

9 https://en.wikipedia.org/wiki/Law_of_large_numbers

$Q_\pi(s, a) = E[r_t + \gamma r_{t+1} + ... + \gamma^n r_n \mid s_t = s, a_t = a]$

In plain english, we can get a label for $Q_\pi(s, a)$ by playing a sufficient number of time from s according to policy π.

From an aggregate of signals:

Figure B-2. One Signal

The actual learning is done by standard Gradient descent, using the labels in batches. The gradient is the standard Mean-Square Error one such that the td-error gets minimised at each iteration.

We use the Mean-Square Error loss function (l2 loss) with a learning rate of α and apply Stochastic Gradient Descent (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").

Note: Even if we use MSE, there is no square in the formula because the loss is applied afterwards 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

$Q_\pi(s_t, a_t) \leftarrow E_\pi[R_t] = E_{s_t, a_t, \ldots, s_n \sim \pi}\left[\sum_{i=t}^{n} \gamma^{i-t} r_i\right]$

We can converge to the rightful expectation:

Figure B-3. Many Signals

So we can now design a naive prototype of our learning algorithm (in Scala but it is intelligible without any Scala knowledge):

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, initalize thethe reward
    var (new_state, accuReward) = transitition(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 this is 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 hard to reach at random).

Exploration vs Exploitation

Exploring at random the environment will converge to the optimal policy … but only guaranteed after an almost infinite time: you will have to visit every possible trajectories (a trajectory is the ordered list of al the states visited and actions choosen during an episode) at least once. Considering how many states and branching there is, it is impossible. The branching issue is the reason why Go is so hard but chess is ok. In the real world, we do not have infinite time (and time is money).

Thus, we should exploit the past informations and our learning of them to focus our exploration on the most promising possible trajectories. This can be achieved through different ways, and one of them is $\epsilon\epsilon$-greedy exploration. $\epsilon\epsilon$-greedy exploration is fairly simple. It is a policy that choose 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.

At each new information, our actual Q functions gets more accurate about the present policy and the exploration is focused on better paths. The policy based on our new Q function gets better (since Q is more accurate) and the ϵ-greedy exploration reach better paths. Focused on those better paths, our q function explore even more the better parts and has to update its returns according to the new policy. This is an iterative cycle that enable convergence to the optimal policy 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:

\(Q_\pi(s, a) = E[r_t + \gamma r_{t+1} + \ldots + \gamma^n r_{n} \mid s_t = s, a_t = a] \\ = E[r_t + \gamma r_{t+1} + V(s_{t+1}) \mid s_t = s, a_t = a] \\ = E[r_t + \gamma r_{t+1} + \ldots + \gamma \max_{a'} Q(s_{t+1}, a')\} \mid s_t = s, a_t = a] \)

As in the Monte-Carlo method, we can do many updates of Q.

MSE:

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha\left[\left(\underbrace{r_t + \max_a Q_\pi(s_{t+1}, a)}_{\text{target}} - Q_\pi(s_t, a_t)\right)_{\text{TD-error}}\right]$$

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 net.

That 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 loosing board) and 0 otherwise. It diffuses the Q-values by finding a point between the two extremes [-1; 1]. A transition with Q value close to 0 represents a transition leading to a balanced board. A transition with 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 (since 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 kill 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 else 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 odd of winning a game for every transition ?

As long as we sample sufficiently 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 (eg: 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[10].

Initial State Sampling

In a 1 player setting (like the 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 only need 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 to sample directly from a played episode by our agent.

Q-Learning Implementation

So we can now design a naive prototype of our Q-Learning:

Example B-2. Naive 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-greddy policy
        val action = epsilonGreedyAction(state)

        //interaction with the environment
        val (nextState, reward) = transition(state, action)

        //Q-update
        update(state, action, reward, nextState)

        state = nextState
```

10 https://kofzor.github.io/Reinforcement_Learning_101/#comparing-reinforcement-learning-algorithms

```
    }
}

//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(_._2)._2

//Retrive action of the max Q value
def maxQAction(state: State) =
    actionsWithQ(state).maxBy(_._2)._1

//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 (else the neural output layer would have to be different at each state). It is many times solvable by having the full set A*A* of actions as output and ignore the impossible actions (some papers put the target of impossible actions at 0).

Experience Replay

There is one issue with using neural network as Q approximator. The transitions are very correlated. This reduce 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 optimise 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 performances 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-greddy 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 to detect 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 with the RGB directly, but then the network would have to also learn that additional pattern. It seems like the brain is hard-wired to combine colors (fortunately!). Thus, it would seem reasonnable to tolerate that pre-processing.

