

O'REILLY®

# Deep Learning

A PRACTITIONER'S APPROACH

**Early Release**

RAW & UNEDITED

Adam Gibson &  
Josh Patterson



---

# 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](mailto:corporate@oreilly.com) .

**Editor:** Mike Loukides

**Proofreader:** FILL IN PROOFREADER

**Production Editor:** FILL IN PRODUCTION EDITOR

**Indexer:** FILL IN INDEXER

**Copyeditor:** FILL IN COPYEDITOR

**Interior Designer:** David Futato

**Cover Designer:** Karen Montgomery

**Illustrator:** Rebecca Demarest

January -4712: First Edition

### **Revision History for the First Edition**

2016-08-29: First Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491914199> 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 author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) 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.

978-1-491-91419-9

[FILL IN]

---

# Table of Contents

Preface.....	xiii
<b>1. A Review of Machine Learning.....</b>	<b>21</b>
The Learning Machines	21
How Can Machines Learn?	22
Biological Inspiration	25
What is Deep Learning?	26
Going Down the Rabbit Hole	27
Organization of This Book	29
Framing the Questions	29
Linear Algebra	30
Scalars	30
Vectors	30
Matrices	31
Tensors	31
Hyperplanes	32
Relevant Mathematical Operations	32
Converting Data Into Vectors	32
Solving Systems of Equations	34
Some Basic Statistics	36
Probability	37
Conditional Probabilities	39
Posterior Probability	39
Distributions	40
Samples vs Population	43
Resampling Methods	43
Selection Bias	43
Likelihood	44

How Does Machine Learning Work?	44
Optimization	44
Convex Optimization	46
Gradient Descent	47
Stochastic Gradient Descent	49
Quasi-Newton Optimization Methods	49
Underfitting and Overfitting	50
Regression	50
Classification	53
Recommendation	54
Clustering	54
Logistic Regression	54
The Logistic Function	55
Understanding Logistic Regression Output	56
Evaluating Models	56
The Confusion Matrix	56
<b>2. Foundations of Neural Networks.</b> . . . . .	<b>61</b>
Neural Networks	61
The Biological Neuron	63
The Perceptron	66
Multi-Layer Feed-Forward Networks	70
Training Neural Networks	76
Backpropagation Learning	77
Activation Functions	82
Linear	83
Sigmoid	83
Tanh	84
Hard Tanh	85
Softmax	85
Rectified Linear	86
Loss Functions	88
Loss Function Notation	88
Loss Functions for Regression	89
Loss Functions for Classification	92
Loss Functions for Reconstruction	94
Hyperparameters	95
Learning Rate	95
Regularization	96
Momentum	97
Sparsity	97

<b>3. Fundamentals of Deep Networks.....</b>	<b>99</b>
Defining Deep Learning	99
What is Deep Learning?	99
Organization of This Chapter	109
Common Architectural Principles of Deep Networks	109
Parameters	110
Layers	111
Activation Functions	111
Loss Functions	113
Optimization Algorithms	114
Hyperparameters	117
Summary	121
Building Blocks of Deep Networks	121
Restricted Boltzmann Machines	123
Autoencoders	129
<b>4. Major Architectures of Deep Networks.....</b>	<b>133</b>
Unsupervised Pre-Trained Networks	133
Deep Belief Networks	134
Convolutional Neural Networks	137
Biological Inspiration	138
Intuition	138
Convolutional Network Architecture Overview	140
Input Layers	141
Convolutional Layers	142
Pooling Layers	152
Fully-Connected Layers	153
Other Popular Convolutional Network Architectures	153
Summary	154
Recurrent Neural Networks	154
Modeling the Time Dimension	155
3D Volumetric Input	157
Why Not Markov Models?	159
Network Architecture	160
Domain Specific Applications	167
Recursive Neural Networks	167
Network Architecture	167
Varieties of Recursive Neural Networks	168
Applications of Recursive Neural Networks	168
Summary and Discussion	169
Will Deep Learning Make Other Algorithms Obsolete?	169
Different Problems Have Different Best Methods	169

When Do I Need Deep Learning?	170
<b>5. Building Deep Networks.....</b>	<b>173</b>
Matching Deep Networks to the Right Problem	173
Columnar Data and Multi-Layer Perceptrons	173
Images and Convolutional Neural Networks	173
Timeseries Sequences and Recurrent Neural Networks	174
Using Hybrid Networks	175
The DL4J Suite of Tools	175
Vectorization and DataVec	175
Runtimes and ND4J	175
Starting a New DL4J Project	176
Java	177
Working with Maven	177
Integrated Development Environments	178
Basic Concepts of DL4J API	179
Loading and Saving Models	179
Getting Input For the Model	180
Setting Up Model Architecture	181
Training and Evaluation	181
Modeling CSV Data with Multi-Layer Perceptron Networks	182
Setting Up Input Data	184
Determining Network Architecture	185
Training the Model	187
Evaluating the Model	187
Modeling Hand-Written Images with Convolutional Neural Networks	188
High-Level Workflow for Modeling MNIST with LeNet	188
Java Code Listing for LeNet Convolutional Network	189
Loading and Vectorizing the Input Images	191
Network Architecture for LeNet in DL4J	191
Training the Convolutional Network	194
Modeling Sequence Data with Recurrent Neural Networks	195
Generating Shakespeare with LSTMs	195
Classifying Sensor Timeseries Sequences with LSTMs	202
Using AutoEncoders for Anomaly Detection	207
Reconstruction Error as Anomaly Indicator	207
High-Level Workflow for Training the AutoEncoder	207
Java Code Listing for Example	207
Setting Up Input Data	210
AutoEncoder Network Architecture	211
Training the AutoEncoder Network	212
Evaluating the Model	212

Applications of Deep Learning in Natural Language Processing	215
Learning Word Embeddings with Word2Vec	215
Other Semantic Embedding Variants	219
<b>6. Tuning Deep Networks.....</b>	<b>221</b>
Basic Concepts in Tuning Deep Networks	221
An Intuition for Building Deep Networks	222
Building the Intuition as a Step-by-Step Process	223
Matching Input Data and Network Architectures	224
Columnar Data	224
Image Data	225
Sequential Data	225
Audio Data	225
Video Data	225
Summary	226
Relating Model Goal and Output Layers	226
Regression Model Output Layer	226
Classification Model Output Layer	226
Working with Layer Count and Number of Neurons	228
Feed-Forward Multi-Layer Networks	228
Controlling Layer and Neuron Counts	230
Weight Initialization Strategies	230
Basic Rule of Thumb	231
Weights Connecting to TanH Units	231
Weights Connecting to ReLU Units	231
Using Activation Functions	231
Target Distributions	232
Linear	232
Sigmoid	233
Tanh	233
Rectified Linear Unit (ReLU)	233
Softmax	234
Softplus	234
Hard Tanh	234
Maxout	235
Summary Table for Activation Functions	235
Summary of Activation Function Usage	235
Applying Loss Functions	236
Intuition	236
Loss Functions for Regressions	236
Loss Functions for Classification	236
Summary	237

Understanding Learning Rates	237
Intuition	238
Summary	239
How Sparsity Affects Learning	240
Leveraging Histograms When Setting Sparsity	241
Sparsity and Other Hyperparameters	241
Applying Methods of Optimization	241
Tuning Intuition	242
Stochastic Gradient Descent	242
L-BFGS	243
Conjugate Gradient	243
Hessian Free	243
Comparing the Methods	244
Choosing An Optimization Algorithm	245
Optimization Methods Seen as Impractical	247
Summary	247
Leveraging Parallelization and GPUs for Faster Training	247
From Batch to Online Learning	248
The Cost of Moving Data	249
Parallel Iterative Algorithms	249
Parallelizing Stochastic Gradient Descent	251
Parallelization Effects on Training	253
GPUs	253
Controlling Epochs and Mini-Batch Size	255
Terminology For Passes Over Training Data	255
Determining Mini-Batch Size	255
Mini-Batch Size and Learning Rate	256
How to Use Regularization	257
Priors as Regularizers	257
Max-Norm Regularization	258
Random Targets	258
Dropout	258
Stochastic Pooling	259
Adversarial Training	260
Curriculum Learning	260
An Introduction to Hyperparameter Optimization	260
Manual Search	260
Random Search	261
Other Tuning-Related Topics	261
Visualizing Learning in Deep Belief Networks	261
Evaluating a Neural Network Model	269
Overfitting	269

Dealing with Class Imbalance	270
<b>7. Tuning Specific Deep Network Architectures.....</b>	<b>273</b>
Restricted Boltzmann Machines	273
Intuition	274
Hidden Units and Modeling Available Information	274
Deep Belief Networks	277
Pre-Training with Restricted Boltzmann Machines	277
Initializing Weights	277
Setting the Learning Rate	277
Using Momentum	278
Using Regularization	278
Sparsity	279
Determining Hidden Unit Count	279
Convolutional Neural Networks	280
Intuition	280
Working with Spatial Arrangement	281
Configuring Filters	283
Common Convolutional Architectural Patterns	285
Configuring Layers	287
Recurrent Neural Networks	288
Intuition	289
Network Input Data and Input Layers	290
Output Layer	292
Padding and Masking	293
Training the Network	294
Evaluation and Scoring With Masking	296
Regularization	297
Variants of Recurrent Network Architectures	297
<b>8. Vectorization.....</b>	<b>299</b>
Introduction to Vectorization in Machine Learning	299
A Missing Link	299
Why do We Need to Vectorize Data?	300
Converting Raw Data Into Vectors	301
Understanding Raw Data	301
Strategies For Dealing with Raw Data Attributes	302
Transforming Raw Source Data	303
Traditional Feature Engineering Techniques	304
Feature Copying	304
Feature Scaling	305
Standardization	305

Binarization	305
Normalization	306
Zero Mean, Unit Variance	306
Dimensionality Reduction	307
Deep Learning and Feature Learning	307
Well Known Vector File Formats	307
SVMLight	308
libSVM	308
ARFF	309
Working with Text in Vectorization	310
Free Text and Vector Space Model	311
Bag of Words	311
Term Frequency Inverse Document Frequency (TF-IDF)	313
N-grams	316
Kernel Hashing	317
Image Vectorization	318
Working with Timeseries in Vectorization	320
DataVec and Vectorization	321
<b>9. Using Deep Learning and DL4J on Spark.....</b>	<b>323</b>
Introduction to Using Spark and Hadoop	323
What is Spark?	323
What is Hadoop?	325
Operating Spark from the Command Line	325
Configuring and Tuning Spark Execution	328
Understanding Spark Execution Modes	328
General Spark Tuning Guide	331
Understanding Spark, the JVM, and Garbage Collection	334
Tuning DL4J Jobs on Spark	334
Understanding Parallel Performance	335
Setting Up a Maven POM for Spark and DL4J	336
Major Dependencies for a DL4J Spark Job	336
Understanding Versions	336
A Pom.xml File Dependency Template	337
Setting Up a POM File for CDH 5.X	340
Setting Up a POM file for HDP 2.4	341
Controlling Jar Size	342
Troubleshooting Spark and Hadoop	342
Common Issues with ND4J	342
Common Issues with Spark	343
List of Key Ports for Spark and YARN	344
DL4J Parallel Execution on Spark	344

Distributed Network Training	345
A Minimal Spark Training Example	346
Working with the TrainingMaster	346
DL4J API Best Practices for Spark	349
Slim Down the Jar	349
A Well-Tuned Cluster	349
Efficient Vectorization Pipelines	349
A Well-Tuned JVM Goes a Long Way	349
Multi-Layer Perceptron Spark Example	349
Since We Last Spoke	349
Spark Code	350
Building the Spark Job Jar	354
Setting Up Network Architecture	354
Tracking Progress and Understanding Results	355
Recurrent Neural Network Spark Example	356
Loading and Vectorizing Data	356
Setting Up Network Architecture	356
Tracking Progress and Understanding Results	356
Modeling MNIST with a Convolutional Neural Network on Spark Local	356
Java Code Listing for Spark Local LeNet Example	356
Configuring the Spark Job in Code	359
Loading and Vectorizing MNIST Data	359
Setting Up the Convolutional Network Architecture	359
Training the Convolutional Network on Spark	360
Building the Spark Job Jar	362
Executing the Spark Job	362
<b>A. What is Artificial Intelligence?.....</b>	<b>363</b>
<b>B. Numbers Everyone Should Know.....</b>	<b>375</b>
<b>C. Setting Up DL4J Projects in Maven.....</b>	<b>377</b>
<b>D. Setting Up GPUs for DL4J Projects.....</b>	<b>379</b>
<b>E. Using the ND4J API.....</b>	<b>383</b>
<b>F. Using DataVec.....</b>	<b>397</b>
<b>G. RL4J and Reinforcement Learning.....</b>	<b>409</b>
<b>H. Other Deep Learning Libraries.....</b>	<b>423</b>

I. Evaluating Deep Learning Platforms.....	427
J. Working with DL4J From Source.....	431
K. Troubleshooting DL4J Installations.....	433
L. References.....	443

## 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

We architected the book in this manner as there was a need for a book in this space 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 democratize the material 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 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) to the broadest audience possible for this reason. The common thread amongst all disciplines interested in Deep Learning is their interest in the topic itself.

If you found 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.

## About Early Release books from O'Reilly

This is an early release copy of *Deep Learning: A Practitioner's Approach*. The text, figures, and examples are a work in progress, and several chapters are yet to be written. We are releasing the book before it is finished because we hope that it is already useful in its current form and because we would love your feedback in order to create the best possible finished product.

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](mailto:jpatterson@floe.tv).

## 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 Adam Gibson and Josh Patterson (O'Reilly). Copyright 2016 Adam Gibson and Josh Patterson, 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](mailto: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/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](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

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

Follow Josh Patterson on Twitter: [@jpatanooga](#)

Follow Adam Gibson on Twitter: [@agibsonccc](#)

## Acknowledgements

### Josh's Acknowledgements

Section content goes here

### 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 2016 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, defined machine learning as a

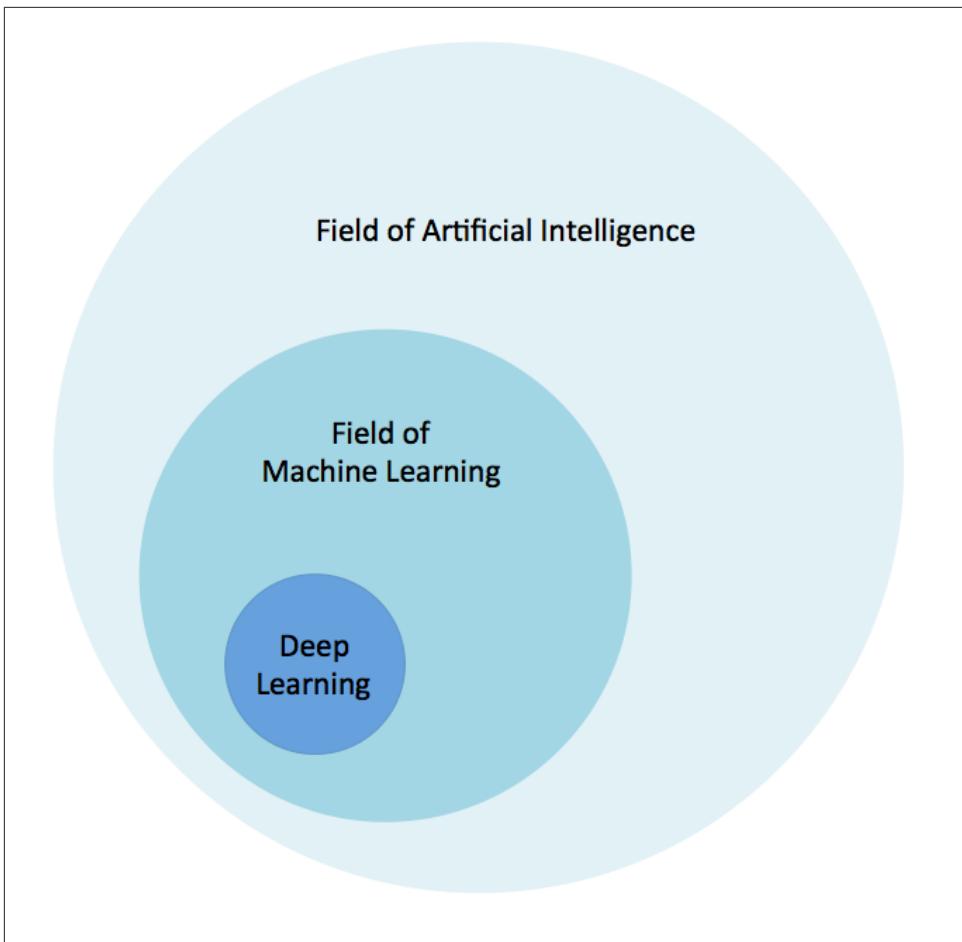
field of study that gives computers the ability to learn without being explicitly programmed.

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 grandfather of Artificial Intelligence, Peter Norvig, when he wrote in the book “Artificial Intelligence: A Modern Approach”:

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

As Norvig alluded to in the earlier quote many of these ideas around how to model learning were inspired from processes and algorithms discovered in nature. To set Deep Learning in context visually, we illustrate the relationship between Artificial Intelligence, Machine Learning, and Deep Learning in the diagram below.



*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. This book explores the relationships between these fields. Let's now take a quick detour into another tangential subject in 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 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 an artificial neuron (or a 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
- bees
- genetic algorithms

Ant colonies, 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. Some definitions today will define deep learning as 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 focus our content on the definition above as we feel that neural networks had to transcend architecturally from the earlier network styles (in conjunction with a lot more processing power) before the results became bleeding-edge 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:

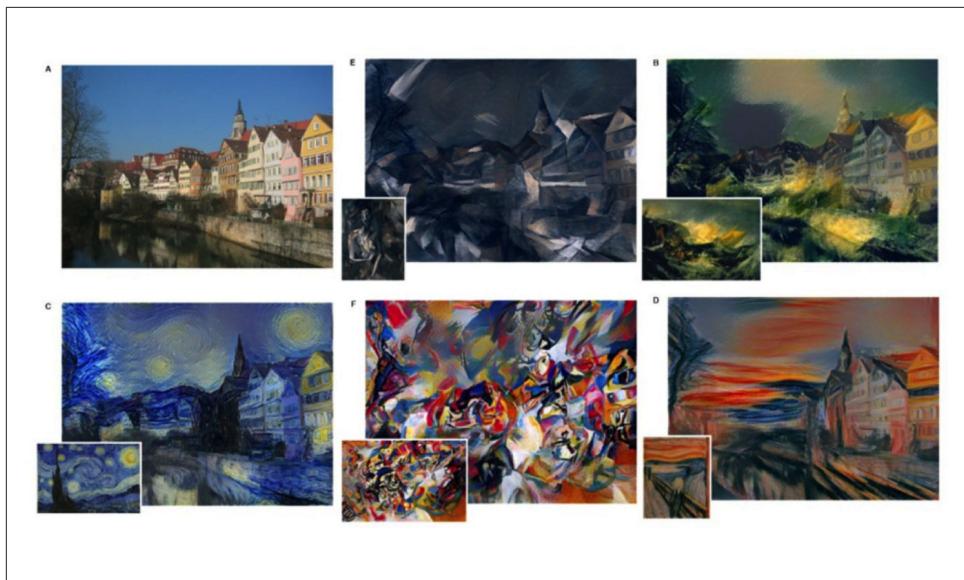
- Unsupervised Pre-Trained Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

There are some variations of the above architectures where we'd see a hybrid convolutional and recurrent neural network, 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 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<sup>1</sup>*

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?”

---

<sup>1</sup> Gatys (et. al, 2015) “A Neural Algorithm of Artistic Style” <http://arxiv.org/pdf/1508.06576v1.pdf>

## Organization of This Book

We take the first half of this book to understand the core concepts and some light machine learning theory for the casual practitioner and then later on apply those concepts in building deep learning networks. From there we take the practitioner through the exercise of building, tuning, and understanding the application of deep learning with the deeplearning4j software package to enable the practitioner to attempt today's more daunting machine learning tasks.

This chapter is designed as a general primer on machine learning and then neural network fundamentals. It's meant as a way to get the newcomer going with just enough basics but the more seasoned data scientist may want to skip it. In chapter two we take the reader through the specific fundamentals of deep networks and then dive into the details of the four major architectures. We'll build the reader's skills at understanding how to apply each architecture and then in chapter 3 we'll take the reader through using deeplearning4j's API to put these ideas into code. In chapter 4 we'll then refine this understanding with a better understanding of how to tune these architectures and develop a methodology. In chapter 5 we'll take a look at the topic of vectorization and getting different data types ready to be modeled. In chapter 6 we'll conclude with a look at using the command line to work with deeplearning4j and using spark with deeplearning4j.

## 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.

# 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"

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

## Scalars

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

In computing the term scalar is synonymous with the term variable and is a storage location paired with a symbolic name. This storage location holds an unknown quantity of information called a "value".

## Vectors

For our use we define a vector as:

*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:

$$X = (x_1, x_2, \dots, x_n) \text{ (note: written vertically)}$$

Or

$$X = (x_1, x_2, \dots, x_n)$$

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.

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

Figure 1-3. A 3x3 Matrix

If our matrix is said to be a  $n \times m$  matrix then it has  $n$  rows and  $m$  columns. In the figure-XX above we see a  $3 \times 3$  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.

Hyperplanes come up in machine learning in linear modeling when we discuss the concept finding an n-dimensional separating plane that best fits the training data. 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 operation takes two vectors of scalars of equal length and returns a single number representing the summation of all the multiplication of the matching scalar entries in both vectors.