What you see:

Figure B-5. A Screenshot from the Game Doom

What the neural network sees:

Figure B-6. Neural Network Input from Screenshot

The image is resized into 84x84. Convolutional layers needs for memory and computations grow with the size of their input. 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 speed up the training.

Skip Frame

In the original atari paper, only 1 in 4 frames is actually processed. For the following 3 images, the last action is repeated. It speeds up roughly by 4 time the training without loosing much information. Indeed, atari game are not supposed to be played frame perfect and for most action it makes more sense to keep them for at least 4 frames.

History Processing

To give information to the Neural network about the current momentum, the last 4 frame (with skip frame, you pick 1 every 4) are stacked into 4 channels. Those 4 frames represent a history as previously discussed in the Observation setting section.

Figure B-7. History Processing and Stacking

To fill the first frames of the history, a random policy or a noop replay is used[11].

Double Q-Learning

The idea behind double DQN is that the network is frozen every M update (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 "jiggering". The Q update becomes:

$$Y_{\text{target}} = r_t + \gamma * \left(Q_{\text{target}}\left(s_t + 1, arg\ \max_a Q_{(}s_t + 1, a\right)\right)\right)$$

11 Side note: random starts can be used for fair evaluation

Clipping

The TD-error can be clipped (bounded 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). It can dramatically alter the efficiency of the learning. This is an important hyperparameter to not neglect.

Prioritized Replay

The idea behind prioritized replay (*https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html#prio*) 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 RL, it is useful to have a visual monitoring of the agent's progress. This is why I built the dashboard webapp-rl4j[12].

12 https://github.com/rubenfiszel/webapp-rl4j

Training: #3

Home / 3

Info

Id	3
MDP	DeadlyCorridor
Configuration	{ "seed" : 123, "maxEpochStep" : 10000, "maxStep" : 8000000, "expRepMaxSize" : 1000000, "batchSize" : 32, "targetDqnUpdateFreq" : 10000, "updateStart" : 50000, "rewardFactor" : 0.001, "gamma" : 0.99, "errorClamp" : 100.0, "minEpsilon" : 0.1, "epsilonNbStep" : 100000, "doubleDQN" : true }
Training Method	QLearningDiscreteConv
Last updated	moments ago
Max step	8 000 000
Progress	0% step:50178

Charts

Cumulative reward and Length Epsilon Start-Q, Mean-Q and Score

Epoch: 189

The most important is to keep track of cumulative rewards. This is a way to check that the agents effectively gets 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

But you might want to also track the loss (score of the neural network) and mean Q-values:

Figure B-10. Score and Mean-Q Graph

Unlike with classic supervised learning, the loss does not necessarily always decrease because the learning impacts the labels!

If used with target network, you should see some discontinuities from the non continuous evaluation of different target networks. Loss should decrease w.r.t to a single target network. The mean Q-values should smoothly converge towards a value proportional to the mean expected reward.

RL4J

RL4J is available on github[13]. Currently DQN with Experience Replay, Double Q-learning and clipping is implemented[14]. 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.

13 https://github.com/deeplearning4j/rl4j

14 For more details check out the original blog post: https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html

Here is a working example with RL4J to play Cartpole with a simple DQN. You can play Doom too. Check rl4j-examples[15] for more examples. It is also possible to provide your own constructed neural network model as an argument to any training method.

Example B-3. Basic RL4J Example 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 DQNFactoryStdDense.Configuration CARTPOLE_NET =
        new DQNFactoryStdDense.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, man

    //train
    dql.train();
```

15 https://github.com/rubenfiszel/rl4j-examples

```
        //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 a somewhat simple algorithm. The field of RL 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 AI. Until then, we can expect to be awed by its diverse applications into 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

Also known as "Jeff Dean's 12 Numbers".

Table C-1. Jeff Dean's 12 Numbers

Computer Architecture Operation	Duration
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk Seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Duration column units "ns" is short hand for nanoseconds.

APPENDIX D

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 into the mathematics, 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, where 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, etc. All of these are valid measures - but some of them are much harder to optimize than others. For example, the accuracy of our network may 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:

1. Quantify the goodness/badness of the networks's current predictions based on a differentiable loss function
2. Calculate gradients for each parameter in the network, by apply the rules of multivariable calculus to the loss function and network structure
3. 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 algorithm is as follows:

Input: network parameters $\mathbf{w}$, loss function L, training data $\boldsymbol{D}$, learning rate $\alpha > 0$
while *termination conditions are not met* **do**
  $(features, labels) \leftarrow D.getRandomMiniBatch()$
  $out \leftarrow getNetworkOutput(\mathbf{w}, features)$
  $\frac{\partial L}{\partial \mathbf{w}} \leftarrow calculateParameterGradients(w, L, out, labels)$
  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial L}{\partial \mathbf{w}}$
end

The key part here is the calculation of the partial derivatives $\frac{\partial L}{\partial \mathbf{w}}$. For those not familiar with partial derivatives, they can be thought of 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.

A Note on Step Size

Note that in most cases, $\frac{\partial L}{\partial w_i}$ is also a function of w_i - which is (a) why we can only take 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, then 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 it's 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 Multi-Layer Perceptron

Enough text, let's get to the mathematics. Here, we'll be considering a simple multi-layer perceptron (i.e., 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:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \Sigma_{i=1}^{N} (y_i - \hat{y}_i)^2$$

where $\boldsymbol{N}$ is the number of outputs, y_i is the ith label, and $\hat{y}_i = \text{output}(\mathbf{w}, \mathbf{f})$ is the network's prediction of y_i, given the feature vector $\mathbf{f}$ and current parameters $\mathbf{w}$.

Figure D-1. Single Layer Neural Network

Referring to Figure D-1 above, 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 relu) by σ, then the forward pass equations for this network are as follows:

$$z_i = b_i + \Sigma_{j=1}^{4} w_{j,i} a_i$$
$$\hat{y}_i = \sigma(z_i)$$

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, $\hat{y}_i$:

$$\begin{aligned}\frac{\partial L}{\partial \hat{y}_j} &= \frac{\partial}{\partial \hat{y}_j}\left(\Sigma_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2\right)\\ &= \frac{\partial}{\partial \hat{y}_j}\left(y_j - \hat{y}_j\right)^2\\ &= -2\left(y_j - \hat{y}_j\right)\end{aligned}$$

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 at softmax do not have this property, however).

$$\begin{aligned}\frac{\partial L}{\partial z_i} &= \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i}\\ &= \sigma'\left(z_i\right)\frac{\partial L}{\partial \hat{y}_i}\end{aligned}$$

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, $\frac{\partial L}{\partial z_i}$:

$$\begin{aligned}\frac{\partial L}{\partial w_{j,i}} &= \Sigma_{k=1}^{3}\frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial w_{j,k}}\\ &= \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial w_{j,i}}\\ &= \frac{\partial L}{\partial z_i}\frac{\partial}{\partial w_{j,i}}\left(b_i + \Sigma_{k=1}^{4} w_{k,i}a_i\right)\\ &= a_i\frac{\partial L}{\partial z_i}\end{aligned}$$

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 $\frac{\partial L}{\partial z_i}$:

$$\begin{aligned}\frac{\partial L}{\partial a_i} &= \Sigma_{j=1}^{3} \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial a_i} \\ &= \Sigma_{j=1}^{3} \frac{\partial L}{\partial z_j} \frac{\partial}{\partial a_i}\left(b_j + \Sigma_{k=1}^{4} w_{k,j} a_j\right) \\ &= \Sigma_{j=1}^{3} \frac{\partial L}{\partial z_j} w_{i,j}\end{aligned}$$

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 backwards, 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 minibatches of data. Additionally, this same approach can be used to calculate the derivatives for other (more complex) models, such as recurrent and convolutional neural networks.

APPENDIX E

Using the ND4J API

ND4J is a scientific computing library for the JVM. It is designed to run fast in production environments. It's main features are:

- versatile n-dimensional array object
- multi-platform functionality including GPUs
- linear algebra and signal processing functions

ND4S

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 check out:

http://nd4j.org/userguide

The Complete ND4J API Javadoc:

For a complete resource on the ND4J API check out the Java Doc:

http://nd4j.org/doc/

Design and Basic Usage

ND4J is designed to run in many places and integrates with today's modern data systems. Features include:

- 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 n-dimensional array. It is a rectangular array of numbers with some number of dimensions. We quantify an NDArray with the following properties:

- rank
- shape
- length
- stride
- data type

Below 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, these two terms can be used interchangeably.