### 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.

### Outer Product

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

## Converting Data Into Vectors

In the course of working in machine learning and data science we need to analyze all types of data. A key requirement is being able to take each data type and represent it as a vector. In machine learning we 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:

```
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", 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 2:0.4166666666666663 3:0.702127659574468 4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.4583333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.583333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.750000000000002 3:0.6808510638297872 4:0.5652173913043479
1.0 1:0.9166666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0.5652173913043479
0.0 1:0.0833333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.958333333333335 2:0.750000000000002 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 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 5.

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	0.4166666666666663	0.702127659574468	0.5652173913043479
0.6666666666666666	0.5	0.9148936170212765	0.6956521739130436
0.4583333333333326	0.3333333333333336	0.8085106382978723	0.7391304347826088

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

### What is a Feature?

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

An example would be a column of input that 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

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

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

## Methods for Solving Systems of Linear Equations

There are two general methods for solving a system of linear equations. The first is called the “direct method” 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.

## Some Basic 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 uncertainty 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.  $0.35 \times 100 == 35\%$ ).

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:

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

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

$$4/52 = 0.077$$

Conversely, odds are defined as:

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

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

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

The primary difference here is the choice of denominator (Total Chances 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.

### 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 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.

## 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:

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

Sometimes we'll hear the second event F referred to as the "condition". Conditional probability is interesting in machine learning and deep learning 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:

$$P(A|B) = \frac{P(B | A)}{P(A)P(B)}$$

## Posterior Probability

In Bayesian statistics we call the posterior probability of the random event the conditional probability we assign after the evidence is considered. Posterior probability distribution is defined as the probability distribution of an unknown quantity conditional on the evidence collected from an experiment treated as a random variable. We see this concept in action 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.

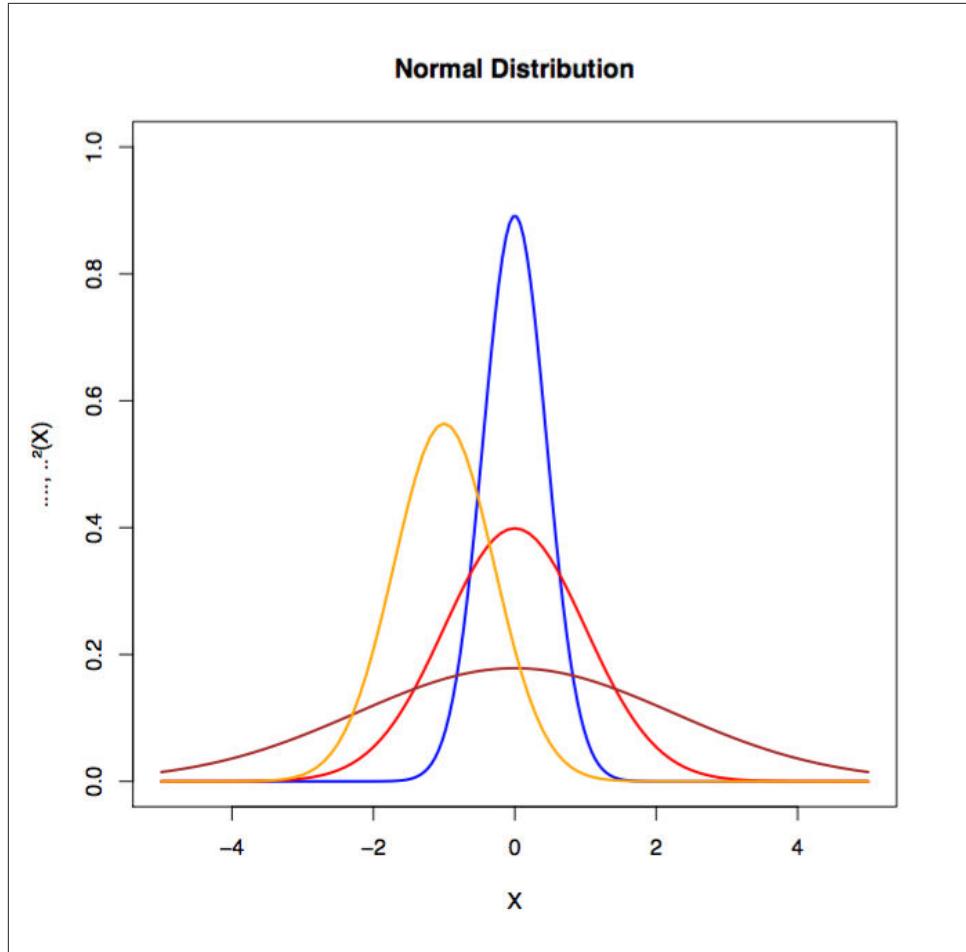


Figure 1-5. Examples of Normal Distributions

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 though 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.

## 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 ( $b\hat{}$ , 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

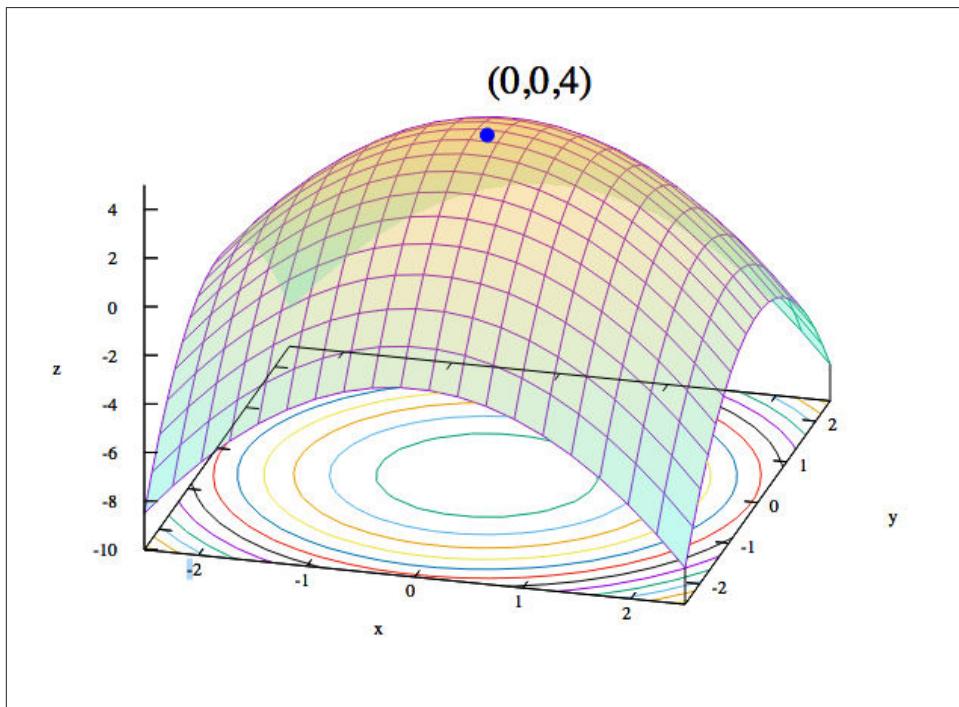
- transform input to help determine the classifications a network infers
- loss function
  - gauges how well it classifies (minimizes error) at each step
- optimization function
  - guides it toward the points of least error.

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 1-6. 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 you're training a network to classify a flower species you may know that petal length is important, but color isn't). Or if you're letting the network do all the work, you might choose the initial weights randomly.

The purpose is to move that weight downhill, to areas of lower error, as quickly as possible. An optimization algorithm like gradient descent can sense the actual slope of the hills with regard to each weight; 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.

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 known as the “stationary point” of the function or the minimum point. In optimization we consider optimizing a function to be minimizing a function (outside of inverting the cost function).

### What is Gradient?

Gradient is defined as the generalization of the derivative of a function in one dimension to a function  $f$  in several dimensions. It is represented as a vector of  $n$  partial derivatives of the function  $f$ . It is useful in optimization in that the gradient points in the direction of the greatest rate of increase of the function 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.

## 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 (?). Some examples include:

### The Jacobian and the Hessian

The Jacobian is a  $m \times n$  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 matrices much in this book but we want the reader to be aware of them and their place in the wider scope of the machine learning landscape.

## Underfitting and Overfitting

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

If the line fits the data too well, we have the opposite problem called "overfitting". Solving underfitting is the priority, but much effort in machine learning is spent attempting not to overfit the line to the data. When we say a model overfits a dataset we mean 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 overfitting is by thinking about probable distributions of data. The training set of data that we're trying to draw a line through is just a sample of a larger unknown set, and the line we draw will 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.

## 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:

$$y = a + b_0 * x_0 + b_1 * x_1 + \dots + b_N * x_N$$

A simple example of a problem that linear regression solves would be predicting how much you'll spend per month on gasoline based on the length of your commute. Here, what you pay at the tank is a function of how far you drive. Your gas costs are the dependent variable and the miles you commute are the independent variable. It's reasonable to keep track of these two quantities and then define a function

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

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

Section content goes here

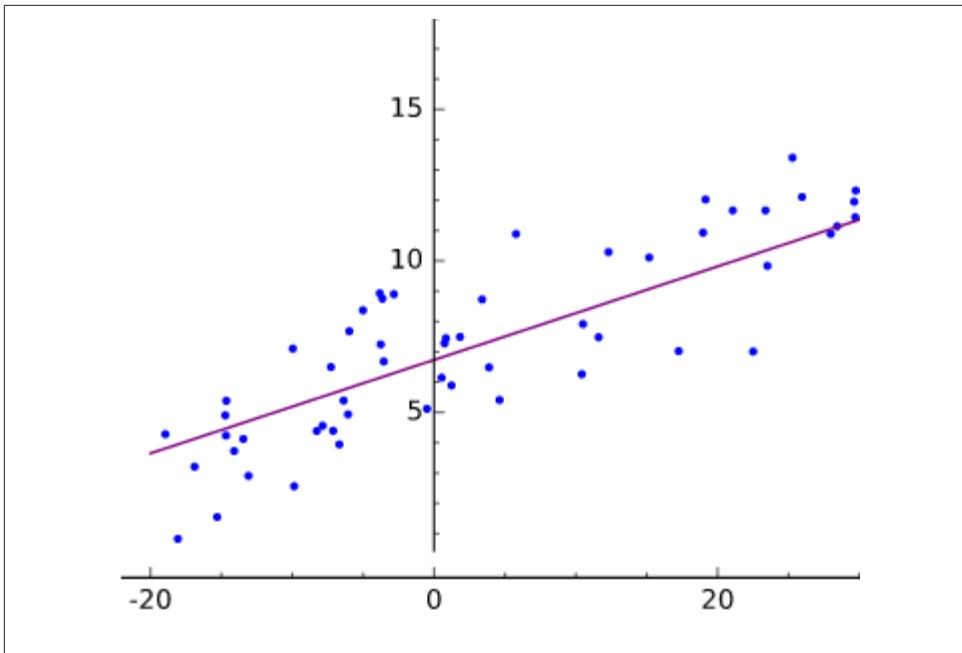


Figure 1-7. Linear Regression Plotted

Visually linear regression can be represented as finding a line that comes close to as many points as possible in a scatterplot of data. *Fitting* is defining a function  $f(x)$  that produces  $y$ -values close to the measured or real-world  $y$  values. The line produced by  $y = f(x)$  comes close to the scattered coordinates, the pairs of dependent and independent variables.

### Relating the Linear Regression Model

We can relate this function to the 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 your training set, and has almost no predictive power beyond the data it has seen (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 certainty. 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 if 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.

## 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
  - $f(x) = 1 / (1 + e^{(-\theta^T x)})$
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:

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

This function is useful in logistic regression as it takes any input on the range of negative infinity 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.

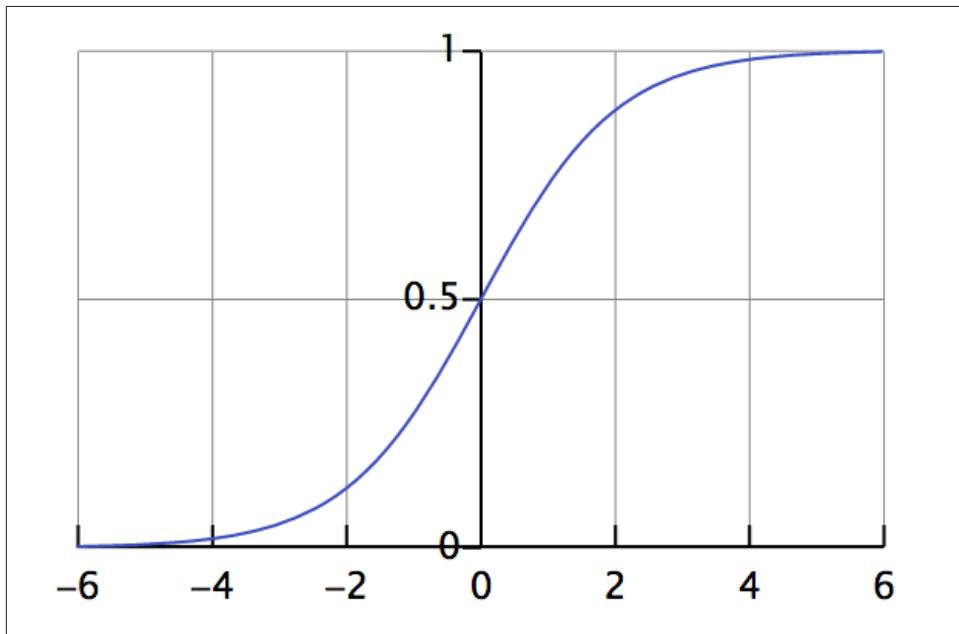


Figure 1-8. Plot of the Logistic Function

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  $\sigma$ , 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 negative number approaches zero; in contrast, the more  $x$  decreases below zero, the more the expression  $(1 + e^{-x})$  grows, shrinking the entire quotient. Since  $(1 + e^{-x})$  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-XX 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 1-9. 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 in example X:

```
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.

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

If our model was to classify a patient from the previous example as not having the condition and 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.

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

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.

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

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

## Precision

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

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

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

## Recall

This is the same thing as sensitivity 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.

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

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:

<https://physionet.org/challenge/2012/>

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

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

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. In chapter 4 we’ll look at ways to deal with class imbalance and understand when our model is over-fitting on the training data but not generalizing well on the important parts of the problem.

# Foundations of Neural Networks

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 figure-XX:

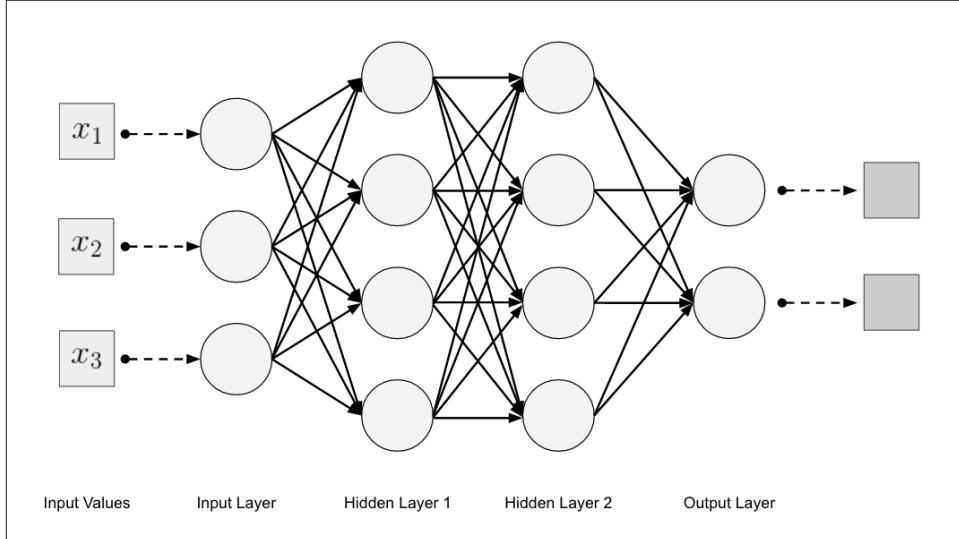


Figure 2-1. Multi-Layer Neural Network Topology

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 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.

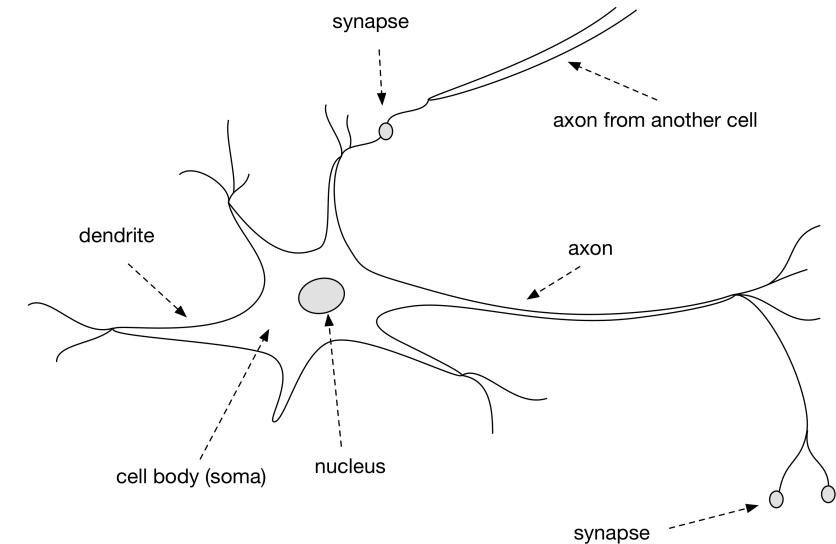


Figure 2-2. The Biological Neuron

In figure-XX 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. 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 and 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 more.

## History of the Perceptron

The perceptron was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. 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.

It should also be noted that McCulloch and Pitts introduced the basic concept of analyzing neural activity in 1943 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-XX. 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.

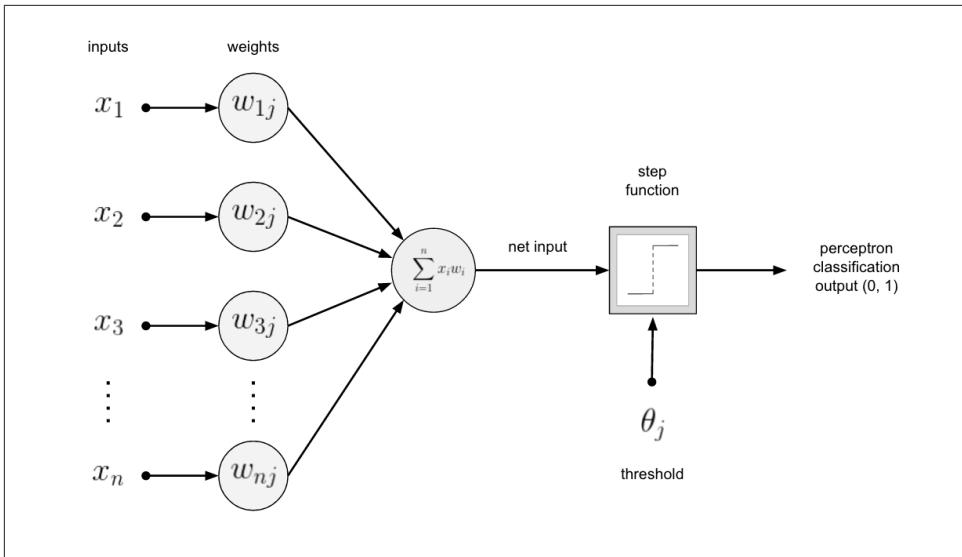


Figure 2-3. Single Layer Perceptron

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

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

Figure 2-4. Heaviside Step Function

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-XX 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

Function Parameter	Description
w (dot) x	dot product ( sum of $w_i * x_i$ )
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.

### 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-XX 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 sep-

arable 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 2-5. 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 illustrated the limitations of the single 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 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-XX we see an updated diagram for the artificial neuron that is based on the perceptron:

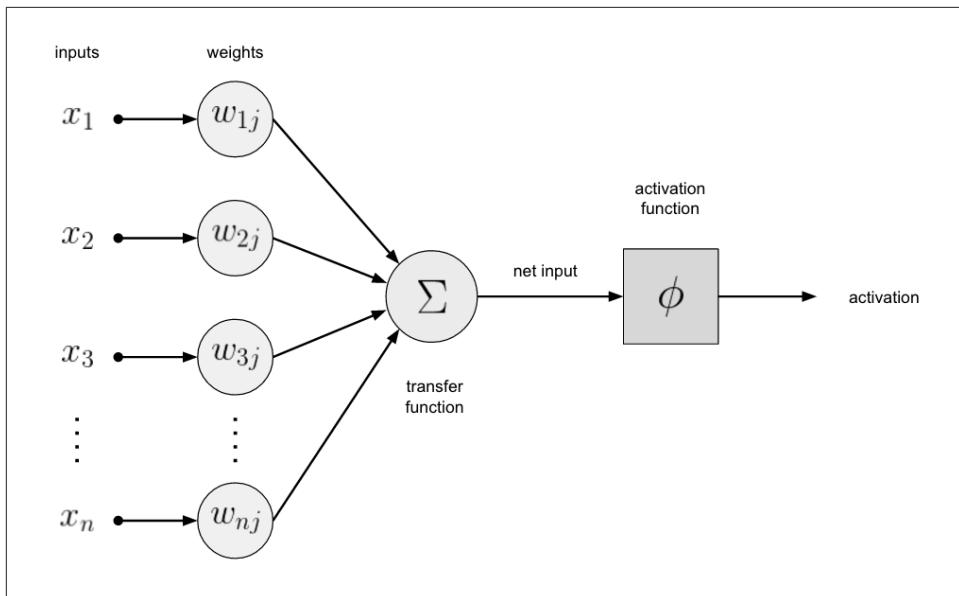


Figure 2-6. Artificial Neuron for a Multi-Layer Perceptron

This diagram is similar to Figure-XX for the Single Layer Perceptron yet we notice a more generalized activation function.



#### 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.

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:

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

where  $\mathbf{W}_i$  is the vector of all weights leading into neuron i and  $\mathbf{A}$  is the vector of activation values for the inputs to neuron i. Mathematically we can express the output of an artificial neuron as:

$$h_{w,b}(x) = f(\mathbf{w}^T \mathbf{x})$$

where  $w$  represents the weight vector (parameter vector),  $b$  represents the bias term, and  $x$  represents the set of input features. The  $T$ -symbol represents the dot product between the weight vector and the  $x$  vector of input features. We can express this differently with the notation:

$$h_{w,b}(x) = g(\sum w_i x_i + b)$$

This is slightly different than our previous equation above. We show this alternate notation to illustrate how some papers will use slightly notations to explain these concepts and we want the reader to be able to recognize the variants. In the equation below we expand this notation to express net input to the activation function for neuron i:

$$= g(\sum w_i x_i + b)$$

Where  $g$  is our activation function. If our activation function is the sigmoid function then we have

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

This output will have the range [ 0, 1 ] which is the same output as the logistic regression function. 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. The most common activation function for artificial neurons in multi-layer perceptrons is the sigmoid activation function.

Activation functions transform the combination of inputs, weights and biases. Products of these transforms are input for the next node layer. Most (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

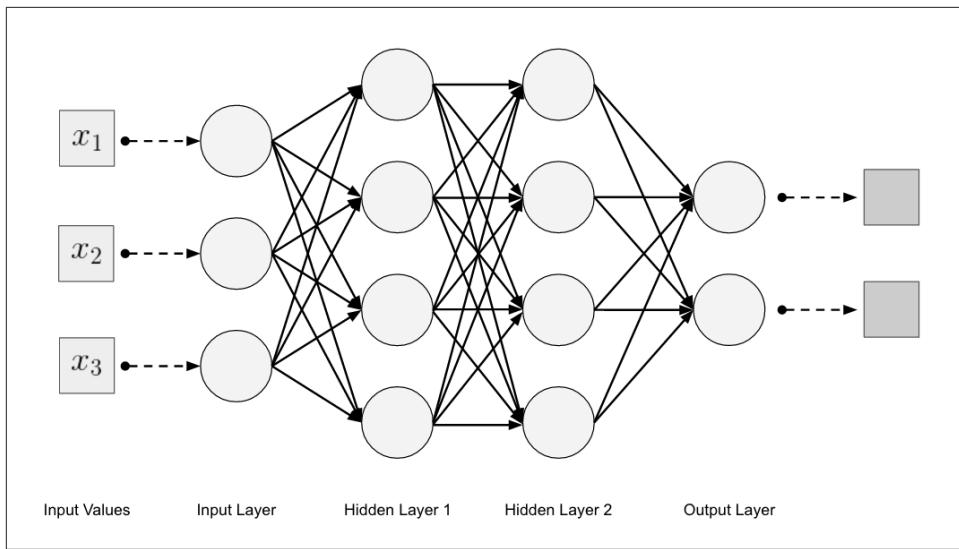
The authors would note (again) that the biological neuron is still more complex than the artificial variant. Research continues towards a better understanding of the biological neuron's function.