Rank

The *rank* of a NDArray is the number of dimensions. 2d NDArrays have a rank of 2, 3d 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 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 of an NDArray (for example, *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: thus 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 differs 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 Java Virtual Machine (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:

1. scalars
2. transforms
3. 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.

Scalars

Scalars just take 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, since the take a single argument and perform an operation on it. Absolute value is a transform that takes the argument `x` like so`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 rowwise op. For example, we might run an accumulation on the array

```
[1 2
 3 4]
```

Which would give us the vector

```
[3
 7]
```

Reducing the columns (i.e. dimensions) from two to one.

Accumulations can be either pairwise or scalar. In a pairwise 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

Section content goes here

The ND4J Class

This class has a variety of helper static methods to help in the creation of NDArrays. Below we discuss a few of the more common methods the user 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...) yet fills the returned NDarray with 1's instead of 0's.

Initializing with Other Values. The Nd4j class methods can be combined with other operations to create arrays with other values. For example, to create an array filled with 10s:

```
INDArray tens = Nd4j.zeros(3,5).addi(10)
```

The above initialization works in two steps: first by allocating a 3x5 array filled with zeros, and then by adding 10 to each value.

Initializing with Random Numbers. Nd4j provides a few methods to generate INDArrays, where 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

`Nd4j.getRandom().setSeed(long)`

Controlling the Shape of NDArrays

Below we discuss how to control the shape of NDArrays in the process of performing basic operations.

Creating Basic Arrays

In the example below we create a 1-dimensional array with 2 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 2x2 NDArray. In the example below we create a 2-dimensional NDArray that has 2 rows and 2 columns. The first row has the values {1, 2} and then 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);
```

Output:

```
[[1.0 ,3.0]
[2.0 ,4.0]
]
```

Be Mindful of Row and Column Arrangements in ND4J

In terms of encoding, an NDArray can be encoded in either C (row-major) or Fortran (column-major) order. For more details on row vs. column major order, see:

https://en.wikipedia.org/wiki/Row-major_order

Nd4J may 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 is is possible to use a specific ordering for a given array, should the need arise.

Example: Add Two 2x2 NDArrays Together. Create a second array (arr2) and adding 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);
```

Output:

```
[[7.0 ,11.0]
[9.0 ,13.0]
]
```

Creating NDArrays from Java Arrays

Nd4j provides convenience methods for the creation of arrays from Java float and double arrays.

To create a 1d NDArray from a 1d Java array, use:

- Rov 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 3 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 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 Performance of NDArray Operations

Getting or setting values individually (for example, 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:

`INDArray.getDouble(int row, int column)`

For arrays of any dimensionality, you can use

`INDArray.getDouble(int...)`.

For example, to get the value at index `i,j,k` use

```
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. In order to get a single row from an INDArray, you can use

```
INDArray.getRow(int)
```

This will obviously return a row vector. Of note here is that this row is a view: changes to the returned row will impact the original array. This can be quite useful at times (for example: `myArr.getRow(3).addi(1.0)` to add 1.0 to the third row of a larger array). To copy of a row, use

`getRow(int).dup()`.

Get Multiple Rows. Similarly, to get multiple rows, use

```
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, you can use

```
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 ith 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()`,`isColumnVector()`

DataSet

The org.nd4j.linalg.dataset.DataSet class represents a data transform with input and output data[1].

In the context of neural networks a DataSet represents a pairing of input features output 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 below we see how we 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 produces the outcome NDArray.

```
DataSet t = ...
INDArray features = t.getFeatureMatrix();
INDArray lables = t.getLabels();
INDArray predicted = model.output(features,false);
```

This simple snippet of code 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. The practitioner needs to understand a vectorization strategy and then the practical application of the ND4J API to implement the vectorization strategy. As we mentioned previously, out input data for modeling is a set of input features associated with an output vector. In this section we will illustrate how to create both and tie them together with a DataSet object.

1 Full java doc: http://nd4j.org/doc/org/nd4j/linalg/dataset/DataSet.html

Simple Examples of Vectorization

These code snippets are mean to be simple to show off the basic concepts of setting features and label values. In practice there are other ND4J methods that may 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.

Let's start with the basics of creating features with ND4J.

Basics of Vector Creation

Let's start by creating a simple feature vector with 2 columns. We can see this example below:

```
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 tells ND4J to create an NDArray that is a single row with 2 feature columns.

Setting Feature Values

There are many ways to create feature values. In the example below 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 simple 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 as both the input feature data and the output data use the same data structure, the NDArray. To setup our label data we have 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 who has the same index as the label to be 1.0 as we see in the example below.

```
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 the practitioner needs to convert an NDArray to a MLLib vector we can use the MLLibUtil.toVector(INDArray) method as seen in the example below:

```
Vector prediction = MLLibUtil.toVector( myNDArray );
```

Converting from MLLib Vector to INDArray

In the situation where the practitioner needs to convert a MLLib vector to an NDArray vector we can use the MLLibUtil.toVector(Vector) method as seen in the example code below:

```
INDArray ndArray = MLLibUtil.toVector( labeledPoint.features() );
```

Making Model Predictions with DL4J

In this section we'll 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 to represent the model and pass in a 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 below 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 feedforward pass based on the input vector.

The output values contained in the NDArray returned by the .output() method is typically of the form:

```
[0.5, 0.5]
```

Above 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 what type of output layer generated the output. In the sections below 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") vs 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 may 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 Multi-Label 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 ) );
        }
    }
```

Giving us output similar to what we'd see below:

```
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 will be using 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 max probability as we see in the code snippet below.

```
INDArray predictions = trainedNetwork.output( ndArray );
int maxLabelIndex = Nd4j.getBlasWrapper().iamax( predictions );
```

Advantages of Using the .output(...) Method Over .predict(...)

The .output() method can be used with regression and classification so its a bit more flexible in how we can use it.

APPENDIX F
Using DataVec

Alex Black

DataVec is a library for handling machine learning data. DataVec handles the ETL (Extract, Transform, Load) 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 (CSV, etc), image and time series data sets, 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 format
- Functionality for performing common data transformation operations (often called data wrangling or data munging).

These two categories of functionality will be 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 (reading image and CSV data, for example), and second, by providing a simple a 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 should be avoided whenever possible.

The set of abstractions that DataVec provides is relatively simple.

Writable is an interface that represents a piece of data. For example, a DoubleWritable can be used to represent double-precision numerical values, and Text instances are used for text data \footnote{Additional types include IntWritable, LongWritable, FloatWritable and NullWritable. DataVec also provides a NDArrayWritable class for efficient handling of numerical array data using ND4J, such as images}.

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 can then be read by Deeplearning4j's RecordReaderDataSetIterator class, which also handles combining examples into minibatch DataSet objects. This process is shown in figure X below.

Figure F-1.
DataVec Processing Pipelines

Similarly, the SequenceRecordReader interface provides a mechanism for loading sequence (time series) data. Whereas a single (non-sequence) example is represented as a List<Writable> in DataVec, a sequence example is represented as a List<List<Writable>> in DataVec: we can think of this 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 jth value at the ith time step. The way of using a sequence record reader is virtually the same as a record reader, and is shown in figure Y below.

Figure F-2. DataVec Sequence Pipeline

One important aspect of the way DataVec and DeeplearningJ handles 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 data sets). The RecordReader, SequenceRecordReader and DataSetIterator interfaces all have .next(), .hasNext() and .reset() methods for this purpose.

Loading CSV Data for Multi-Layer Perceptrons

Loading CSV data can be done in a few lines. The process here can also be used for other delimited data, such as tab-delimited formats.

The code below shows how to load a CSV data file, and create a DataSetIterator ready for training a network in Deeplearning4j.

Example F-1. DataVec CSV Loading Example

```
//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, numClasse


//Train the network with the DataSetIterator:

myNetwork.fit( iterator );
```

Some points of note here:

- The no-arg constructor (new CSVRecordReader()) defaults to comma-delimited format and 0 lines skipped
- In RecordReaderDataSetIterator, the minibatch size is freely configurable by the user
- A number of other constructors exist on RecordReaderDataSetIterator for cases such as multi-label 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.
- Alternative delimiters can be used. 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 data is in a non-random order (such as all class 0 examples followed by all class 1 examples), data 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 perform common image operations such as cropping. ImageRecordReader supports a wide variety of image formats (including jpg, gif, png, tiff and bmp, amongst others) with efficient loading and image operations provided by JavaCV and OpenCV.

The ImageRecordReader implementation contains a great degree of flexibity as to how data files are loaded. A simple example for loading images is provided below. In this example, we will 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/" for example. Note that label directory names themselves can be anything.

Example F-2. DataVec Image Loading Example

```
//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, labelMake

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, numClasse



//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 following code to the above example:

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. For example, to randomly flip and randomly crop images:

Example F-4. Transform Steps Example

```
int maxCropPixels = 20;

ImageTransform transform = new MultiImageTransform( new Random(), new FlipImageTransform(), new CropImag


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 will focus on a simple but useful format:

- CSV data, with one time series per file (time series lengths may 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 (non-sequence) CSV example earlier, 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 Example

```
//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, l

DataSetIterator iterator = new SequenceRecordReaderDataSetIterator( reader, minibatchSize, labelIndex, n

//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 (for example, separate files).