## Feed-Foward 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-XX 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 connection 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. The hidden layers in a typical multi-layer perceptron neural network will use a sigmoid activation function.

**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-forward network the connections between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the next layer. We change these weights progressively as our algorithm finds the best solution it can with the backpropagation learning algorithm. We can understand the weights mathematically by thinking of them as the parameter vector in the earlier linear algebra section describing the machine learning process as optimizing the parameter vector (e.g. "weights", here) to minimize error. We've now covered the basic structure of feed-forward neural networks. The rest of this chapter 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. Optimization algorithms, such as stochastic gradient descent, reward the network for good guesses and penalize it for bad ones. Lacking absolute certainty, neural networks deal in probabilities as they train.

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-forward 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.

## 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 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.

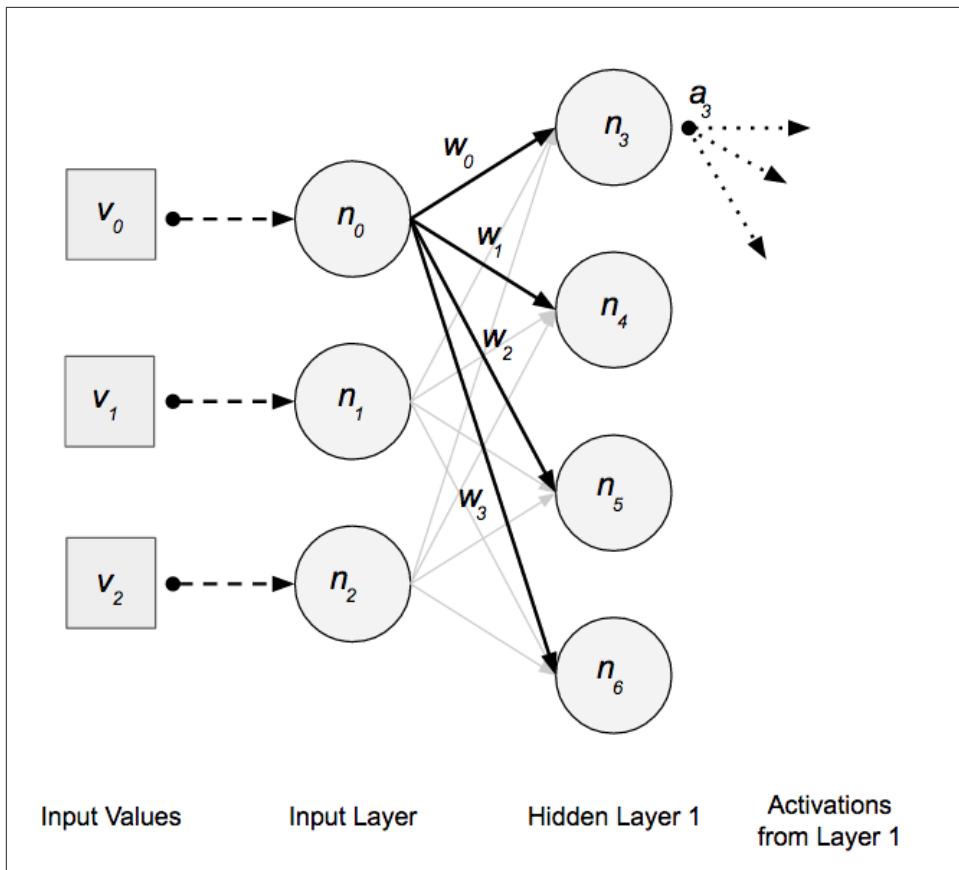


Figure 2-8. Multi-Layer Perceptron Network Zoomed-In with Component Labels

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 incoming connection in network to neuron $i$
$a_i$	Activation value of neuron $i$ (output of neuron $i$ )
$A_j$	vector of activations values for the inputs into neuron $i$

Notation	Meaning
$g$	activation function
$g'$	derivative of the activation function
$\text{Err}_i$	difference between the network output for unit $o$ for an input and the actual label value
$W_i$	vector of weights leading into neuron $i$
$\text{in}_i$	weighted sum of inputs to neuron $i$
$\alpha$	learning rate
$I_i$	vector of input from layer $i$

To further set the stage for explaining this algorithm let's take a look at the pseudo-code 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 + learning-rate * a_j * Err_e_i * g'( in_i )

            for each subsequent-layer in network do

                // compute the error at each node
                delta_err_j <- g'( in_j ) Sum_i( W_j_i delta_i )

                // update the weights leading into the layer
                W_k_j <- W_k_j + learning-rate * I_i * delta_err_j

            end for

        end for
    end loop when network has converged
    return network

```

### **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.** We can see the output layer weights update being calculated with:

```
W_j_i <- W_j_i + learning-rate * a_j * Err_e_i * g'( in_i )
```

Breaking this line down further we see we are working with the neuron j weight from the previous hidden layer connecting to neuron i. We're multiplying the learning-rate (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:

$in_j = W_j \cdot (dot\text{-}product) A_j$

Then we'd compute the activation of neuron j:

$a_j <- g( in_j )$

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:

$g'( in_i )$

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

**Further Expressing the Error Term.** We saw in the previous section where  $g'( z )$  gave us the derivative of the activation function. The error term is commonly expressed as delta\_i giving us:

$\delta_i = Err_i \cdot g'( in_i )$

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

```
W_j_i <- W_j_i + alpha * a_j * delta_i
```

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

**The New Propagation Rule for the Error Value.** The propagation rule for the delta values now becomes the following:

```
delta_err_j <- g'(in_j) Sum_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. 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. We then add this to the previous value for the weight and this is our update value:

```
W_k_j <- W_k_j + learning-rate * I_i * delta_err_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.

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.

## 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  $f(x) = Wx$ ; i.e. the dependent variable has a direct, proportional relationship with the independent variable. As such, linear transforms resemble linear regression. In practical terms, it means the node passes the signal through as is. Linear transforms are commonly used with single-layer perceptrons.

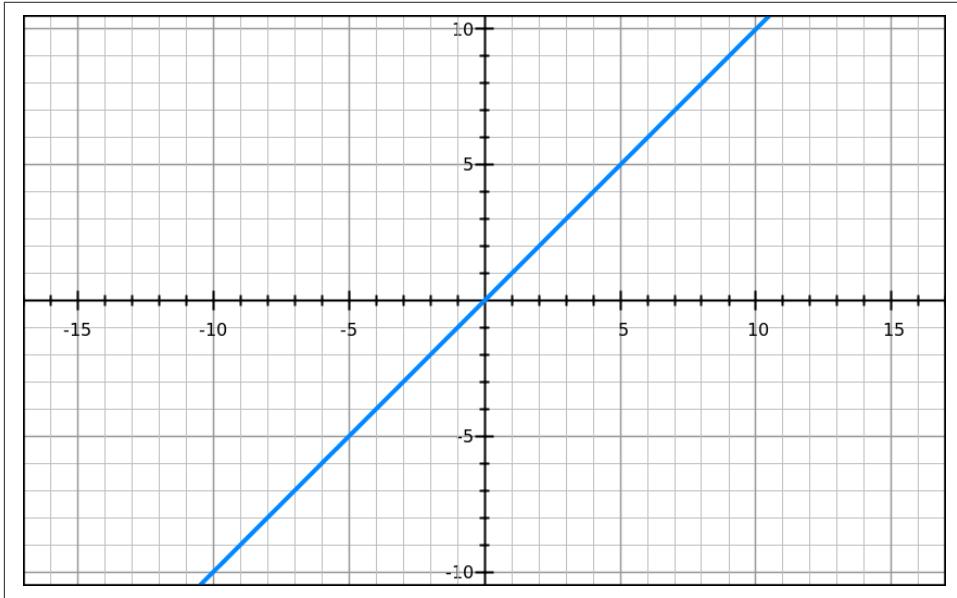


Figure 2-9. Linear Activation Function

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.

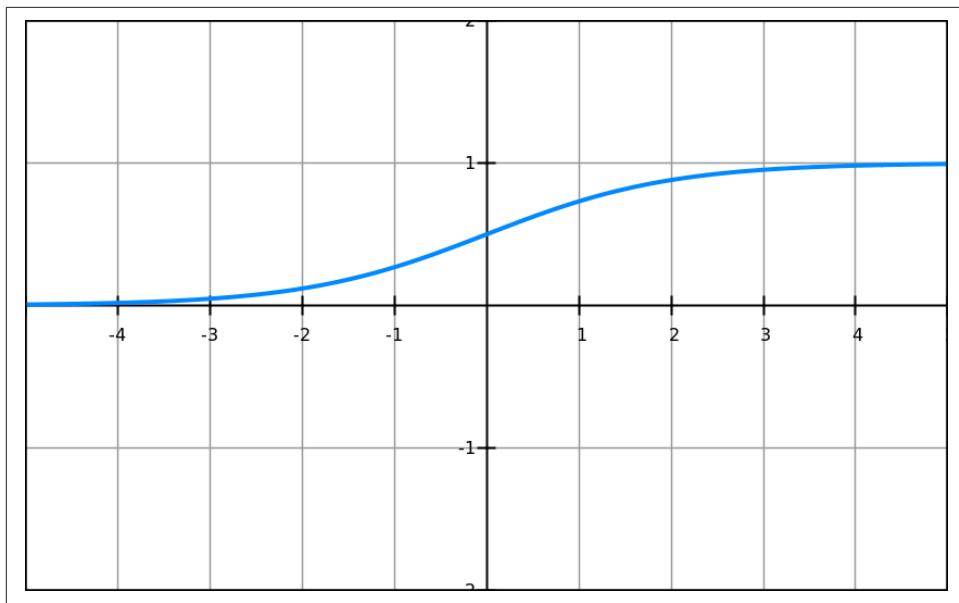


Figure 2-10. Sigmoid Activation Function

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.

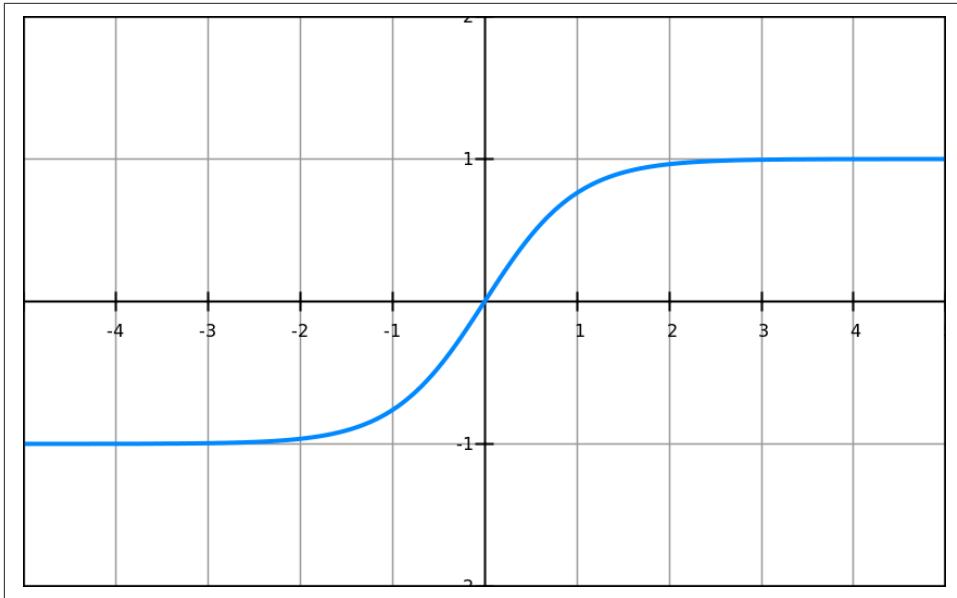


Figure 2-11. Tanh Activation Function

## 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. 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.

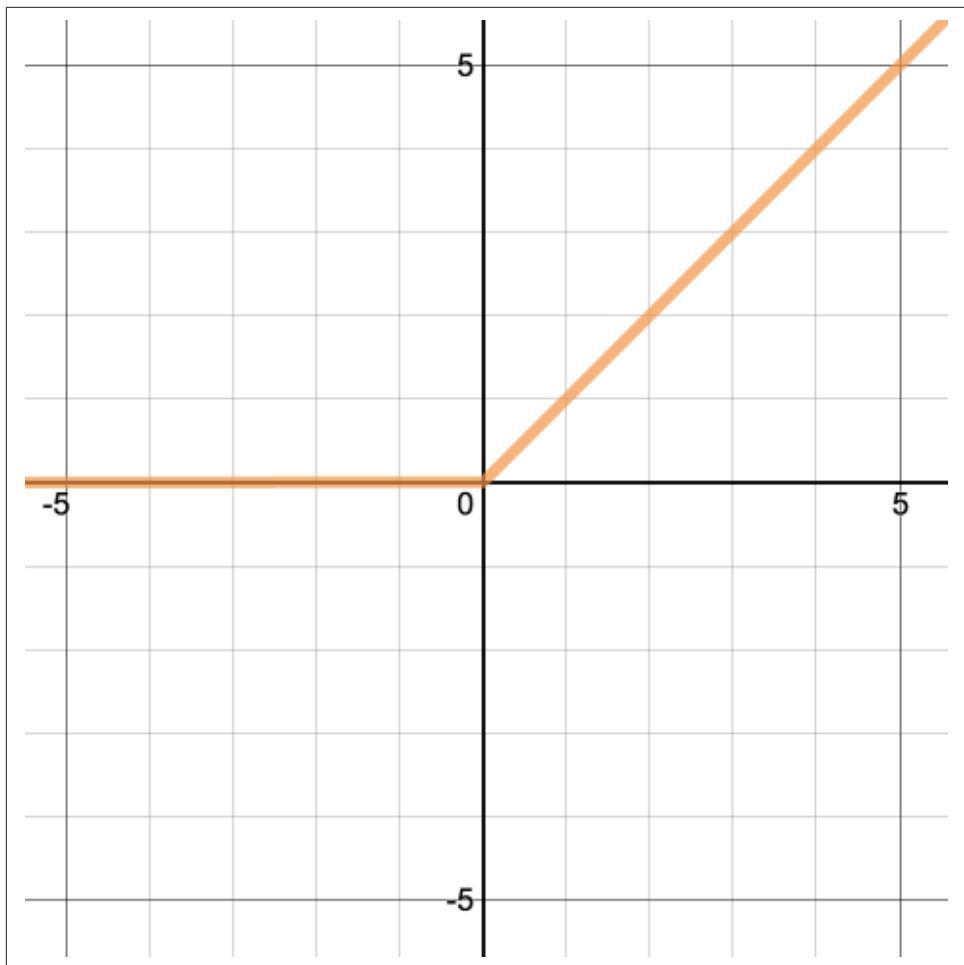


Figure 2-12. Rectified Linear (ReLU) Activation Function

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 have shown to train better in practice than sigmoid activation functions.



## The Unreasonable Effectiveness of ReLU Activation Functions

Compared to the Sigmoid and Tanh activation functions, the ReLU activation function does not suffer 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.

# 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 net's predictions. We then aggregate these errors over the entire dataset and average them and now we have a single number representative of how close the neural network is to its ideal.

Looking for this ideal state is equivalent to finding the parameters (weights and biases) that will minimize the “loss” incurred from the errors. In this way loss functions help reframe training neural networks as an optimization problem. In most cases these parameters cannot be solved for analytically but more often than not they can be approximated well with iterative optimization algorithms like gradient descent. The 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  $h(X_i) = \hat{Y}_i$ , to denote the neural net transforming the input  $X_i$  to give the output  $\hat{Y}_i$ . We will alter this notation a little later to emphasize its dependence on weights and biases.

- When referring to 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  $y_{ij}$  refers to the jth feature observed in the ith sample collected.
- We will represent the loss function by  $L(W,b)$ .

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 your losses wiggle. Wiggle those for a given input and your output wiggles.

So our notation  $h(X) = y_{\text{hat}}$  should be conditioned on a set of weights and biases and so we will amend our notation to say  $h_w,b(X) = y_{\text{hat}}$ . 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 datapoints, plain and simple.

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

*Figure 2-13. Mean Squared Error (MSE) Loss Equation*

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  $y_{\text{hat}}$  respectively, are an ordered list of numbers or in other words, vectors. The loss function boils down the difference between desired and predicted, be that they are vectors, into a single number.

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

Figure 2-14. MSE #2 Loss Equation

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 net 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 back-propagation)

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

Figure 2-15. DL4J MSE Equation

Optimizing the mean squared error is equivalent to optimizing for the mean. One of the advantages of the MSE is that it is a convex function and therefore will always find its way to a global minimum with enough time with small enough steps, though that is easier said than done for a host of reasons we will discuss later.

### Other Loss Functions for Regression

While the MSE is used very 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. It simply averages the absolute error over the entire dataset.

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

Figure 2-16. Mean Absolute Error (MAE) Loss Equation

**Mean Squared Log Error (MSLE) Loss.** The other loss functions used for regression are the mean squared log error (MSLE)

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

Figure 2-17. Mean Squared Log Error (MSLE) Equation

**Mean Absolute Percentage Error (MAPE) Loss.**

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

Figure 2-18. Mean Absolute Percentage Error (MAPE) Loss Equation

### Regression Loss Function Discussion

These are all valid choices and there certainly is no silver bullet that will thwart the loss 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 your net is predicting outputs that vary largely in range. Say a net 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).

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 built 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 eg, 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 the 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”.

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

Figure 2-19. Hinge Loss Equation

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 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 \$ 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 as shown below.

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

*Figure 2-20. Logistic Loss Eq. 1, Probability of 1*

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

*Figure 2-21. Logistic Loss Eq. 2, Probability of 0*

The above two equations can be combined and expressed as:

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

*Figure 2-22. Logistic Loss Equations Combined*

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.

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

*Figure 2-23. Logistic Loss: Product of the Probabilities*

**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-

ties. And hence the product of the probabilities transforms to the sum of the log of the probabilities. 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:

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

*Figure 2-24. Logistic Loss Rewritten as Negative Log Likelihood*

Extending the loss function from two classes to M classes give us the following equation.

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

*Figure 2-25. Logistic Loss Equation*

### 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.

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.

## Loss Functions for Reconstruction

This set of loss functions relate to what is called reconstruction. The idea is simple. A neural net 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 restric-

ted 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.

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

Figure 2-26. Equation for KL Divergence

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 chapters 3 and 4.

## 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. We'll see this in practice in chapter 3 and 4 with examples and tuning neural networks respectively.



# 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 setup the foundations of machine learning and general neural network. 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?

Deep learning has many meanings to many people but for the context of this book we're going to set some boundaries. Revisiting our definition of deep learning from chapter one, the facets we differentiate deep learning networks in comparison to 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 neu-

rons 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 a cambrian explosion of computing power to occur 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 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<sup>1</sup> 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

---

<sup>1</sup> [http://people.inf.elte.hu/lorincz/Files/RL\\_2006/SuttonBook.pdf](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 functions 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<sup>2</sup>. 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.

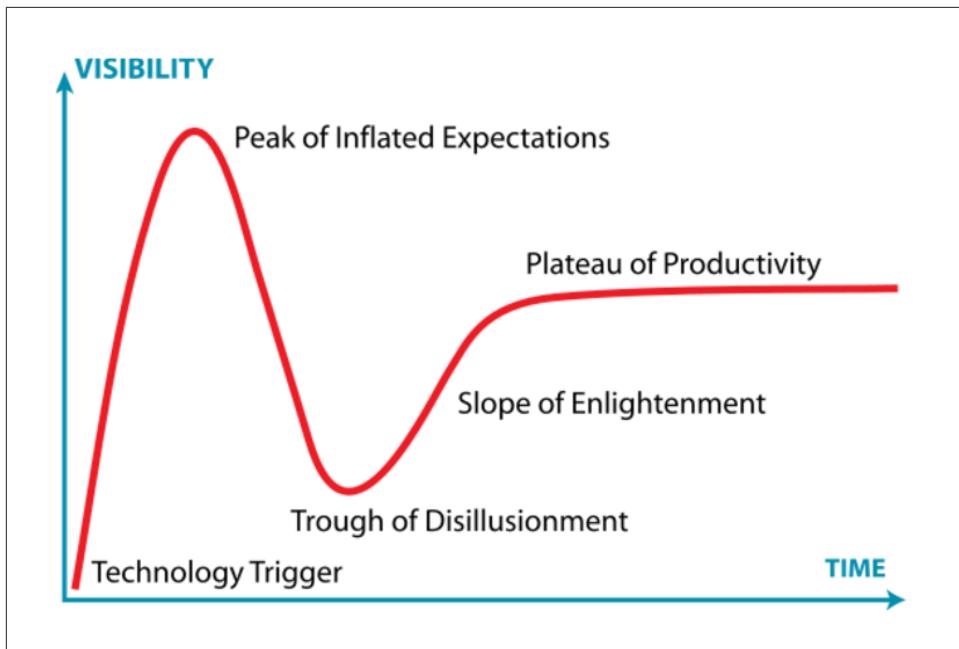
Deep Reinforcement Learning has risen in popularity over the course of production of this book and hopefully we'll see it included in a future edition. For now we'll leave the reader with an example in the Appendix of RL4J, the Reinforcement Learning sub-project in the DL4J suite of tools.

## 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-XX below) there were many lines of research still doing important research in the realm of neural networks.

---

<sup>2</sup> <http://www.nature.com/nature/journal/v518/n7540/abs/nature14236.html>



*Figure 3-1. Trough of Disillusionment<sup>3</sup>*

One important development in neural networks was Yann LeCun's work at AT&T Bell Labs on optical character recognition. 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<sup>4</sup> (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<sup>5</sup>.

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

<sup>3</sup> Diagram from: [https://en.wikipedia.org/wiki/Hype\\_cycle](https://en.wikipedia.org/wiki/Hype_cycle)

<sup>4</sup> <http://yann.lecun.com/exdb/mnist/>

<sup>5</sup> <http://image-net.org/>

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 apply these advances in domains such as:

- Self Driving Cars
- Google Translate
- Amazon Echo
- AlphaGo

Self driving cars in the 2006 Darpa Grand Challenge<sup>6</sup> 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

The 2012 AlexNet is hailed as an advancement in computer vision. 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 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.

---

<sup>6</sup> <http://archive.darpa.mil/grandchallenge/>

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 has 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. 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 Boltzmann Machines as layers in pre-training to build features. Convolutional Neural Networks used new and different types of activation functions in layers and changed how we connected layers (from fully connected to locally connected patches). Recurrent Neural Networks explored the use of connections that better modeled the time domain in modeling 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.

## Automated Feature Extraction

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<sup>7</sup>

---

<sup>7</sup> <http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/facebook-ai-director-yann-lecun-on-deep-learning>

Geoffrey Hinton talks about this theme in Deep Belief Networks when he explains<sup>8</sup> 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 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

---

<sup>8</sup> <https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>

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 the reader progresses through this chapter we’ll get a better sense on 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 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).

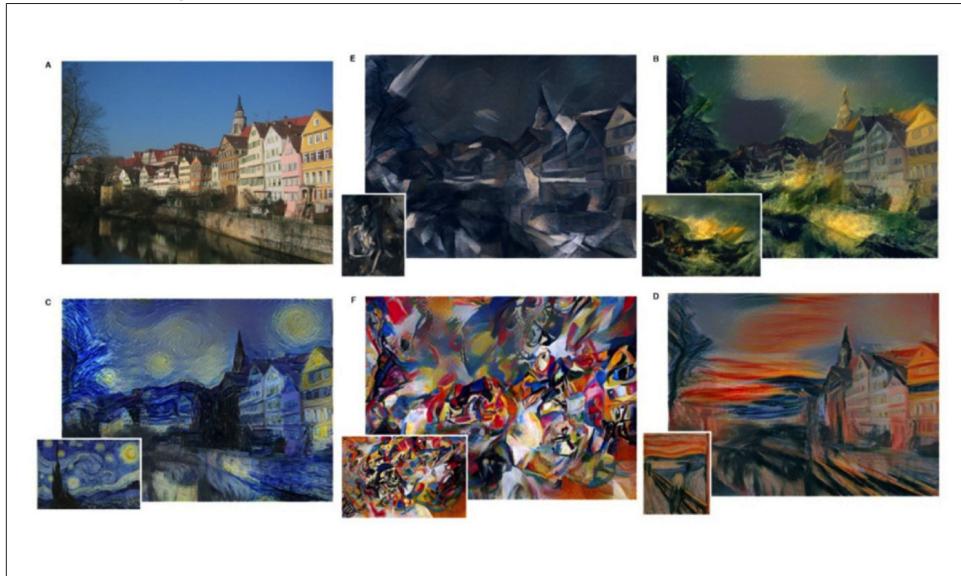


Figure 3-2. Stylized Images by Gatys et. al., 2015

Gatys (et. al, 2015) published a paper <sup>9</sup> 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 can best be described as synthesizing novel images by modeling the distributions of input data seen by the network.

**Recurrent Neural Networks.** Recurrent neural networks have been shown to model sequences of characters and generate new sequences that are lucidly coherent. In chapter 3 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:

<sup>9</sup> Gatys (et. al, 2015) “A Neural Algorithm of Artistic Style” <https://arxiv.org/abs/1508.06576>

*"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."<sup>10</sup>*

## 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 Principles 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

---

<sup>10</sup> <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 Deep-learning4j 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-XX (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 uses 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 what's called 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 relate 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) (plus 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 in the event that the ReLU did not achieve good results.



### 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 our 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 values) 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 argmax() 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 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 can be defined as a differentiable function, which approximates a given point  $x$  (a number in the parameter vector). This is the direct approximation, mapping a given input onto  $F$ , which is our function. [ review ]

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
- Conjugate gradient
- Hessian Free

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.

**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  $n \times n$  inverse matrix, Hessian L-BFGS stores only a few vectors that represent a local approximation of it. L-BFGS performs faster because it uses approximated second-order information. L-BFGS and conjugate gradient in practice can be faster and more stable than SGD methods.

**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 output. For the input layer this will match up to the number of features in the input vector. For the output layer this will either be a single output neuron or a number of neurons matching the number of classes we are trying to predict.

Deciding on neuron counts for each hidden layer is where hyperparameter tuning becomes a challenge. We can use an arbitrary number of neurons to define a layer and there are no rules about how big or small this number can be. However, how complex of a problem we can model is directly correlated to how many neurons are in the hidden layers of our networks. This 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.



### The Importance of Learning Rate as a Hyperparameter

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

**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:

- momentum
- RMSProp
- Adam
- AdaDelta

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.

**AdaGrad.** AdaGrad is one technique that has been developed to help augment finding the “right” learning rate. AdaGrad is named in reference to how it “adaptively” uses subgradient methods to dynamically control the learning rate of an optimization algorithm. AdaGrad is the square root of the sum of squares of a window of the most

recent gradient computations. AdaGrad speeds our training in the beginning and slows it appropriately towards convergence, allowing for a smoother training process.

**Other Learning Rate Advancements.** There are other techniques for feature wise learning rates. Adadelta for example is a variant of adagrad that only keeps the most recent history rather than accumulating it like adagrad does. 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 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, 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 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.

**Drop Connect.** Dropconnect 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 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 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 efficient 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 1 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

In chapter 1 we introduced the canonical feed-forward networks. Inspired by biological neural networks, 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 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 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<sup>11</sup> 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.<sup>12</sup> 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. Dr. Hinton has been a proponent of basic research and the long term persistence it takes to get results:

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

## Network Layout

There are five main parts of a basic Restricted Boltzmann Machine:

---

<sup>11</sup> <https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf>

<sup>12</sup> <https://www.cs.toronto.edu/~hinton/>

- 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 the diagram 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.

### A Symmetrical, Bipartite, Bidirectional Graph with Shared Weights

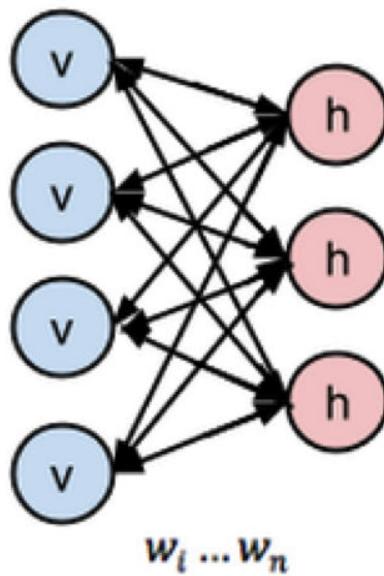


Figure 3-3. Restricted Boltzmann Machine

With Restricted Boltzmann Machines every visible unit is connected to every other 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. Restricted 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. 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-XX below we see a visual explanation of the network involved in reconstruction in restricted boltzmann machines.

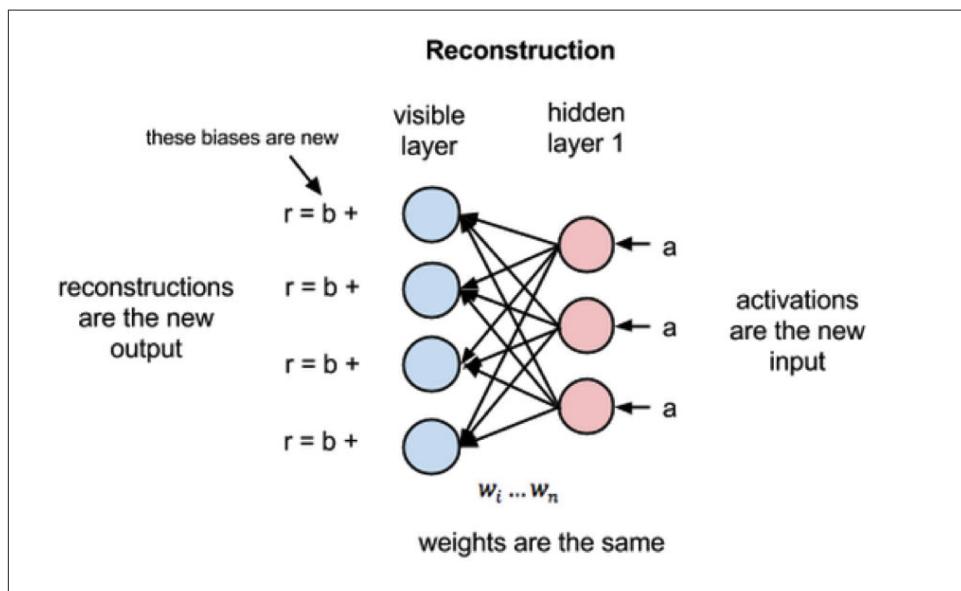


Figure 3-4. Reconstruction in Restricted Boltzmann Machines

We can visually explain reconstruction in Restricted Boltzmann Machines work by looking at the MNIST dataset<sup>13</sup>. 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.

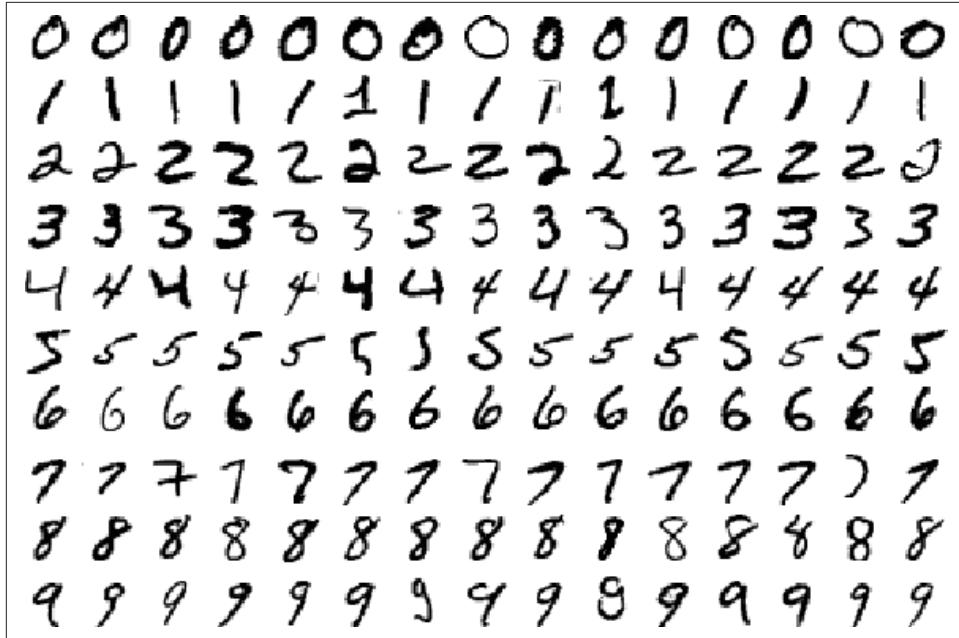


Figure 3-5. Sample of MNIST digits

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<sup>14</sup> 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:

---

13 <http://yann.lecun.com/exdb/mnist/>

14 <http://yosinski.com/media/papers/Yosinski2012VisuallyDebuggingRestrictedBoltzmannMachine.pdf>

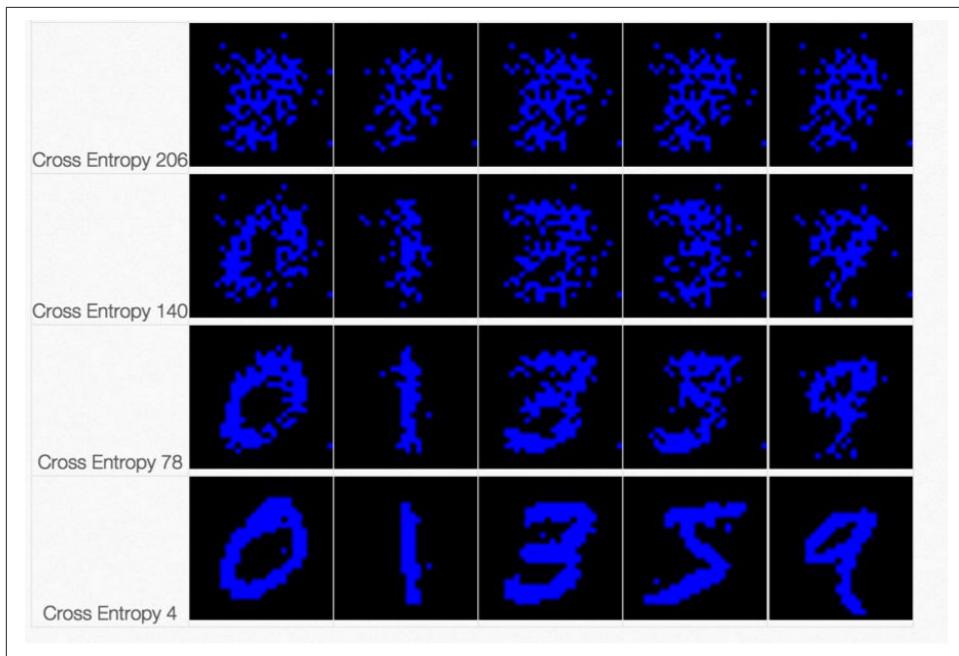