Transforming Data: Data Wranging with DataVec

Frequently, machine learning data is in a format that requires some level of preprocessing before it can be used. This preprocessing can be as simple as removing some unnecessary columns from the data set, 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

Note that when 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, then loading it from disk when required for network training.

For executing a set of operations on a data set, Apache Spark is used; this enables execution on both a cluster and a single machine (via running Spark in local mode). Furthermore, by executing operations on Spark, it is possible scale to large data sets (hundreds of millions of records, or more). Though Spark execution is currently the only option, it should be noted that the DataVec API itself is independent of the execution platform; other cluster computing frameworks may be added in the future.

DataVec Transforms: Key Concepts

Unifying View on Loss Functions
For transformation functionality, DataVec has a few key design ideas and classes to be aware of.

First, examples in DataVec are represented in the same was as discussed in the section above: 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 the RecordReader and SequenceRecordReader classes can be used to load the raw data we wish 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 (for example, 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 below), or optionally using a JSON or YAML file.

DataVec provides a number of types of operations that can be executed on data. These are summarized in table Z below.

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 (non-sequence) format	Split sequences of custom transactions into separate records

DataVec Transform Functionality: An Example

This section will show a very simple example of how to perform some common operations on a small dataset.

For this example, assume we have some transaction data, from which we want to predict a label (fraud/legitimate).
For this example, let's assume we have the following simple data set, 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 using 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.

Now, this data set cannot be used as-is. To train a neural network, we require numerical inputs for our data. In order 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

These operations can all be specified using using a 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 have to load our data and execute the operations defined above, and save our data. This can be done 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 data sets), 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.

APPENDIX G

Working with DL4J From Source

Some developers may 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 setup 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 DeepLearning4J 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 should avoid downloading the source code.

Verifying Git is Installed

Type the following into the command line to verify git is installed and working:

```
git --version
```

If the command returns an error then we suggest installing git. If the reader doe not already have a github account we suggest signing up for a free github account.

Cloning Key DL4J Github Projects

To work with source, clone ND4J or DL4J, enter 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
```

The reader might also want to clone the DL4J examples so they can work with ND4J or DL4J's pre-built 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, please see our Quickstart page:

http://deeplearning4j.org/quickstart.html#walk

Downloading Source Via Zip File

Another way to get the source code is by clicking on the "download ZIP" button from the ND4J GitHub page.

```
https://github.com/deeplearning4j/nd4j/archive/master.zip
```

Using Maven to Build Source Code

Maven can be used in conjunction with Git to ensure that ND4J, DataVec and Deeplearning4j build correctly. To make sure we have the most recent, working version of these libraries locally we can change into their root directories and enter the following command into your prompt:

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

Running a clean install on ND4J, DataVec and Deeplearning4j, 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 Deep Learning tools that together provide a full platform for Deep Learning. There are multiple dependencies that can be wired together to perform different functions in support of Deep Learning models. DL4J uses maven to control how projects dependencies are wired together in a project. In this section we show the reader a few of the relevant dependencies they can use to build their own Deep Learning models, tools, and integrations.

Creating a new DL4J Project

Deeplearning4j 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 and above
2. Maven 3.2.5 or above (Dependency management and automated build tool)
3. Git

There are also some option steps the reader may want to perform including installing:

- Cuda 7 for GPUs
- Scala 2.10.x
- Windows
- Github

Let's begin setting up the reader's environment by setting up 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 IoT devices. It's a "write once, run anywhere" language.

To test which version of Java you have (and 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 Deeplearning4j projects easily. It works well with Integrated Development Environments (IDEs) such as IntelliJ.

To check if Maven is installed in your machine, and which version you have, enter the following into the command line:

```
mvn --version
```

If you do not have the most recent version of Maven, please update it. (As of this writing, it was 3.3.x). Instructions to install Maven are at:

http://

Download the compressed file containing Maven's latest stable version. Lower on the same page, follow the instructions specific to your operating system; e.g. *"Unix-based Operating Systems (Linux, Solaris and Mac OS X)."*

A Minimal POM 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 below. This includes:

```
<?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
    <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>
```

POM Explanation. POM files can be complicated so we'll note a few key configurations to help the reader better understand what's going on. Below we list a few 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

Integrated Development Environments

An Integrated Development Environment (IDE) will allow you to work with our API and build your nets 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 with IntelliJ

The free community edition of IntelliJ has installation instructions.

After those installs, if you can follow these steps, you'll be up and running (Windows users please see the Walkthrough section below):

http://deeplearning4j.org/quickstart.html#walk

Enter

```
git clone https://github.com/deeplearning4j/dl4j-0.4-examples.git
```

in your command line. (We are currently on examples version 0.0.4.x.). In IntelliJ, create a new project using Maven by going to

```
File/New/Project from Existing Sources
```

in the menu tree. Point to the root directory of the examples above, and that will open them in your IDE. Copy and paste the POM.xml file below. Wait for IntelliJ to download all the dependencies. (You'll see the horizontal bar working on the lower right.). Select an example from the lefthand file tree and click "run".

Setting Up Other Maven POMs

In this section we look at how to quickly setup Maven pom.xml files for ND4J, and other components of 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:

1. Click the linked version number under "Latest Version"
2. copy the dependency code on the left side of the subsequent screen
3. paste it 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 Deeplearining4j Getting Started page.

APPENDIX I

Setting Up GPUs for DL4J Projects

Vyacheslav Kokorin, Susan Eraly

Training neural networks involve a host of linear algebra calculations. 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 $, 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. And that means 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 book's writing, a pair of NVidia GeForce GTX 1070 GPUs would be a solid choice to begin with.

The features to consider when buying a GPU are the following:

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 as 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 200-300%, depending on the data dimensionality and model size.

It is also important to consider the interconnect options available between devices. As of writing this book there are only two options available on the market: PCIe and NVLink, with NVLink being the clear winner, capable of providing 160GB/sec bandwidth. This makes NVLink a highly-desirable feature for multi-gpu systems for all possible deep learning parallelism models. NVIDIA even offers a pre-built server powered with 8 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 parallel wrapper class that can be used to convert an existing model to one that will train in parallel.

A simple example is as shown below

```
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 number of GPUs. Exact values are subject to tuning, as they are task and hardware dependant.

Within the ParallelWrapper, the initial model will be duplicated, and each worker will be training its own copy of the model. After every X iterations, defined by averagingFrequency(X), all models will be averaged and then copied over 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% should be a good starting value.

CUDA on Different Platforms

Below we show where to find extended instructions for CUDA on different platforms.

CUDA on Linux

Instructions at the URL:

http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/#axzz4C8adWDKM

CUDA on Windows

Instructions at the URL:

http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/#axzz4C8adWDKM

CUDA on OSX

Instructions at the URL:

http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-mac-os-x/#axzz4C8adWDKM

Monitoring GPU Performance

Once you begin training neural networks on GPUs, you will want to monitor whether and how well the GPUs are working. Below we list some resources to help debug and monitor GPU operations.

Nvidia System Management Interface (SMI)

Install the Nvidia System Management Interface (SMI).

https://developer.nvidia.com/nvidia-system-management-interface

Using the Nvidia System Management Interface

There will be many things in the logs but look for "Java" in the output to better triage what ND4J is doing.

APPENDIX J

Troubleshooting DL4J Installations

If anything goes wrong, in attempting to run the examples, you'll need to do some troubleshooting. In the following sections we discuss some of the common issues seen by new users of DL4J.

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 may 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, enter `env` in the command line. To add more heap space, enter this command in your console:

```
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. This can be fixed by upgrading to the latest version of Maven, which is currently 3.3.x. To check your Maven version, enter `mvn -v` in the command line.

Maven and Path Variables

After installing Maven we may receive a message like this:

```
mvn is not recognised as an internal or external command, operable pro
gram or batch file.
```

That means you need Maven in our PATH variable, which we can change like any other environmental variable.

Bad JDK Versions

If we see the error