Figure 3-6. Reconstructing MNIST Digits with Restricted Boltzmann Machines

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

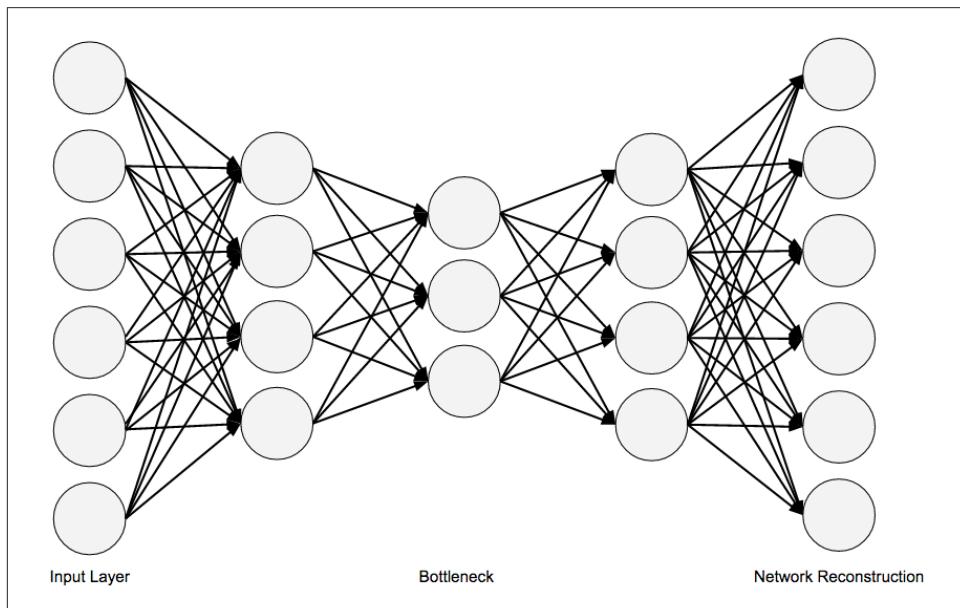
## Autoencoders

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-XX where we see an example of Autoencoder network:



*Figure 3-7. This is an image caption*

Beyond the output layer there are 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 unlabelled 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.** This is the architecture in the diagram above. The network input must pass through a “bottleneck” region of the network before being expanded back into the output representation.

**Denoising Autoencoders.** This 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 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.



# Major Architectures of Deep Networks

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.

## Unsupervised Pre-Trained Networks

In this group we cover two specific architectures:

- AutoEncoders
- Deep Belief 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.

## 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 X you can see a diagram showing the network architecture of a deep belief network:

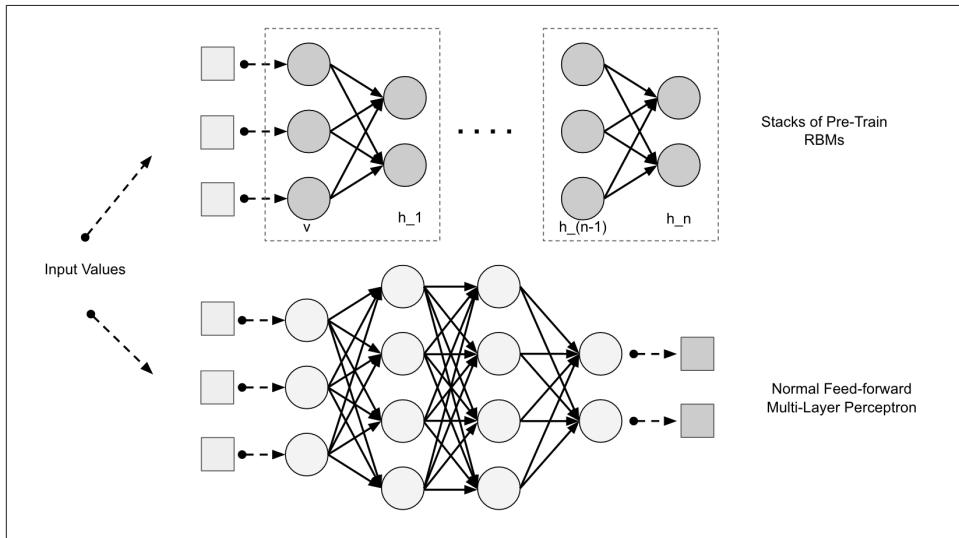


Figure 4-1. Deep Belief Network Architecture

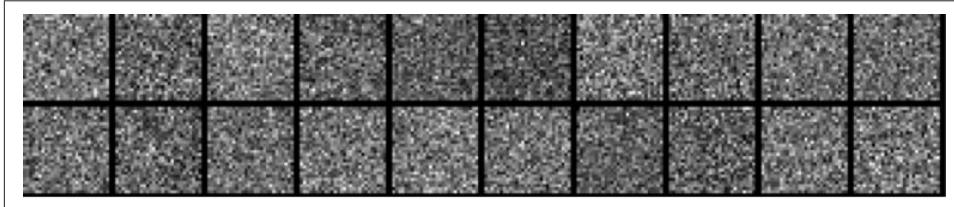
### Feature Extraction with Restricted Boltzmann Machine Layers

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 should 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.

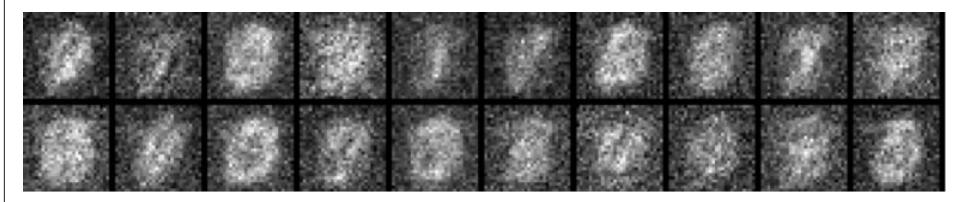
**Learning Higher Order Features Automatically.** Learning these features in an unsupervised fashion is considered the pre-train phase of deep belief networks. Each hidden layer of Restricted Boltzmann Machines in the pre-train phase learns progressively more complex features from the distribution of 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:



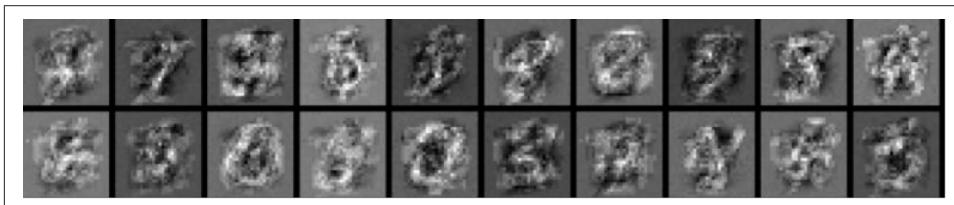
*Figure 4-2. Activation Render at Start of Training*

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 4. 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" backpropagation.

**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 labels.

### 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.

## 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 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 rotation invariant features from raw image data. They 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 generation and sentiment analysis. A convolution is a powerful concept for helping to build a more robust feature space based on a signal.

## Biological Inspiration

The biological inspiration for convolutional networks is the visual cortex in animals. 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

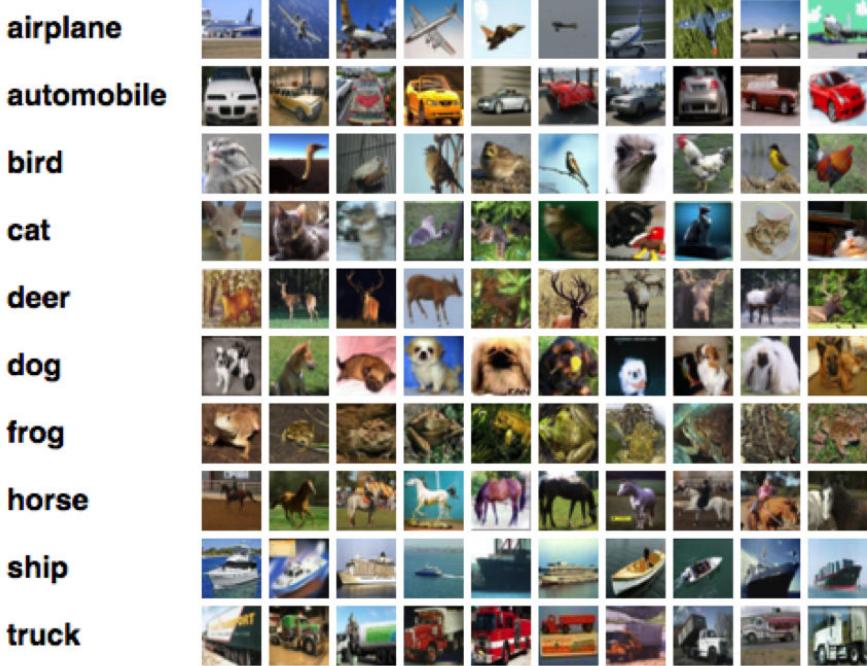
Feed-forward multi-layer neural networks take input as a single one-dimensional vector and transform the data with one or more hidden layers (fully connected). The network then gives a result from the output layer. The issue we run into with traditional multilayer neural networks and image data is these 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 multiply those weights as well.

### What is the CIFAR-10 Dataset?

The CIFAR-10 dataset is a well known image classification benchmark dataset<sup>1</sup> 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:

---

<sup>1</sup> <http://www.cs.toronto.edu/~kriz/cifar.html>



*Figure 4-5. 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 multilayer 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

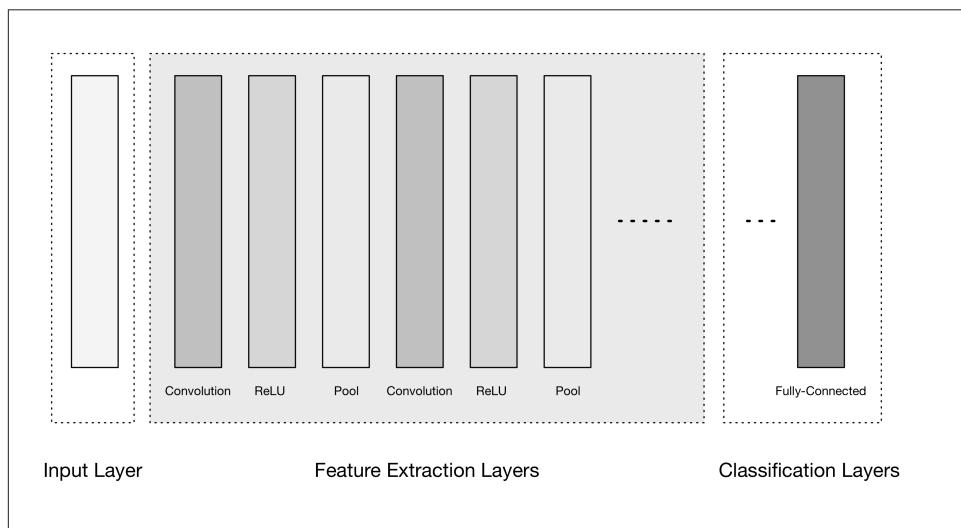
- The image width in pixels
- The image height in pixels

- 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 diagram XX:



*Figure 4-6. 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). 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  $[b \times N]$  where  $b$  is the number of examples in the mini-batch and  $N$  is the number of classes we're interested in scoring.

## Neuron Spatial Arrangements

Recall how in traditional multilayer neural networks the layers are fully connected and every neuron in a layer is connected to every neuron in the next layer. The neurons in 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.

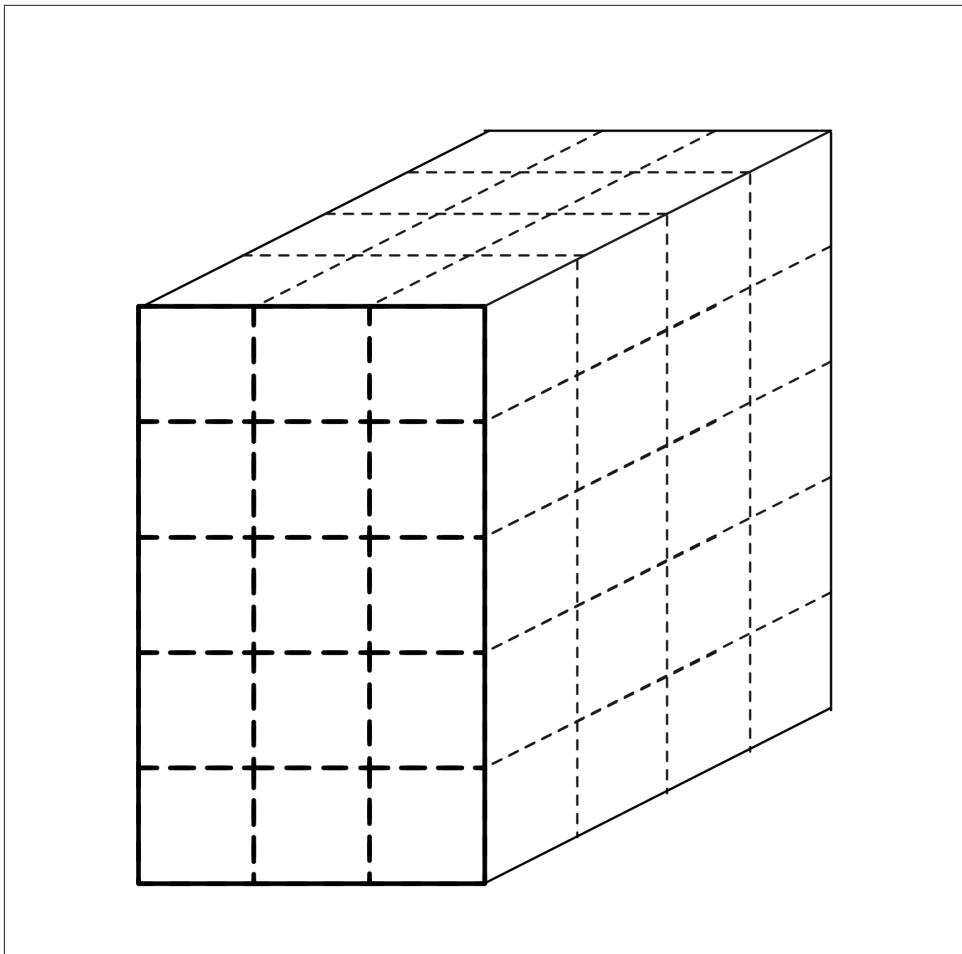
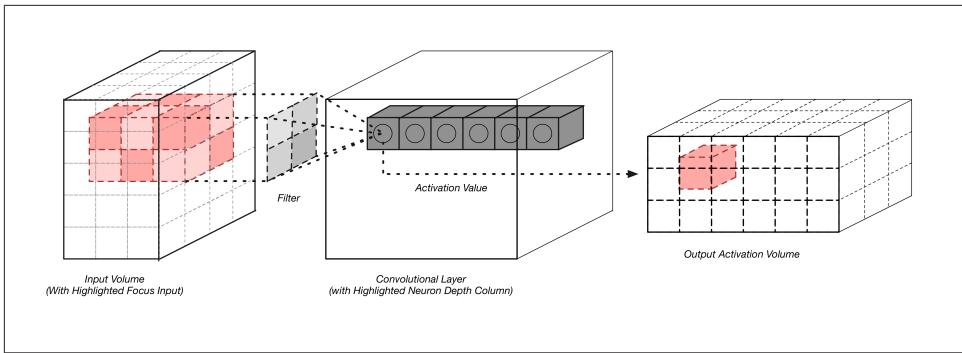


Figure 4-7. Input Layer 3D Volume

## 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 4-8. Convolution Layer with Input and Output Volumes*

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

## Convolution

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

The convolution operation 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).

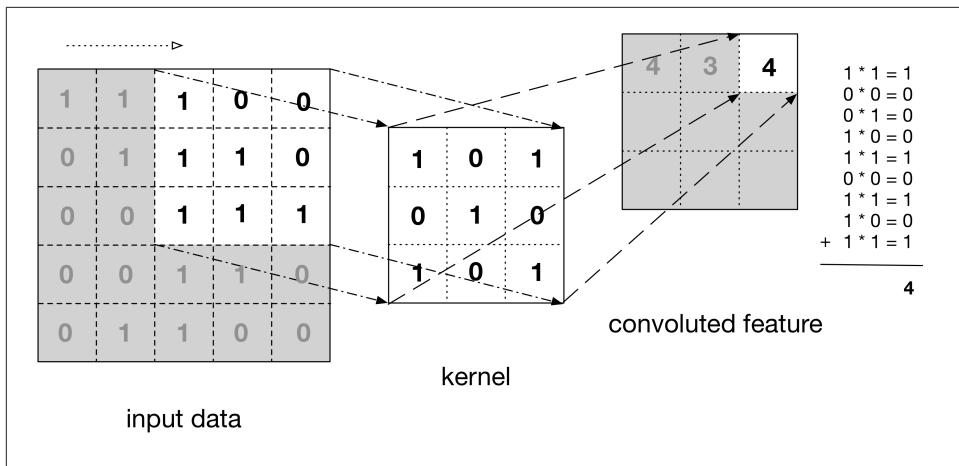


Figure 4-9. Convolution Operation

As we see in the diagram above, the kernel is slid across the input data to produce the convolved 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 diagram X. 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 the diagram below we can see how this activation map relates to our previously introduced concept of a convoluted feature.

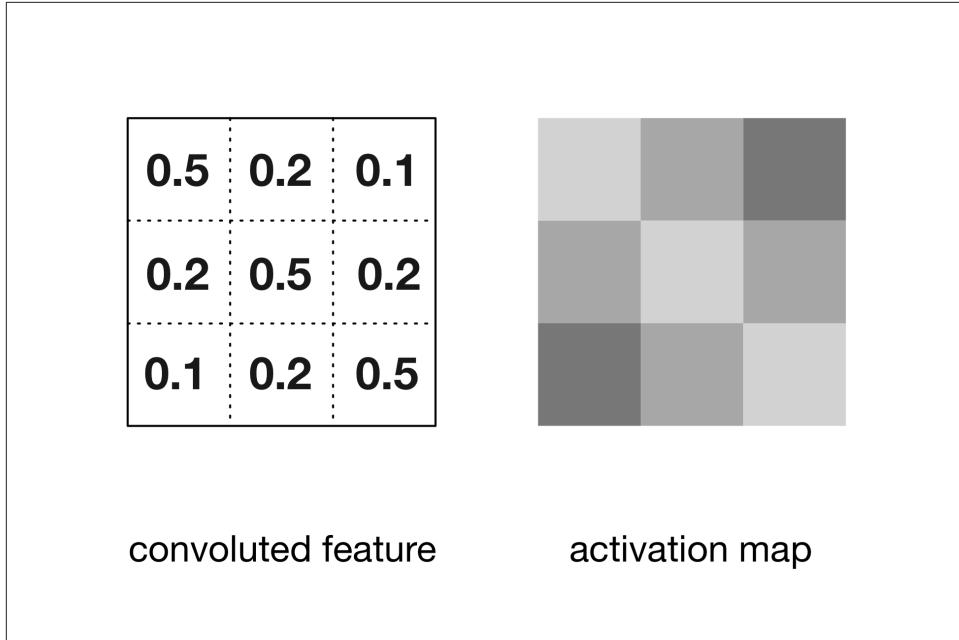


Figure 4-10. Convolutions and Activation Maps

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



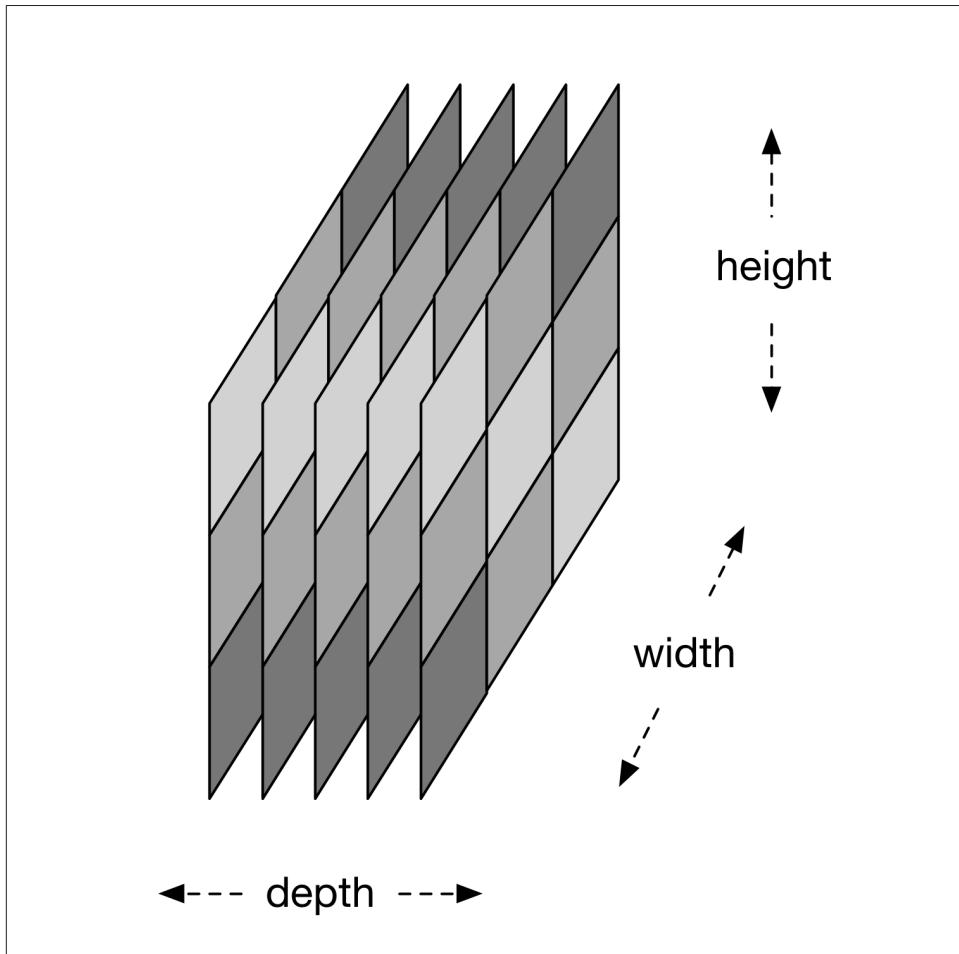
### Activation Maps

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 (window) and the input volume. Networks learn filters that activate when they see certain types of features in the input data in a specific spatial position.

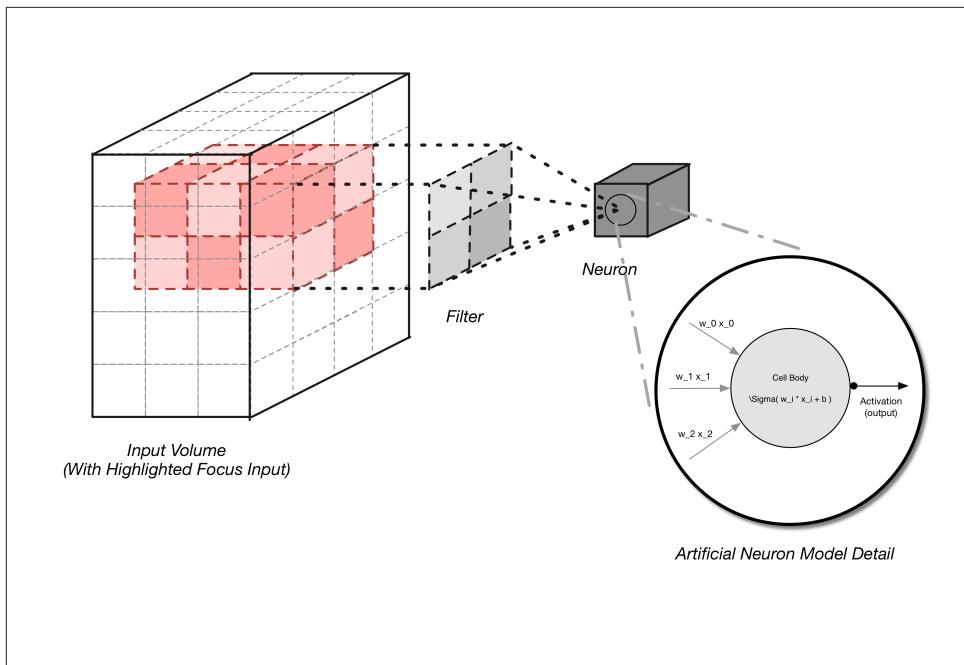
We create the 3-dimensional output volume for the convolution layer by stacking these activation maps along the depth dimension in the output. The output volume

will have entries that we consider the output of a neuron that looked at only a small window of the input volume.



*Figure 4-11. 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 diagram XX:



*Figure 4-12. Input Volume with Highlighted Focus Input*

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  $32 \times 32 \times 3$  and our receptive field hyperparameter is set as  $5 \times 5$ . Each neuron in the convolutional layer will have weights to a  $[5 \times 5 \times 3]$  region in the input volume. This gives us  $5 * 5 * 3 = 75$  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.

### Learned Filters and Renders

Below we see an example of the learned 96 filters of size [11x11x3] (Krizhevsky et al). 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.

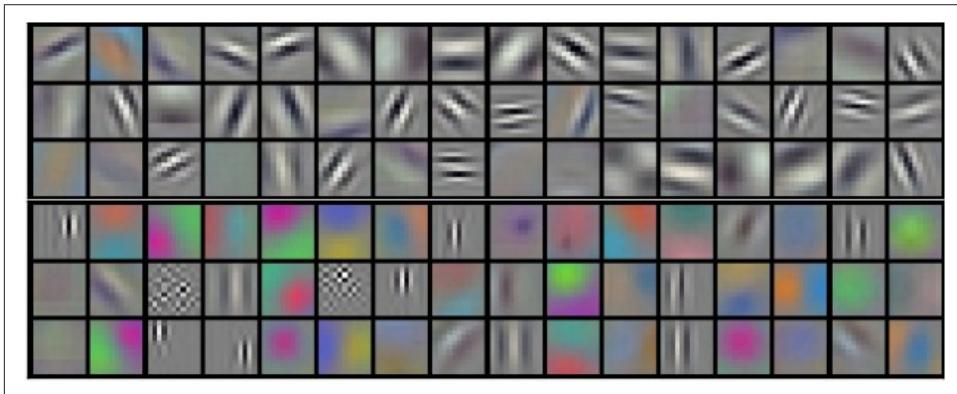


Figure 4-13. Example filters learned by Krizhevsky et al<sup>2</sup> (96 filters, [11x11x3])

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.

### 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.

---

<sup>2</sup> <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>



## 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

Three hyperparameters dictate the spatial arrangement and size of the output volume from a convolutional layer:

- 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.



## 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.

## 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<sup>3</sup> is an example of this where it has 2 fully-connected layers followed by a softmax layer at the end.

## Other Popular Convolutional Network Architectures

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

- LeNet
  - One of the earliest successful architectures of Convolutional Networks.
  - Developed by Yan Lecun (detailed below in the breakout)
  - Originally used to read digits in images
- AlexNet
  - Helped popularize Convolutional Networks in Computer Vision
  - Developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton
  - Won the ImageNet Large-Scale Visual Recognition Challenge 2012 (ILSVRC 2012)
- ZF Net
  - Won the ImageNet Large-Scale Visual Recognition Challenge 2013 (ILSVRC 2013)
  - Developed by Matthew Zeiler and Rob Fergus

---

<sup>3</sup> [http://vision.stanford.edu/teaching/cs231b\\_spring1415/slides/alexnet\\_tugce\\_kyunghee.pdf](http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf)

- GoogLeNet
  - 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
  - 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

## 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 (from Chapter 1) 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 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 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 Turing’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
- Speech synthesis
- Music generation
- Playing video games

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
- 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:

- 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

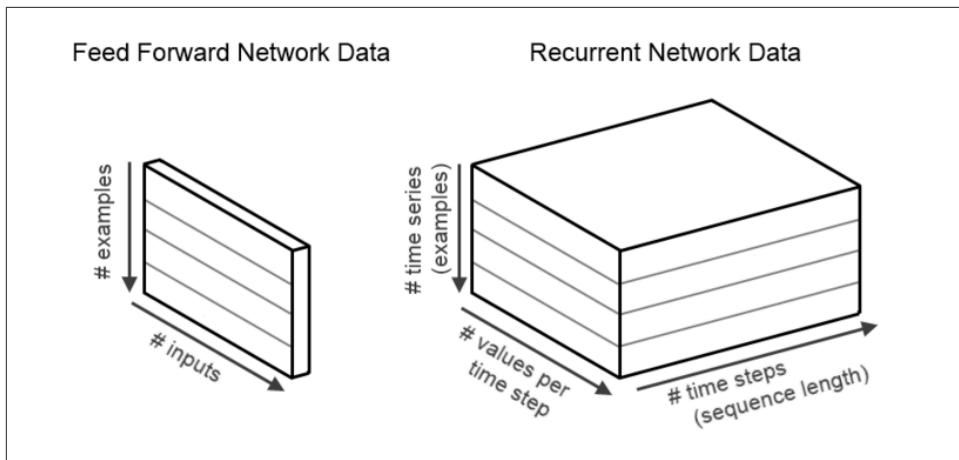


Figure 4-14. Normal Input Vectors Compared to Recurrent Neural Networks Input

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-XX:

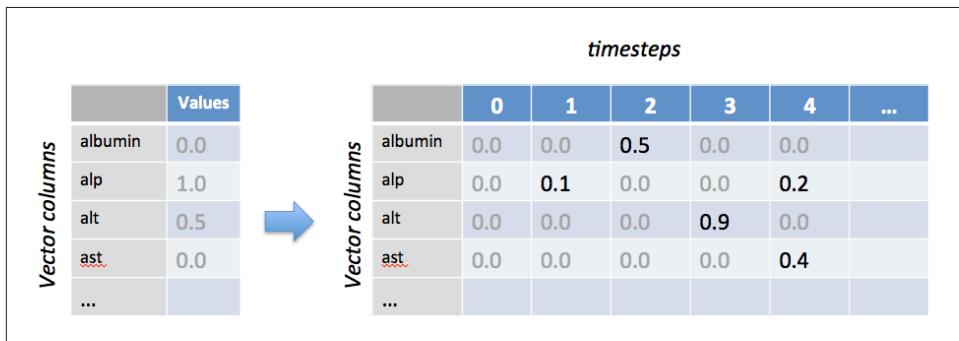


Figure 4-15. The Timestep Aspect of Recurrent Neural Network Input

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 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 the figure below:

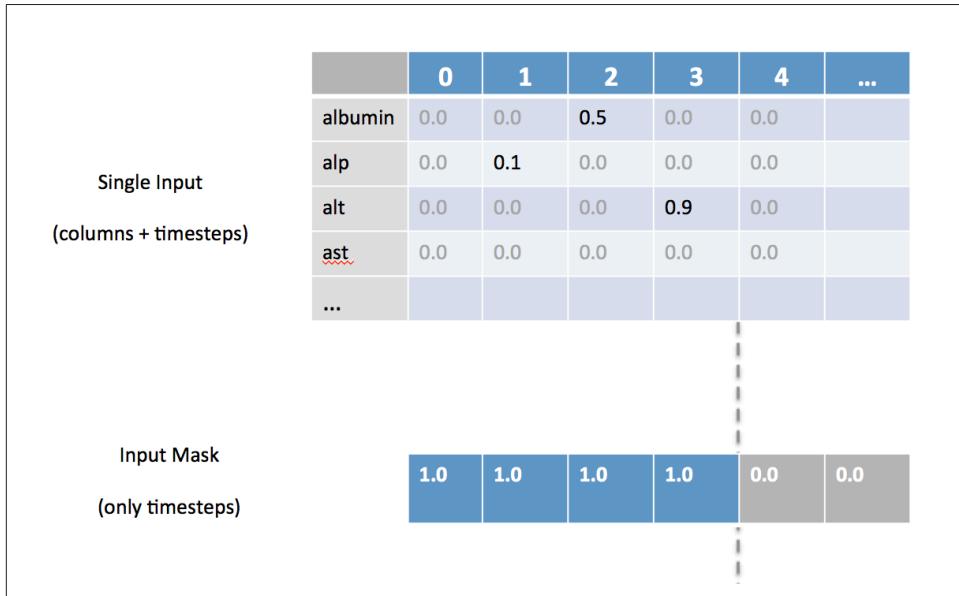


Figure 4-16. Masking Specific Timesteps

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 as an option. 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 makes recurrent networks exceptional at capturing a lot of time-dimension relevant information across many input vectors.

## 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  $2^{64 \times N}$  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.

## 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) is the most commonly used variation of Recurrent Neural Networks. LSTM Networks were introduced in 1995 by Hochreiter and Schmidhuber.

The critical component of the LSTM is the memory cell and the gates (including the forget gate<sup>4</sup>, but also the input gate). The contents of the memory cell are modulated by the input gates and forget gates<sup>5</sup>. 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. Normal recurrent neural networks have this problem, where samples from “long ago” are forgotten and their effect diminishes. 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

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

---

<sup>4</sup> <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.5709&rep=rep1&type=pdf>

<sup>5</sup> <http://www.cs.toronto.edu/~graves/phd.pdf>

- 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 components of the LSTM.

### 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 diagram-XX.

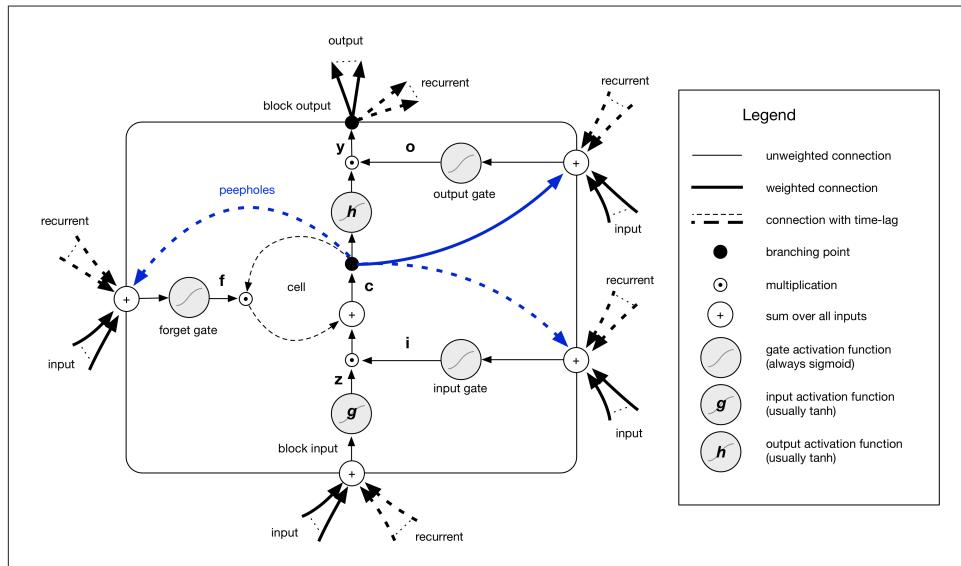


Figure 4-17. LSTM Block Diagram

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 irrelevant input events. 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.

Using the notation from Greff et al<sup>6</sup> 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 4-18. Vector Formulas for LSTM Layer Forward Pass

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

<sup>6</sup> <http://arxiv.org/pdf/1503.04069v1.pdf>

*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

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>

## 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 network. 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 Propagation

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 diagram-X:

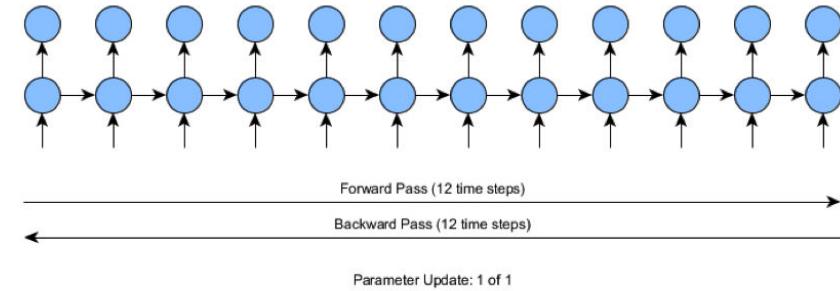


Figure 4-19. Standard Back Propagation Through Time

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.

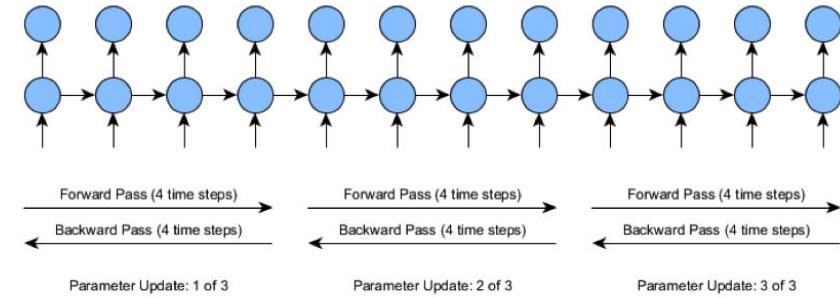


Figure 4-20. Truncated Back Propagation Through Time

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

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 classification
- Image captioning
- Video captioning
- Visual question answering

Another emerging area of research in computer vision with RNNs is “Recurrent Models of Visual Attention”. These networks blend together Convolutional Neural Networks for raw perception and Recurrent Neural Networks for the time domain modeling.

## 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 hierachial 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.

## 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 introduced the fundamental concepts of deep networks. We covered the basic structures in deep networks such as loss functions, activation functions, and optimization algorithms. We also looked at the building block networks, Restricted Boltzmann Machines and AutoEncoders, that we use as layers in larger Deep Networks. We then dove into the four major architectures of deep networks.

All of these have the same core structure of a series of layers (each subsequent one learning from the previous), following an output layer at the end tied to some objective. In the following chapters, we will illustrate real world code examples 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.



# **Building Deep Networks**

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 think of matching 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 engineering features in the input data. In this chapter we'll see examples of Deep Networks matched to specific types of problems. Below we review the data type network architecture pairings briefly.

### **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.

## Timeseries Sequences and Recurrent Neural Networks

Timeseries data represent a waveform when graphed on a two-dimensional plot. 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 challenge.

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.

## 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.

## 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 as Spark and HDFS. The DL4J project also has robust vectorization capabilities with the introduction of Canova 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 pre-processing step. The Deeplearning4j suite includes a library called Canova that vectorizes various common file formats such as CSVs into formats that DL4J networks can parse, notable *svmlight* and *ARFF*.

## 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; with time, they will 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.

[ add info from slide on nd4j ]

## Starting a New DL4J Project

This is a multistep install. We recommend you join our Gitter Live Chat if you have questions or feedback. If you're feeling anti-social or brashly independent, you're still encouraged to lurk and learn.

Here are the system configuration requirements:

1. Java 7 and above
2. Integrated Development Environment: IntelliJ (or another IDE)
3. Maven 3.2.5 or above (Dependency management and automated build tool)
4. Git

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 Canova, our vectorization library, will be automatically installed by following the quickstart instructions below. 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-4.0.0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>

<groupId>MyGroupId</groupId>
<artifactId>MyArtifactId</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <nd4j.version>0.4-rc3.8</nd4j.version>
    <dl4j.version>0.4-rc3.8</dl4j.version>
    <anova.version>0.0.14</anova.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>

    <dependency>
        <artifactId>anova-api</artifactId>
        <groupId>org.nd4j</groupId>
        <version>${anova.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
- **anova-api:** Anova is our vectorization and ETL library

## 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".

## Basic Concepts of 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. 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.

## 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.

### Using the *.fit()* Method

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

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

### Setting Up Training Epochs

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 see this pattern for training in the of the examples below.

## 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 called "saturn":

<https://github.com/deeplearning4j/dl4j-0.4-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/classification/MLPClassifierSaturn.java>

This is a dataset generated by Dr. Jason Baldridge to test neural network frameworks for basic functionality. Below we see the code from the example:

### *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;  
        double learningRate = 0.005;  
        //Number of epochs (full passes of the data)  
        int nEpochs = 30;  
  
        int numInputs = 2;  
        int numOutputs = 2;  
        int numHiddenNodes = 20;  
  
        //Load the training data:  
        RecordReader rr = new CSVRecordReader();  
        rr.initialize(new FileSplit(new File("dl4j-examples/src/main/resources/classification/saturn.csv")));  
        DataSetIterator trainIter = new RecordReaderDataSetIterator(rr, batchSize, 0, 2);  
  
        //Load the test/evaluation data:  
        RecordReader rrTest = new CSVRecordReader();
```

```

rrTest.initialize(new FileSplit(new File("dl4j-examples/src/main/resources/classification/saturn")));

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("relu")
        .build())
    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .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 update

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 b
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("dl4j-examples/src/main/resources/classification/saturn.csv")));
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
rrTest.initialize(new FileSplit(new File("dl4j-examples/src/main/resources/classification/saturn.csv")));
rrTest.reset();
int nTestPoints = 100;
testIter = new RecordReaderDataSetIterator(rrTest,nTestPoints,0,2);
ds = testIter.next();
INDArray testPredicted = model.output(ds.getFeatures());
PlotUtil.plotTestData(ds.getFeatures(), ds.getLabels(), testPredicted, allXYPoints, predictionsAtXYPoints);

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:

```

RecordReader rr = new CSVRecordReader();
rr.initialize(new FileSplit(new File("dl4j-examples/src/main/resources/classification/saturn_data"));
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:

```

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("relu")
        .build())
    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .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 up

```

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.

## Input Layer

The input takes on the raw values 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("relu")
      .build())
```

Here our input layer needs to have the same number of input neurons as the number of independent variable columns in our input vectors:

```
.nIn(numInputs)
```

The number of outputs for the input 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("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("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).

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.

## 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 approximating 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 labels = t.getLabels();
INDArray predicted = model.output(features, false);
eval.eval(labels, predicted);
}

System.out.println(eval.stats());

```

As the reader will recall from chapter 1 we introduce the concept of the F1 score and other 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.

### High-Level Workflow for Modeling MNIST with LeNet

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 actual dataset is the MNIST hand-written image dataset.

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-

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 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 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;  
        int outputNum = 10;  
        int batchSize = 64;  
        int nEpochs = 10;  
        int iterations = 1;  
        int seed = 123;  
  
        log.info("Load data....");  
        DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);  
        DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);  
  
        log.info("Build model....");  
        MultiLayerConfiguration.Builder builder = new NeuralNetConfiguration.Builder()  
            .seed(seed)  
            .iterations(iterations)  
            .regularization(true).l2(0.0005)  
            .learningRate(0.01)//.biasLearningRate(0.02)  
            //.learningRateDecayPolicy(LearningRatePolicy.Inverse).lrPolicyDecayRate(0.001).lrPolicyDecayStart(1000)  
            .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  
                .nIn(nChannels)
```

```

        .stride(1, 1)
        .nOut(20)
        .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 needed be specified in later layers
        .stride(1, 1)
        .nOut(50)
        .activation("identity")
        .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
    .layer(4, new DenseLayer.Builder().activation("relu")
        .nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .activation("softmax")
        .build())
    .backprop(true).pretrain(false);
// The builder needs the dimensions of the image along with the number of channels. these are
new ConvolutionLayerSetup(builder,28,28,1);

MultiLayerConfiguration conf = builder.build();
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.

```
MultiLayerConfiguration.Builder builder = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations)
    .regularization(true).l2(0.0005)
    .learningRate(0.01)//.biasLearningRate(0.02)
    //learningRateDecayPolicy(LearningRatePolicy.Inverse).lrPolicyDecayRate(0.001).lrPolicyP
    .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
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .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 needed be specified in later layers
        .stride(1, 1)
```

```

        .nOut(50)
        .activation("identity")
        .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
    .layer(4, new DenseLayer.Builder().activation("relu")
        .nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .activation("softmax")
        .build())
        .backprop(true).pretrain(false);
// The builder needs the dimensions of the image along with the number of channels. these are 28x28
new ConvolutionLayerSetup(builder,28,28,1);

MultiLayerConfiguration conf = builder.build();
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();

```

## General Hyperparameters

We can see some of the main hyperparameters below.

```

.regularization(true).l2(0.0005)
.learningRate(0.01)
.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.

**Learning Rate.** Our learning rate is set to 0.01.

**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. 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

[ *todo: why?* ]

**Momentum.** Momentum is set as 0.9

## 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
    .nIn(nChannels)
    .stride(1, 1)
    .nOut(20)
    .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.

[ *todo: is this an effect of LeNet?* ]



### Convolution Layers and Activation Functions

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.

**Sizing the Filter.** This pooling layer has a filter size of (2,2)

**Setting the Pooling Layer Stride.** The filter stride is set as (2,2)

## 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("softmax")
    .build())
```

This output layer has a negative log likelihood loss function 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 initialize the MultiLayerNetwork object to train on our input datasets. The code snippet below shows how we take the configuration from the previous section and setup the network.

```
MultiLayerConfiguration conf = builder.build();
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”:

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

#### Source Text for Training LSTMs

This example is set up to train on the Complete Works of William Shakespeare, downloaded from Project Gutenberg. 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 shakespeare 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 nereepad werer;
Motadid. Mert hatterhirl. Iit nesdoesd'nlowhednhanieivetranns deugheuind
Bred yetide rathane fojlond thivh uweet.
Thy lametom theuegfast lart souclalitoloe ilntangylrt or

----- Sample 1 -----
l,, ne agly
Lot Bolncanbom bavantenfircasle womlidibl.
NTERIOO. IrdmisfuUoItolleeeddortiss hot buye.
The hetenle of ile,
'merlliydingiponI, bomgule? Shurtstarer of ate,
Onbibly ot ire pomxatgillant, dakl.
Oxt Mtanlonfyre 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.
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
```

Exit COURSTIO

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 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.  
        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 network.  
        int nSamplesToGenerate = 4;                            //Number of samples to generate after each training epoch.  
        int nCharactersToSample = 300;                          //Length of each sample to generate.  
        String generationInitialization = null;              //Optional character initialization; a random character  
        // Above is Used to 'prime' the LSTM with a character sequence to continue/complete.  
        // Initialization characters must all be in CharacterIterator.getMinimalCharacterSet() by default.  
        Random rng = new Random(12345);  
  
        //Get a DataSetIterator that handles vectorization of text into something we can use to train  
        // our GravesLSTM network.  
        CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);  
        int nOut = iter.totalOutcomes();  
  
        //Set up network configuration:  
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()  
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT).iterations(1)  
            .learningRate(0.1)  
            .rmsDecay(0.95)  
            .seed(12345)  
            .regularization(true)  
            .l2(0.001)  
            .weightInit(WeightInit.XAVIER)  
            .updater(Updater.RMSPROP)  
            .list()  
            .layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns()).nOut(lstmLayerSize)  
                  .activation("tanh").build())  
            .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)  
                  .activation("tanh").build())  
            .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT).activation("softmax")  
                  .nIn(lstmLayerSize).nOut(nOut).build())  
            .backpropType(BackpropType.TruncatedBPTT).tbPTTForwardLength(tbpttLength).tbPTTBackwardLength(tbpttLength)  
            .pretrain(false).backprop(true)  
            .build();  
  
        MultiLayerNetwork net = new MultiLayerNetwork(conf);  
        net.init();  
        net.setListeners(new ScoreIterationListener(1));  
    }  
}
```

```

//Print the number of parameters in the network (and for each layer)
Layer[] layers = net.getLayers();
int totalNumParams = 0;
for( int i=0; i<layers.length; i++ ){
    int nParams = layers[i].numParams();
    System.out.println("Number of parameters in layer " + i + ":" + nParams);
    totalNumParams += nParams;
}
System.out.println("Total number of network parameters: " + totalNumParams);

//Do training, and then generate and print samples from network
int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
        net.fit(ds);
        if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
            System.out.println("-----");
            System.out.println("Completed " + miniBatchNumber + " minibatches of size " + miniBatchSize);
            System.out.println("Sampling characters from network given initialization \\" + (generationInitialization));
            String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,miniBatchSize);
            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 re-use
 * 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 IOException {
    //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 pro

    char[] validCharacters = CharacterIterator.getMinimalCharacterSet(); //Which characters are
    return new CharacterIterator(fileLocation, Charset.forName("UTF-8"),
        miniBatchSize, sequenceLength, validCharacters, new Random(12345));
}

/** Generate a sample from the network, given an (optional, possibly null) initialization. Initialization
 * can be used to 'prime' the RNN with a sequence you want to extend/continue.<br>
 * Note that the initialization is used for all samples
 * @param initialization String, may be null. If null, select a random character as initialization
 * @param charactersToSample Number of characters to sample from network (excluding initialization)
 * @param net MultiLayerNetwork with one or more GravesLSTM/RNN layers and a softmax output layer
 * @param iter CharacterIterator. Used for going from indexes back to characters
 */
private static String[] sampleCharactersFromNetwork(String initialization, MultiLayerNetwork net,
                                                 CharacterIterator iter, Random rng, int charactersToSample)
{
    //Set up initialization. If no initialization: use a random character
    if( initialization == null ){
        initialization = String.valueOf(iter.getRandomCharacter());
    }

    //Create input for initialization
    INDArray initializationInput = Nd4j.zeros(numSamples, iter.inputColumns(), initialization.length());
    char[] init = initialization.toCharArray();
    for( int i=0; i<init.length; i++ ){
        int idx = iter.convertCharacterToIndex(init[i]);
        for( int j=0; j<numSamples; j++ ){
            initializationInput.putScalar(new int[]{j,idx,i}, 1.0f);
        }
    }

    StringBuilder[] sb = new StringBuilder[numSamples];
    for( int i=0; i<numSamples; i++ ) sb[i] = new StringBuilder(initialization);

    //Sample from network (and feed samples back into input) one character at a time (for all samples)
    //Sampling is done in parallel here
    net.rnnClearPreviousState();
    INDArray output = net.rnnTimeStep(initializationInput);
    output = output.tensorAlongDimension(output.size(2)-1,1,0); //Gets the last time step output

    for( int i=0; i<charactersToSample; i++ ){
        //Set up next input (single time step) by sampling from previous output
        INDArray nextInput = Nd4j.zeros(numSamples,iter.inputColumns());
        //Output is a probability distribution. Sample from this for each example we want to generate
        for( int s=0; s<numSamples; s++ ){
            double[] outputProbDistribution = new double[iter.totalOutcomes()];
            for( int j=0; j<outputProbDistribution.length; j++ ) outputProbDistribution[j] = output[s][j];
            int sampledCharacterIdx = sampleFromDistribution(outputProbDistribution,rng);
            nextInput[s] = iter.getCharacter(sampledCharacterIdx);
        }
    }
}

```

```

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

    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.

```
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("tanh").build())
.layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
    .activation("tanh").build())
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT).activation("softmax") //MCXENT +
    .nIn(lstmLayerSize).nOut(nOut).build())
.backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength).tBPTTBackwardLength(
.pretrain(false).backprop(true)
.build();

```

**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.

[ *why is this different?* ]

**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.

## Training the LSTM Network

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

```

int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
        net.fit(ds);
        if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
            System.out.println("-----");
            System.out.println("Completed " + miniBatchNumber + " minibatches of size " + miniBatchSize);
            System.out.println("Sampling characters from network given initialization \" " + (generationInitialization));
            String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,rng,randomSeed);
            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  
}
```

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,nCharactersToGenerate);  
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<sup>1</sup>. This code example will build a model to classify univariate time series as belonging to one of six categories:

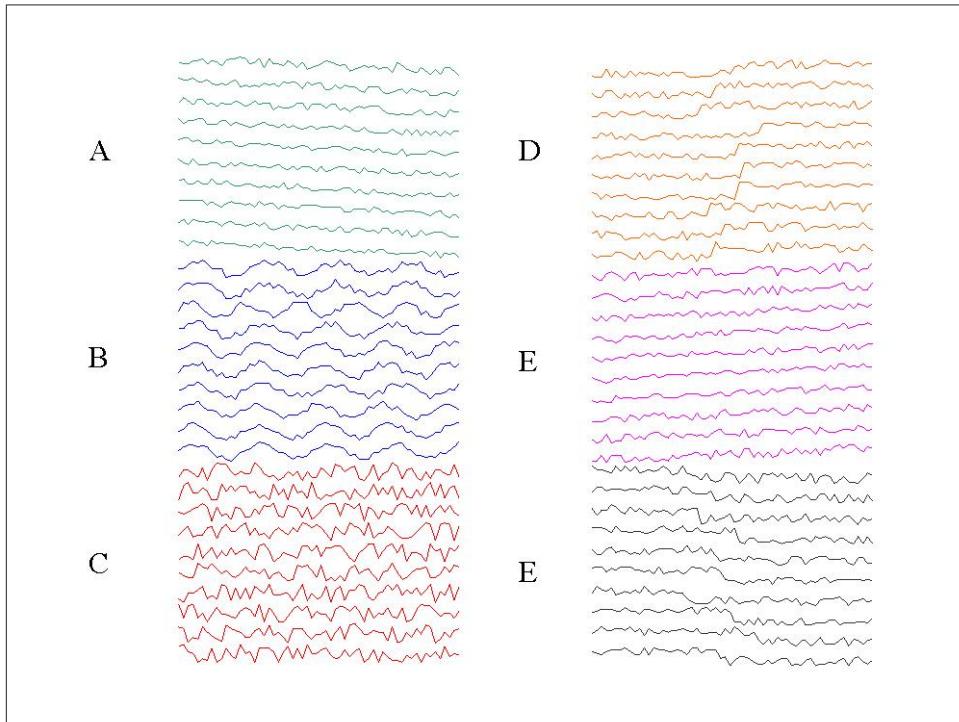
1. Normal (C)

---

<sup>1</sup> <https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series>

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

In the diagram below we can see renders<sup>2</sup> 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.

### **Java Code Listing for Recurrent Classification Example**

In the code example 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”.

---

<sup>2</sup> Image from: <https://archive.ics.uci.edu/ml/datasets/Synthetic+Control+Chart+Time+Series>

#### *Example 5-4. UCI Sequence Classification Example*

```
public class UCISequenceClassificationExample {  
    private static final Logger log = LoggerFactory.getLogger(UCISequenceClassificationExample.class);  
  
    // 'baseDir': Base directory for the data. Change this if you want to save the data somewhere else  
    private static File baseDir = new File("src/main/resources/uci/");  
    private static File baseTrainDir = new File(baseDir, "train");  
    private static File featuresDirTrain = new File(baseTrainDir, "features");  
    private static File labelsDirTrain = new File(baseTrainDir, "labels");  
    private static File baseTestDir = new File(baseDir, "test");  
    private static File featuresDirTest = new File(baseTestDir, "features");  
    private static File labelsDirTest = new File(baseTestDir, "labels");  
  
    public static void main(String[] args) throws Exception {  
        downloadUCIData();  
  
        // ----- Load the training data -----  
        // Note that we have 450 training files for features: train/features/0.csv through train/features/449.csv  
        SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();  
        trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain.getAbsolutePath() + "%d.csv");  
        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 'trainData'  
        trainData.setPreProcessor(normalizer);  
  
        // ----- Load the test data -----  
        // Same process as for the training data.  
        SequenceRecordReader testFeatures = new CSVSequenceRecordReader();  
        testFeatures.initialize(new NumberedFileInputSplit(featuresDirTest.getAbsolutePath() + "%d.csv");  
        SequenceRecordReader testLabels = new CSVSequenceRecordReader();  
        testLabels.initialize(new NumberedFileInputSplit(labelsDirTest.getAbsolutePath() + "%d.csv");  
  
        DataSetIterator testData = new SequenceRecordReaderDataSetIterator(testFeatures, testLabels,  
            false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);  
  
        testData.setPreProcessor(normalizer); // Note that we are using the exact same normalization  
  
        // ----- 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 needed
    .gradientNormalizationThreshold(0.5)
    .list()
    .layer(0, new GravesLSTM.Builder().activation("tanh").nIn(1).nOut(10).build())
    .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
        .activation("softmax").nIn(10).nOut(numLabelClasses).build())
    .pretrain(false).backprop(true).build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();

net.setListeners(new ScoreIterationListener(20)); //Print the score (loss function value) every 20 iterations

// ----- Train the network, evaluating the test set performance at each epoch -----
int nEpochs = 40;
String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);

    //Evaluate on the test set:
    Evaluation evaluation = net.evaluate(testData);
    log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

    testData.reset();
    trainData.reset();
}

log.info("----- Example Complete -----");
}

}

```

## Setting Up Input Data and Vectorization

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, 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.csv
SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain.getAbsolutePath() + "0",
                                                 featuresDirTrain.getAbsolutePath() + "449"));

```

```

SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain.getAbsolutePath() + "%d");

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 'trainData' will be normalized
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.