```
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
```

our install may have a Maven issue. Please update to version 3.3.x.

C++ and Other Development Tools

To compile some ND4J dependencies, install some dev tools for C and C++.

ND4J Specific Instructions

Please see our ND4J guide for specific instructions:

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 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.

standardio.h

This will affect files such as standardio.h. More information is available at:

http://nd4j.org/getstarted.html#windows

Monitoring GPUs

As mentioned previously in this chapter we can monitor GPUs with the Nvidia System Management Interface (SMI).

For Further Help On Monitoring GPUs

Instructions on monitoring your GPUs:

http://nd4j.org/getstarted.html#gpu

Using the JVisualVM

One major reason to use Java is its pre-baked diagnostics in the **JVisualVM**. Just enter `jvisualvm` in the command line 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. Click on the `Sampler` tab on the upper right, 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 .properties files 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'.
```

OSX and Float Support

Float support is buggy on OSX. 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 Outof-Memory error that looks like this, fork join is the problem:

```
java.util.concurrent.ExecutionException: java.lang.OutOfMemoryError ….java.util.concurrent.ForkJoinT
```

Online Support Through Gitter

Please feel free to ask us about error messages on our Gitter Live Chat.

https://gitter.im/deeplearning4j/deeplearning4j

When you post your question, include the following information (it will really speed things up!):

- Operating System (Windows, OSX, Linux) and version
- Java version
- Maven version
- Stacktrace

Precautions

Below we list some things to check when the DL4J examples don't build or run correctly.

Other Local Repositories

Make sure you have not cloned other repositories locally, the main deeplearning4j repo is undergoing continuous improvements, the latest of which may not be thoroughly tested with examples.

Check Maven Dependencies

Make sure all your dependencies for examples are downloaded from Maven rather found locally. To remove older dependencies type:

```
rm -rf ls ~/.m2/repository/org/deeplearning4j
```

Reinstall Dependencies

To rebuild the examples from source type:

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

in the dl4j-0.4-examples directory to make sure it's installed correctly

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 dev tools for C, including gcc. To check if you have *gcc*, enter *gcc -v* on your terminal or command prompt.

OSX

Some versions of the Apple developer tool Xcode will install *gcc* for you. If the reader does not already have gcc, enter:

```
brew install gcc
```

into the command prompt.

[*how does this work in practice with the xcode download?*]

https://developer.apple.com/xcode/downloads/

Windows

Windows users may need to install Visual Studio Community 2013, which is free. We can download Visual Studio Community 2013 from:

https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

Setting up PATH Environment Variables

The practitioner will need to add Visual Studio's path to their PATH environment variable manually. The path will look something like this:

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin
```

To determine if Visual Studio's path is setup correctly type into the command prompt:

```
cl
```

We may get a message informing you that certain *.dll* files are missing. Make sure that the VS/IDE folder is in the path (see above). If the command prompt returns usage info for the cl command then it's in the right place.

Setting up Visual Studio

To setup and configure Visual Studio follow the following steps:

1. Set up `PATH` environment variable to point to `\bin\` (for `cl.exe` etc)
2. Also try running `vcvars32.bat` (also in bin) to set up environment before doing `mvn clean install` on ND4J (it may save you from copying headers around)

vcvars32

`vcvars32` may 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 have to copy header files. But if you run the bat file from Explorer, since the settings are temporary, they're not properly set. So 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 on C++

In Visual Studio, you also have to click on C++. It is no longer set by default.

Java CPP and Windows

In addition, the include path for Java CPP (*https://github.com/bytedeco/javacpp*) 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

The practitioner can get the dll for the Windows 64 platform at:

http://avulanov.blogspot.cz/2014/09/howto-to-run-netlib-javabreeze-in.html

Download dll libraries and place them in the Java bin folder (e.g.`C:\prg\Java\jdk1.7.0_45\bin`).

Other Dependencies

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 liblapack3.dll`

(`liblapack3.dll` and `libblas3.dll` are just renamed copies of`libopeblas.dll`)

You can download compiled libs from:

http://www.openblas.net/

Linux

Ubuntu and Centos users 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 enter 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 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

Enter the following in your terminal (or ssh session) 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*, enter this line:

```
gcc --version
```

For more complete instructions, go here:

```
http://www.cyberciti.biz/faq/centos-linux-install-gcc-c-c-compiler/
```

About the Authors

Adam Gibson is a deep-learning specialist based in San Francisco who 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. Adam has been a computer nerd since he was 13, and actively contributes to the open-source community through deeplearning4j.org.

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 Josh 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 Masters in Computer Science from the University of Tennessee at Chattanooga where he did published research on mesh networks (tinyOS) and social insect optimization algorithms. Josh has over 17 years in software development and is very active in the open source space contributing to projects such as deeplearning4j, Apache Mahout, Metronome, IterativeReduce, openPDC, and JMotif.