## Network Architecture

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
    .gradientNormalizationThreshold(0.5)
    .list()
    .layer(0, new GravesLSTM.Builder().activation("tanh").nIn(1).nOut(10).build())
    .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
        .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.

## Training the LSTM Network

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:

```

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 pretraining.

### Reconstruction Error as Anomaly Indicator

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

### High-Level Workflow for Training the AutoEncoder

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 Example

Below we have the java code for the Anomaly AutoEncoder example.

*Example 5-5. AutoEncoder Example Code to Find Anomalies*

```

public class MNISTAnomalyExample {

    public static void main(String[] args) throws Exception {
        //Set up network. 784 in/out (as MNIST images are 28x28).
        //784 -> 250 -> 10 -> 250 -> 784
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()

```

```

    .seed(12345)
    .iterations(1)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.ADAGRAD)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(0.05)
    .regularization(true).l2(0.0001)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
        .build())
    .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)
        .build())
    .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
        .build())
    .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
        .lossFunction(LossFunctions.LossFunction.MSE)
        .build())
    .pretrain(false).backprop(true)
    .build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.setListeners(Collections.singletonList((IterationListener) new ScoreIterationListener(1)));

//Load data and split into training and testing sets. 40000 train, 10000 test
DataSetIterator iter = new MnistDataSetIterator(100,50000,false);

List<INDArray> featuresTrain = new ArrayList<>();
List<INDArray> featuresTest = new ArrayList<>();
List<INDArray> labelsTest = new ArrayList<>();

Random r = new Random(12345);
while(iter.hasNext()){
    DataSet ds = iter.next();
    SplitTestAndTrain split = ds.splitTestAndTrain(80, r); //80/20 split (from miniBatch = 100)
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argmax(dsTest.getLabels(),1); //Convert from one-hot representation
    labelsTest.add(indexes);
}

//Train model:
int nEpochs = 30;
for( int epoch=0; epoch<nEpochs; epoch++ ){
    for(INDArray data : featuresTrain){
        net.fit(data,data);
    }
    System.out.println("Epoch " + epoch + " complete");
}

//Evaluate the model on test data

```

```

//Score each digit/example in test set separately
//Then add triple (score, digit, and INDArray data) to lists and sort by score
//This allows us to get best N and worst N digits for each type
Map<Integer, List<Triple<Double, Integer, INDArray>> listsByDigit = new HashMap<>();
for( int i=0; i<10; i++ ) listsByDigit.put(i,new ArrayList<Triple<Double, Integer, INDArray>>());

int count = 0;
for( int i=0; i<featuresTest.size(); i++ ){
    INDArray testData = featuresTest.get(i);
    INDArray labels = labelsTest.get(i);
    int nRows = testData.rows();
    for( int j=0; j<nRows; j++){
        INDArray example = testData.getRow(j);
        int label = (int)labels.getDouble(j);
        double score = net.score(new DataSet(example,example));
        listsByDigit.get(label).add(new ImmutableTriple<>(score, count++, example));
    }
}

//Sort data by score, separately for each digit
Comparator<Triple<Double, Integer, INDArray>> c = new Comparator<Triple<Double, Integer, INDArray>>(){
    @Override
    public int compare(Triple<Double, Integer, INDArray> o1, Triple<Double, Integer, INDArray> o2){
        return Double.compare(o1.getLeft(),o2.getLeft());
    }
};

for(List<Triple<Double, Integer, INDArray>> list : listsByDigit.values()){
    Collections.sort(list, c);
}

//Select the 5 best and 5 worst numbers (by reconstruction error) for each digit
List<INDArray> best = new ArrayList<>(50);
List<INDArray> worst = new ArrayList<>(50);
for( int i=0; i<10; i++){
    List<Triple<Double, Integer, INDArray>> list = listsByDigit.get(i);
    for( int j=0; j<5; j++ ){
        best.add(list.get(j).getRight());
        worst.add(list.get(list.size()-j-1).getRight());
    }
}

//Visualize the best and worst digits
MNISTVisualizer bestVisualizer = new MNISTVisualizer(2.0,best,"Best (Low Rec. Error)");
bestVisualizer.visualize();

MNISTVisualizer worstVisualizer = new MNISTVisualizer(2.0,worst,"Worst (High Rec. Error)");
worstVisualizer.visualize();
}

private static class MNISTVisualizer {
    private double imageScale;

```

```

private List<INDArray> digits; //Digits (as row vectors), one per INDArray
private String title;

private MNISTVisualizer(double imageSize, List<INDArray> digits, String title ){
    this.imageSize = imageSize;
    this.digits = digits;
    this.title = title;
}

public void visualize(){
    JFrame frame = new JFrame();
    frame.setTitle(title);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(0,5));

    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<768; 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)(imageSize*28),(int)(imageSize*28),Image.SCALE_SMOOTH);
        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.

```

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
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argMax(dsTest.getLabels(),1); //Convert from one-hot represent
    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

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("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.

## Training the AutoEncoder Network

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.

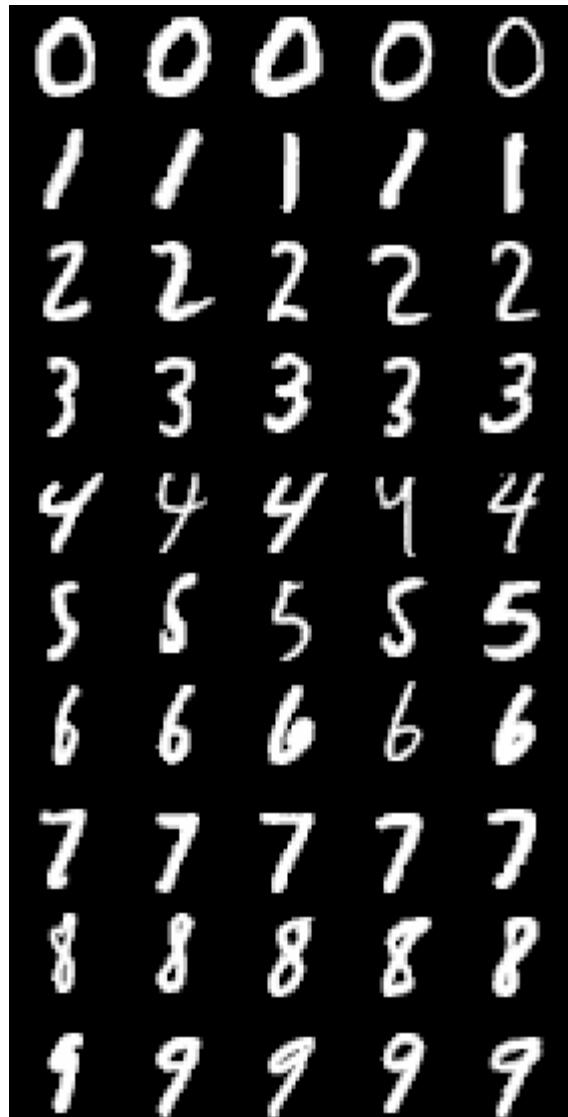


Figure 5-2. Digits Best Learned by the AutoEncoder

The images that had the highest reconstruction error are composed in another render as seen below.

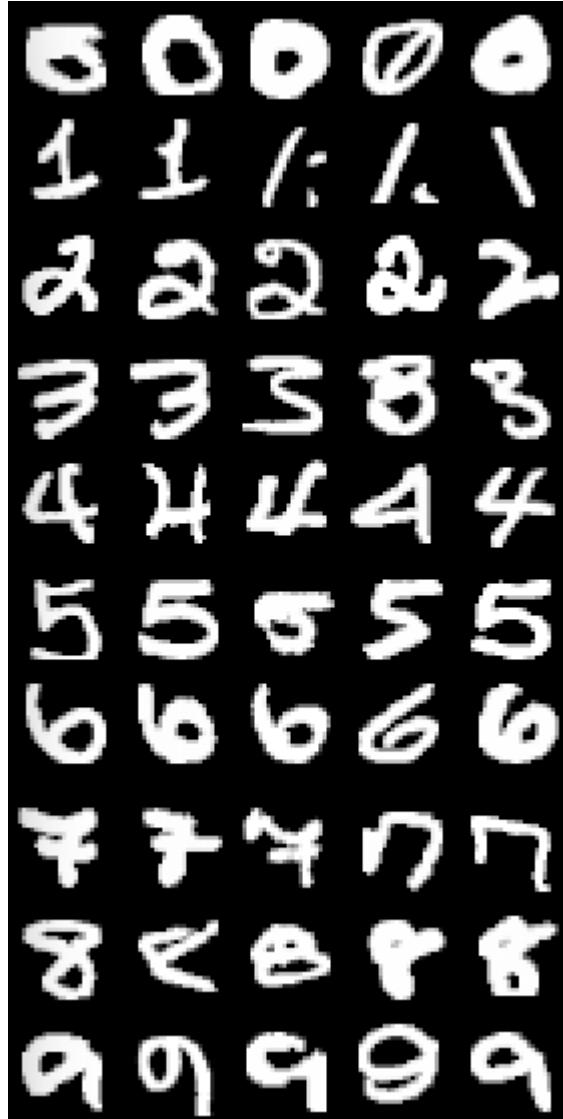


Figure 5-3. Hard to Learn Hand-Written Digits

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.

# 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, Character generation, and learning word embeddings are common applications of Deep Learning.

## 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.

### 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 Net Language Model (NNLM) to build these contextual word vectors. In practice this is a two layer neural network trained with gradient descent.

### Word2Vec Vectors

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 sig-

nificantly better than LSA for preserving linear regularities among words and be less computationally expensive than LDA.

## Modeling Context

Word vectors retain context of the source text around the word in the form of multi-word windows. For the purpose of machine learning we define a word's meaning as the words that surround it most consistently. With enough data we can 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 or distance we described previously in the chapter or Cosine distance. This gives us words that have short cosine similarity distances or, more generally, "close representations". Below in table [table-X] we see some examples of words most similar to "france" based on a trained model:

[ TODO: \*\*\*Insert Cosine Distance table\*\*\* ]

A perfect match represented by the value zero would be between two of the same words (identity, or "Amsterdam" equals "Amsterdam"). The higher the value above zero we see for the similarity metric the greater the dissimilarity between the meanings of the words. We need to train our word2vec model on large datasets with sufficient vector dimensionality to see strong regularities in the word vector space.

We can 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 X measured by our distance metric (cosine distance here) in a 1NN search fashion to find our word vector. When our word vectors are well trained we should find the correct word (“smallest”) using this technique. We can also find subtle relationships such as a capital cities of states (France is to Paris as Germany is to Berlin). These semantic relationships can be effective in improving existing NLP techniques such as machine translation, information retrieval, and question answering systems. So far we’ve looked at the semantic properties of word vectors but we haven’t looked at how they are used in the more traditional vectorization sense because we wanted to show how this technique has useful applications beyond generating vectors for input instances.

### Practical Uses of Word2Vec

The output of the Word2vec neural net 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.

To do canonical document classification after we’ve generate 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.

Compared to other text vectorization techniques word2vec retains a tremendous amount of context about the meaning of words. The bag of words model, as we wrote about previously, considers only the number of times a word occurs and retains nothing about the context around a word. TF-IDF when combined with n-grams can give us some degree of context, but it does not provide the semantic mechanics of words in the ways that word2vec does. As we’ve seen word2vec can be powerful when used in semantic applications and straight forward document vectorization.

### Java Code Listing for Word2Vec Example

The code example 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 5-6. Word2Vec Example*

```
public class Word2VecRawTextExample {  
    private static Logger log = LoggerFactory.getLogger(Word2VecRawTextExample.class);  
    public static void main(String[] args) throws Exception {
```

```

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();
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
WordVectorSerializer.writeWordVectors(vec, "pathToWriteto.txt");

log.info("Closest Words:");
Collection<String> lst = vec.wordsNearest("day", 10);
System.out.println(lst);
UiServer server = UiServer.getInstance();
System.out.println("Started on port " + server.getPort());
}
}

```

Below we'll discuss specific parts of this code to better understand what is happening in the example.

## Setting Up Input Data for Word2Vec

Section content goes here

## Configuring Word2Vec Model Architecture

Section content goes here

## **Training the Word2Vec Model**

Section content goes here

## **Other Semantic Embedding Variants**

Section content goes here

<https://github.com/deeplearning4j/dl4j-0.4-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/sequencevectors/SequenceVectorsTextExample.java>



## 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<sup>1</sup>

*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
- Configuring and tuning specific types of Deep Networks
- 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. We then focus specifically on tuning techniques for the most well known architectures in Deep Networks. We start out the first half of the chapter looking at deep network hyperparameters in a general sense, and then progressively get more specific to the major architectures of deep networks:

- Deep Belief Networks

---

<sup>1</sup> <https://en.wikipedia.org/wiki/Paracelsus>

- Convolutional Neural Networks
- 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 arbitrary real-valued numbers) 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

Number of layers and neuron 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, layers, and how we'll initialize the weights, 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 functions.

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. 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. visualization
  - c. 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 training on more data faster
    - a. tweak hyper parameters from final subset parameters for Spark run
    - b. rule of thumb: need a few more epochs and higher learning rate with Spark due to parameter averaging

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 each data type in the sections below, we'll walk through how the data is represented and what architecture(s) of networks work best for the given data type.

### Columnar 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.

## Image Data

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

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.

## Audio Data

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.

## Video Data

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. 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. In this case we use the identity (linear) output function and only have one output on the output layer.

## 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. In this variation we might be classifying a document as to what category (e.g. “sports”, “business”, “politics”) it belongs to.

### Model with a Single Label

In the basic single label example 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. In the multiple

label scenario we'd want to use another type of activation function for our output layer because now we need scores for multiple labels.

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 layer optimization method (XENT).



### Single Output or Two Outputs for Binary Classifier?

There is a variation where we might use a softmax 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 optimization method.

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).

## Model with Multiple Labels

In this scenario we'd use an output layer with a softmax activation function. The number of output units is equal to the number of classes we want to classify. 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 see a convolutional neural network setup in the example below with a softmax activation function on the output layer:

*Example 6-1. Setting up an output layer in a Convolutional Network in DL4J*

```
.layer(2, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
    .nOut(outputNum)
    .weightInit(WeightInit.XAVIER)
    .activation("softmax")
    .build())
```

The specific output layer section is highlighted in the example above where we see the layer type being specified with the .activation() method.



## Softmax vs Sigmoid

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.

## Model with a High Number of Labels

In the case where the number of intended labels reaches the ten-thousands we’d want to use a hierarchical softmax output layer. Hierarchical softmax is a fast(er) approximation to softmax suitable for a large number of classes. We should consider it for computational reasons and not to improve accuracy.

# Working with Layer Count and Number of Neurons

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. Similarly a DenseLayer has more neurons than a similar 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 Neuron 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 [13]. 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 [48]. What you 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. [48]



### 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).

## 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.

## Basic Rule of Thumb

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

Use `WeightInit.XAVIER` weight initialization, or use `WeightInit.RELU` when using “relu” or “leakyrelu” activations.

We can speed learning by choosing a larger initial random values but this may lead to worse quality in the final model generated by the network’s learning process. 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 zero. Initial hidden biases of 0 are usually fine outside of scenarios where you are targeting a specific sparsity probability. Below we discuss the major strategies for weight initialization.

## Weights Connecting to TanH Units

Other initialization methods involve concepts such as “Sparse Initialization” [28] and using tanh units [28].

## Weights Connecting to ReLU Units

ReLU weight initialization should be performed in the manner:

$$N(0,1) * \sqrt{2/n}$$

where n is number of connections for the given layer [48][49]. In some variations we can use small constant values (such as 0.01) for the bias initialization of ReLUs. The intuition is that we want ReLU units to fire at the beginning of training.

## Using Activation Functions

In chapter 1 we introduced the basic activation functions used in neural networks:

- Linear
- Sigmoid (Logistic)
- Tanh
- Hard Tanh
- Softmax
- ReLU



### Reminder About Activation Functions

A deep network is defined as a neural network with several layers, which transfer the output of each layer as input to the next layer via a nonlinear transform called an *activation function*.

Above we discussed how activation functions are related to different goals in output layers. Let's take a look at the use of activation functions in the context of hidden layers. To start off let's look at a high-level table of how we apply activation functions based on the type of output values desired.

## Target Distributions

Below in table X we have notes for matching data distribution types to some of the activation functions in feedforward networks. We'll discuss each class of activation function in the following sections and then close with a table explaining how to match up each to a network architecture.

*Table 6-1. Target Distributions vs Activation Functions in Feedforward Networks*

Target Distribution	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)	Logarithmic Normalization
Continuous-Valued (Not bounded)	Linear Activation Function (equates to no activation function)

Let's now dig into each of the specific activation functions with some guidance on how to use each and where they are most effectively used.

## Linear

This is known as the “identity” activation function (where  $f(x) = x$ ). We typically use this as the output layer for regression problems. We commonly see this activation used in the following architectures:

- Output layer for regression problems

## Sigmoid

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.

### Comparison to Other Activation Functions

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 [15].

### Issues with Sigmoid Activation Functions in Practice

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 [48]



#### 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"

## Tanh

The Tanh non-linearity shapes numbers to be on the range [-1, 1]. The hyperbolic tangent function is a common activation function today yet is less common than the ReLU family of activation functions at this point.

## Rectified Linear Unit (ReLU)

ReLU activation functions (Piecewise linear units) are the most popular type of hidden unit [15]. We see them used commonly in Convolutional Neural Networks as the activation function of convolution layers.

## Advantages Compared to Other Activation Functions

Stochastic Gradient Descent has been found to speed up learning with ReLU's when compared to sigmoid and Tanh functions [48]. 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 [48] which (unlike the ‘vanilla’ ReLU) has non-zero gradient for all input values.

Let’s take a look at an example network setup to demonstrate the ReLU activation function in practice. In the example below we can see a ReLU activation function being setup for a layer in DL4J.

*Example 6-2. Setting up a ReLU activation function in DL4J*

```
.layer(4, new DenseLayer.Builder().activation("relu")
      .nOut(500).build())
```

The activation function is set with the `.activation()` method as in bold above.

## Softmax

We see softmax activation functions used as the activation function of output layers that are classifying multiple labels.

## Softplus

This is a smooth version of the rectifier [ define ]. We tend to use this unit for multi-layer perceptrons and RBMs [15].

## Hard Tanh

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.

## Maxout

A maxout model is a feed-forward network (MLP, CNN) 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.

### Advantages of Maxout Layers

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	<ul style="list-style-type: none"><li>• Output Layer for Binary Classification</li><li>• Outputs values on the range [0,1]</li><li>• Fallen out of favor, not to be used in hidden layers</li></ul>
Tanh	<ul style="list-style-type: none"><li>• Continuous data (more than 0-1)</li><li>• LSTM Layers</li></ul>
Softmax	Output Layer for Classification where scores all sum to 1.0
Rectified Linear Units (ReLU)	<ul style="list-style-type: none"><li>• Restricted Boltzmann Machines</li><li>• Convolutional Neural Network Layers</li><li>• Multi-Layer Perceptron Network Layers</li></ul>

## Summary of Activation Function Usage

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

In chapter 1 and 2 we introduced the fundamentals of loss functions. Here we'll take a look at ways to use loss functions specifically for training in Deep Learning.

[fill in w new loss material ]

## Intuition

Loss functions let the optimization function know how well its doing at the intended task. With DL4J we can have a loss function per layer which allows us to better optimize for the goal at hand depending on what each layer of the network is trying to accomplish (e.g. “hidden layers vs output layer”). Below we list a few notes about each loss function, where to use it most effectively, and any caveats about how it is used.

### 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. 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.

## Loss Functions for Regressions

When working on a regression model that requires a real valued output we use the squared loss (error) function, much like the case of ordinary least squares in linear regression

## Loss Functions for Classification

In this section we'll list the most common loss functions for classification models. These functions were initially reviewed in Chapter 2.

## Hinge 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.



### 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.

## 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 \$ number. Predicting valid probabilities means generating numbers between 0 and 1.

## Summary

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	RBM <sup>s</sup> , AutoEncoders	Used for feature engineering
Squared Loss	Output Layer	Regression
Cross Entropy	Output Layer	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

## Understanding Learning Rates

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. When we progressively drop the learning rate we tend to see the reconstruction error fall significantly as we train. We slowly increase momentum (described in the next section) as we train as the learning rate falls. Increasing learning rate is not always helpful because it creates a smaller noise level in the stochastic weight updates

which gives us slower learning in the long term. We want the learning rate drop to only start to drop near the end of learning.

## Learning Rates and Updating Parameter Vectors

Note that not all parameter vector update methods use learning rate. [ todo: example? ]

## Intuition

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.



### Parallelization and Regularization

In some cases of parallelization we'd use no prior because parameter averaging achieves the intended effect itself.

A great recipe for setting the learning rates for the weights and biases is to examine the histogram of the weight updates and a histogram of the weights. The updates should be about 10-3 times the weights and within an order of a magnitude. We look at creating renders of histograms during training later in this chapter.

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. Per-parameter adaptive learning rates can be aided by techniques like adagrad, RMSProp, and momentum. Implementations such as DL4J have adagrad built in so users can worry less about hand tuning the learning rate.



## 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 monotonically decreasing<sup>2</sup>.

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.

## Summary

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]

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 [48]. We want this decay rate to adapt per parameter and not be too aggressive to hinder learning too quickly.

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.



## 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 [13] 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

---

<sup>2</sup> <http://cs231n.github.io/neural-networks-3/#ada>

We suggest trying learning rate methods in the following order:

1. RMSProp
  - Uses moving average of squared gradients
  - Doesn't have issue where learning rates get monotonically smaller as in Adagrad
2. Momentum (Nesterovs)
3. Adagrad

Other supported options in DL4J include:

- Adam
- Adadelta
- Nesterovs



### Working with Momentum/Nesterovs in DL4J

Momentum helps the learning process escape these local minima and find better solutions to the optimization process [28].

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 ] as described in [48]. A typical setting is to start the momentum value at 0.5 and anneal it to 0.9 or so over multiple epochs [48].

It's also critical to have `.updater(Updater.NESTEROVS)` in addition to `.momentum(0.5)`. Without these two settings momentum will not be enabled.

## 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 [51].

Sparsity helps training get unstuck when weights get large values and have issues backing out of them. 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. We see sparsity used in practice with Restricted Boltzmann Machines and Autoencoders.



### 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  $0.1^9$ .

## Leveraging Histograms When Setting Sparsity

To understand how well the sparsity setting is working we take a look at the histogram of the mean activities of the hidden units. We show how to look at histograms later in Chapter 6. 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 Other Hyperparameters

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.

## Applying Methods of Optimization

In chapter 1 we touched on solving systems of linear equations. In the sub-domain of deep learning our problem may not be linear but we still see some of these concepts to be relevant. 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.

Some researchers have made the argument that adding additional units 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.

To that extend we review more exotic methods of optimization in Limited memory BFGS (L-BFGS) and Conjugate Gradient (CG). These methods leverage second-order optimization information to take “better” steps when training. Combined with linear search these methods can significantly speed up the pre-train training process in Deep Learning.

## Tuning Intuition

We start out with an optimization heuristic hierarchy to frame how we think of which optimization algorithm to use:

1. What is my problem?
2. What is my network architecture?
3. Determine an optimization algorithm that best fits 1 and 2

Let's look at each of these to build our argument for this heuristic.

### What is My Problem?

Generally this is tied to the network architecture as we determined earlier in the chapter. There are some caveats and layer specific scenarios, as we'll cover later in this section.

### What is the Network Architecture?

Earlier in this chapter we discussed how to match up network architecture to input data type. Based on this decision we already have the answer to this question.

Before we create a table to match optimization algorithm to optimization scenario, let's look at some tuning specifics around each of the optimization algorithms.

## Stochastic Gradient Descent

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 [37] 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 [1].

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.

### Stochastic Gradient Descent Best Practices

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 [12]
- 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 [12]

- 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 your data (not specific to SGD, but worth repeating as it's often forgotten)

## L-BFGS

In some problem context's researchers have found second-order optimization methods such as L-BFGS to be faster and more stable than stochastic gradient descent. A drawback of using the basic versions of L-BFGS and Conjugate Gradient is that their batch nature dictates that we computed the gradient on the entire dataset to make an update. With larger datasets this does not scale gracefully.

A solution that has been found is to use vectorized implementations with mini-batch (Described early in this chapter) to compute the gradient on subsets of the data progressively. This allows us to scale out both of these optimization techniques with parallelization and GPUs. L-BFGS has been shown to effectively fine tune pre-trained deep autoencoders, train convolutional networks, and train distributed Deep Belief Networks.



### Second-Order Methods and Batch Sizes

In some cases with second order methods we'll see the batch size be the entire training set [15]. Note that this can be (a) impossible due to memory requirements, or (b) inefficient, with large data sets.

## Conjugate Gradient

Methods such as L-BFGS or Conjugate Gradient in conjunction with line search are generally more stable to train and easier to check for convergence. These methods can also be run in a parallel fashion (Chu et al., 2007) or in a vectorized fashion where we can augment their speed with GPUs (Raina et al., 2009). These added runtime possibilities have shown L-BFGS and Conjugate Gradient to be fast in practice.

## Hessian Free

Hessian free (HF) optimization is another 2nd-order optimization method commonly seen in deep learning literature. Hessian Free does not make any approximation to the Hessian giving it the advantage of not requiring the costly full matrix inversion [30].

Hessian Free iteratively performs line search and calculates the next step direction using the conjugate gradient method. In this context we're only required to compute the curvature-vector products and not the entire curvature matrix (the Hessian). Methods using second-order information operate in batch settings updating the weights less but with more substantial updates. As we've seen previously in this chapter, these batch methods are vectorized by the mini-batch technique as we'll discuss more in a moment. There are also methods for parallelizing the gradient and curvature information across multiple machines. Hessian-free has been shown to train deep autoencoders without the use of pre-training and separately shown to train recurrent networks.

### Stochastic Hessian Free

In a later evolution we see a version of HF called Stochastic Hessian-free (SHF) optimization that achieves competitive performance in terms of training results. A defining characteristic of SHF is that it operates in a mini-batch manner as described previously in this chapter. SHF integrates dropout by randomly omitting feature detectors on both gradient and curvature mini-batches from the last hidden layer during each iteration. [ is this pre-train? ] In this case we're assuming the same feature detectors are omitted between the gradient and curvature mini-batches since curvature mini-batches are a subset of gradient mini-batches. SHF has been shown to perform well for classification in deep networks by adapting the batch sizes and number of CG iterations. However, Adagrad has not been extensively experimented with for hessian free optimization so far.

## Comparing the Methods

In lower dimension problems L-BFGS is the faster optimization method of the group. As the dimensionality of the input vectors grows we see Conjugate Gradient becoming the faster optimization method and outperforms both SGD and Conjugate Gradient. SGD's performance can be improved with large mini-batch sizes and line search. Hessian free automatically derives the step size but is not generalizable to all architectures. With certain modern advances such as rectified linear (ReLU) units the hessian free methods are not as necessary.

With careful initialization and a good momentum strategy we see Stochastic Gradient Descent perform on par with hessian-free methods [37]. Hessian free methods have a reputation for being better than Stochastic Gradient Descent in the domain of deep learning. However (depending on the problem), tuning momentum or using Nesterov's has shown to bring Stochastic Gradient Descent performance up to be on par with Hessian Free [37]. In the table below we summarize this information.

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

## Choosing An Optimization Algorithm

Now that we've provided some usage context let's mold these ideas into a set of rules for picking an optimization algorithm based on the problem context.

### When Working with Smaller Datasets

When dealing with smaller datasets we suggest using the second-order optimization methods and set the batch size to the full dataset.

### When Working with Larger Datasets

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 [15].

### When to Use Stochastic Gradient Descent

Restricted Boltzmann Machines tend to benefit from Stochastic Gradient Descent as an optimization algorithm [31]. Overall Stochastic Gradient Descent is the place most practitioners will start from for optimization purposes.



## Stochastic Gradient Descent as Most Frequent Starting Place

In practice people seem to use Stochastic Gradient Descent + (momentum or adagrad or RMSProp etc) more frequently than second order methods

## When to Use L-BFGS or Conjugate Gradient

Research and experimentation has found L-BFGS to be highly competitive on low-dimension problems as compared to Stochastic Gradient Descent and Conjugate Gradient. We also see L-BFGS used as a good optimization algorithm for Convolutional Neural Networks.

As the problem input dimensionality increase we begin to see Conjugate Gradient outperform L-BFGS and Stochastic Gradient Descent. Neural network architectures that benefit from L-BFGS and Conjugate Gradient [31] include:

- Autoencoders
- Deep Autoencoders
- Sparse Autoencoders

## Something to Consider

“In our experiments, different optimization algorithms appear to be superior on different problems.” [31]

L-BFGS and Conjugate gradient have both been shown in practice to be faster and more stable than Stochastic Gradient Descent for some scenarios. Many times the practitioner has to experiment with different optimization algorithms to find the best fit.

## When to Use Hessian Free

Hessian Free has been seen as an effective way to train Recurrent Neural Networks [28]. Hessian Free is not immune to poor weight initialization strategies and will perform poorly if we don't pay attention to how we initialize the weights. It rarely exceeds 200 epochs of the data in practice [30] compared to Stochastic Gradient Descent which may need 1000's of epochs.

In some contexts the mini-batch size tends to be larger and may be increased as the learning progresses [30]. We tend not to use Hessian Free methods with pre-training (e.g. Deep Belief Networks) where Stochastic Gradient Descent seems to work better.

## Optimization Methods Seen as Impractical

Newton's algorithm requires  $O(N^3)$  operations per iteration to generate the inverted Hessian matrix and is seen as an impractical approach in training deep learning networks in general [1]. Standard BFGS requires  $O(N^2)$  operations per iteration and falls into the same camp as Newton's. BFGS is only practical where we're optimizing a small network and (memory reduced) L-BFGS is generally used instead. Lastly, the Gauss-Newton and Levenberg Marquardt algorithm has the same issues around complexity ( $O(N^3)$ ) and is considered impractical as well for larger networks.

## Summary

Quick reference on training different networks:

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
Recursive Networks	L-BFGS

## 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 quickly hit the limits of sequential learning in modern machine learning activities (Zinkevich, 2011). 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) 6 hours to read the content of a single HD

## From Batch to Online Learning

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. We impractical because to simply the data takes anywhere from many hours to many days. These leads us to spreading the load amongst many machines with the goal of bringing the processing time back to some manageable span. We tackle this task with parallelization of algorithms.

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.

## Amusing Digression

More amusing details go here

### 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.

## The Cost of Moving Data

In the world of big data we need to move the data less (we'll touch on this more below) so we end up doing a lot of data parallelism. We can see recent work where traditionally serial learning processes have been redesigned for parallel computations. This is accomplished by having a core perform the local computations and then combining the work from several cores centrally to produce a global result (Dean and Ghemawat, 2004; Chu et al., 2006). As our dataset exceeds the capacities 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).

## Parallel Iterative Algorithms

As we parallelize iterative-class algorithms such as Stochastic Gradient Descent we can use local thread-parallelism effectively with the parameter-averaging technique (Langford 2007, Dean 200x, McDonald, 2010). 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 it's used far and wide. The authors of this book define the term “big data” in the practical sense that the data is processed where it is stored where we move the computation to the data. MapReduce is architecturally built around this concept. Many companies and tools claim to handle big data, but simply put if you move the data before you process it you aren't working with big data. [ end highlight note ]

### 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.

Hadoop 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 included in Hadoop distributions 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.



### Issues With Iterative Algorithms and MapReduce

If we wanted to make 100 passes over the dataset and our MapReduce scheduling cost is 30 seconds per dataset pass, this gives us an overhead of  $(100 \times 30)$  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 recently (Mann et al., 2009; Zinkevich et al., 2010) that have parallelized SGDs without using the Map-Reduce framework. We also see the evolution of open source frameworks (Vowpal Wabbit, Spark, IterativeReduce) for the parallelization of iterative methods where parameter averaging is a central theme in the implementation.

## Parallelizing Stochastic Gradient Descent

Another publication (reference) 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 diagram-X:

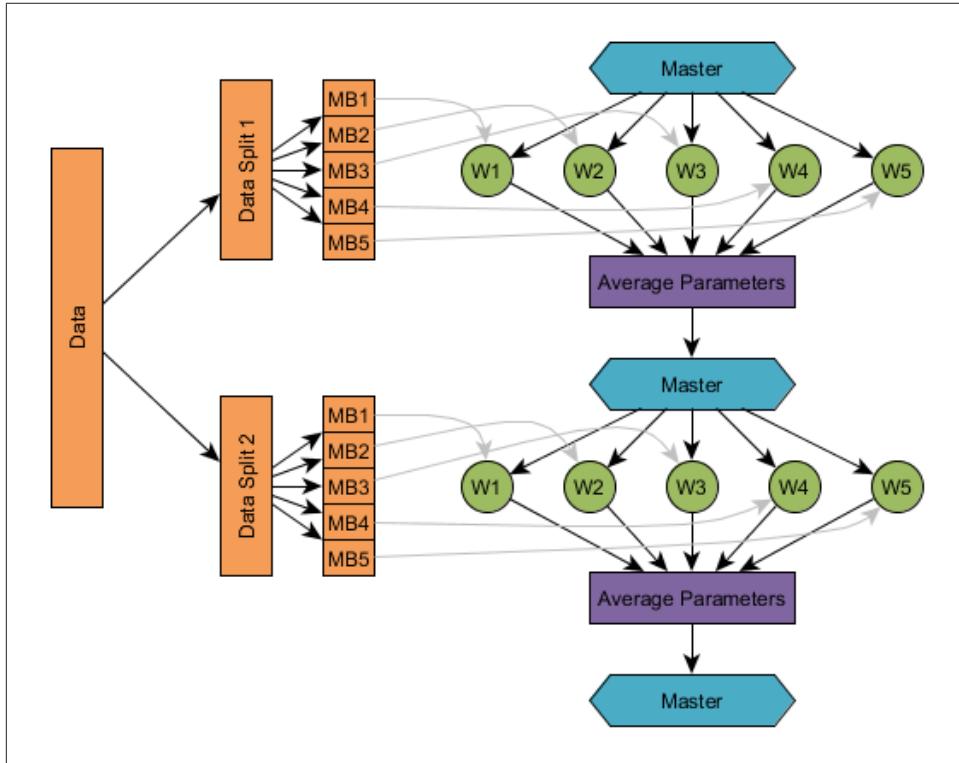


Figure 6-1. DL4J's Parameter Averaging Parallelization Strategy

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 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

While currently DL4J does not use a parameter server the project owners are actively working on a design that is appropriate for the project. A parameter server design will eventually show up in 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

Averaging frequency is configurable. Per minibatch is inefficient. 5-20 is common 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 implementations show that the speed improvements are close to linear in the number of machines.

## Parallelization Effects on Training

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<sup>3</sup>

This has proven a fertile area for evolution in how we train large Deep Belief Networks and other variants of neural networks. An interesting property of GPUs is that they can work concurrently with thousands of threads and scheduling linear algebra work on cores is possible with little overhead. This class of fine-grained parallelism makes GPUs an attractive candidate for speeding up our large-scale linear algebra computations in machine learning.

We want to perform our memory accesses in a fashion that allows greater parallelism. GPUs give us a variant of this where memory accesses are coalesced and the hardware can perform them in parallel where the effective access speed is several times faster than between the normal CPU and RAM scenario. This makes the main bottleneck transferring data between the CPU's RAM and the GPU's RAM. To give a relative sense in this context, we see that in most cases this transfer time dominates even large 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 distributed systems 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.



## 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 around options on how to train and deploy models in production.

---

<sup>3</sup> [https://en.wikipedia.org/wiki/List\\_of\\_Nvidia\\_graphics\\_processing\\_units#GeForce\\_10\\_Series](https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#GeForce_10_Series)

# 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.

## Terminology For Passes Over Training Data

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.

## Determining Mini-Batch Size

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 [13].

### Smaller vs Larger Mini-Batch Size

If we get too aggressive and make our mini-batches too large when using stochastic gradient descent we run the risk of not increasing the maximum stable learning rate by the same factor as our mini-batch size increase.

Lower mini-batch sizes allow the network to encounter different orientations of the dataset and more quickly find convergence.



### A Good Starting Rule of Thumb for Mini-Batch Size for CPUs

In practice mini-batch sizes that are on the lower end of the 10-100 range to be better.

Note for GPUs – often use a larger mini-batch size for better GPU utilization.



### As Mini-batch Size Increases

As minibatch size increases, more computation means gradients may be “smoother” but are more costly to compute.

## Recipe for Setting Mini-Batch Size

When we have a training dataset that has a small balanced number of classes the ideal mini-batch size is often equal to the number of classes in the dataset. 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.

In other cases where the dataset properties are different we'd want to first randomize the ordering of the training samples and then use a mini-batch size of 10. We can raise the mini-batch size until it begins to slow down convergence. To summarize:

- Mini-batch size is between 10 and 100
- When we have equal number of classes the mini-batch size is equal to the outcome count (where ideally we have an example of each in each mini-batch)
- For a default mini-batch size we use 10

## Mini-Batch Size and Learning Rate

The learning rate is a key hyperparameter in stochastic gradient descent. We see typical starting values with standardized inputs in the range of less than 1.0 and greater than  $10^{-6}$ .



### Adjusting for Mini-Batch Size

A helpful trick to avoid issues with learning rates in the context of different mini-batch sizes is to divide the total gradient computed on a mini-batch by the size of the mini-batch.

In this context we assume that learning rates are multiplied by the average per-case gradient computed on a mini-batch and not the total gradient for the batch.

# How to Use Regularization

Regularization helps drive down weights in the parameter vector that become too large too fast. In some contexts or machine learning textbooks weight decay is referred to as regularization. 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 reduce different parameter's value to zero. 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 Typically performed through a regularizing function such as the L1 or L2 prior functions. In some cases these functions are combined (“elastic net”, Zou and Hastie 2005) which is common for deep networks [15]. 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 [13]. L1 regularization has been consistently shown to act as a useful form of feature selection.



### A Note of Prior Function Performance

In practice we see L2 regularization give better performance over L1 outside of explicit feature selection [48].

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].

## Random Targets

The addition of random targets [define] can improve the learning process. This involves adding output nodes to the network to achieve essentially multi-task learning [11].

## 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 [15]. Dropout works by setting a unit's activations to 0.0 either permanently or temporarily.

### Settings For Dropout

For input layer neurons dropout is done with a probability 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.

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 [18]. Dropout used in deep belief network pre-training has been shown to be state of the art for MNIST as of 2015 [15].

## 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 [15]. 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.

To make up for this the practitioner should look at using 10-100 times the learning rate that was used for the standard version of the network (without dropout). Another 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 [18].



## DL4J, Dropout, and Recurrent Neural Networks

Dropout as implemented for RNNs in DL4J is done on the input connections/activations only, not the recurrent connections.

## DropConnect

Dropconnect simply sets some weights (temporarily) to zero. Dropout is related to dropout, but isn't really a variant of dropout – they operate on different things.

## Stochastic Pooling

A regularization technique for convolutional networks (specifically) where we use a form of randomized pooling for building ensembles of convolutional networks. Each network attends to a different spatial location of each feature map [15].

## Adversarial Training

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 [22].

## Curriculum Learning

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 [38]. Results have shown to have mixed success.

# An Introduction to Hyperparameter Optimization

Hyperparameter optimization is the process of changing hyperparameters (as listed previously in this chapter) to maximize model accuracy. We can manually perform the search for a good set of hyperparameters or use an automated / heuristic process to perform the hyperparameter search.



### Arbiter and Hyperparameter Optimization

For more on how to use hyperparameter search with DL4J take a look at Arbiter<sup>4</sup>.

## Manual Search

A general heuristic for manually tuning back-propagation:

- Always change only one hyperparameter at a time
- Some variables such as learning rate are more sensitive to small changes than others and require more tuning

---

<sup>4</sup> <https://github.com/deeplearning4j/Arbiter>

- Sometimes it helps to do a coarse manual search of hyperparameters and then come back and do a more fine tune search in the most promising ranges [48]

### Keep In Mind When Tuning Hyperparameters

Different hyperparameters matter for any given dataset and a small subset of hyperparameters matter for any specific dataset [50].

## Random Search

Grid search can get computationally expensive (too many variations of hyperparameters) so we can approach it differently with random search. The general rule for generating hyperparameters is to use a uniform sampling distribution for each parameter in log space [13]. It can also be a good practice to have a second pass of random search that searches around the more promising regions found in the first pass.

## Other Tuning-Related Topics

In this section we quickly review other topics relevant to tuning neural networks.

### Visualizing Learning in Deep Belief Networks

At the start of learning the reconstruction error on the whole training dataset should fall quickly and consistently in the beginning and then slow down as training progresses.



#### High Momentum

If momentum is turned up high we sometimes see gentle oscillations in the loss score for a few mini-batches.

### Unsupervised Learning

Reconstruction error reduction does not always mean the model is improving and conversely small increases do not necessarily mean the model is getting worse. This is due to how Restricted Boltzmann Machine training works under the hood. However, we can say that large increases in reconstruction error are a bad sign except when they exhibit only temporarily.



## Temporary Spikes in Loss Scores

Temporary spikes in loss function scores can be due to changes in learning rate, momentum, weight-cost, or sparsity meta-parameters.

A robust method for understanding what is going on during the learning process is visualizing different aspects of the network during training.

## Debugging Unsupervised Learning

In general we consider visualization of filters to be useful as visualizations can detect complicated issues in domains such as:

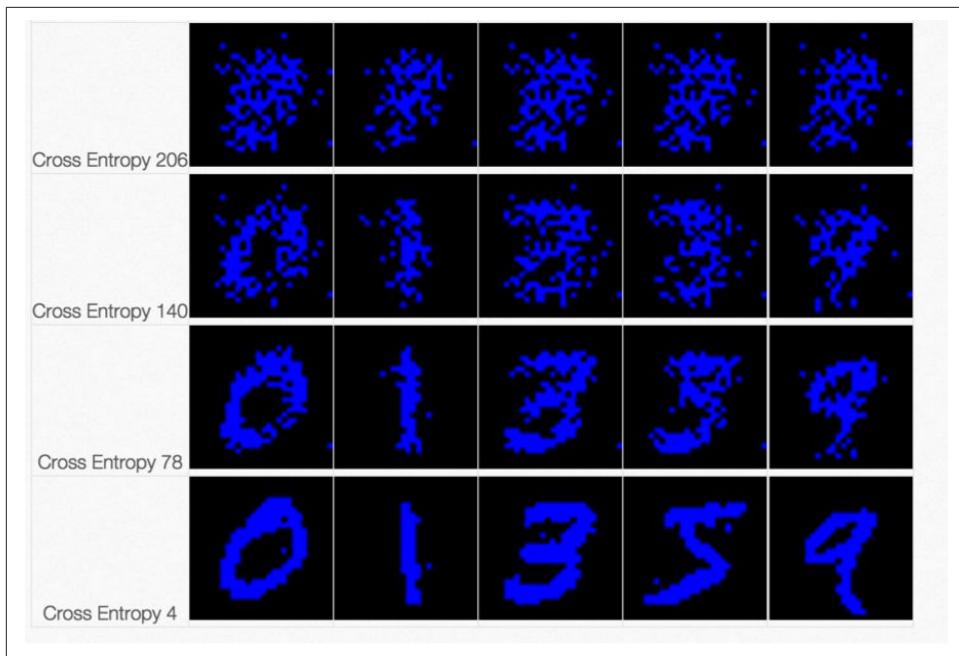
- images
- audio
- timeseries

With Restricted Boltzmann Machines, for instance, Hinton suggests generating samples from the network can be useful to understand what the model has learned [13]. Other visualizations that have shown value are:

- activations
- activation gradients
- parameters
- parameter gradients (e.g., different layers, biases vs weights, and across training iterations) [13]

## Visualizing Training Progress

There are many different variants of renders that are useful to a practitioner. Something interesting that renowned Deep Learning researcher Geoffrey Hinton has proposed is that we can take a look about what machines “dream about” in their sleep state. By that he means that we can look at what a Restricted Boltzmann Machine generates when we show it a certain input sample. Below is a sampling from the renders of Restricted Boltzmann Machines in DL4J:



*Figure 6-2. Restricted Boltzmann Machine Reconstructions*

In these renders the Restricted Boltzmann Machines are learning representations of the MNIST dataset. As we can see in the learned filters above, portions of digits are clearly visible. What we want from the Restricted Boltzmann Machine in Deep Learning is for them to learn progressively higher-level representations of the input dataset in the pre-train phase. We want to know what the machine “dreams” when we stimulate it with a certain input. When monitoring training we have three main things that we can render to give us insight into what is happening beyond simply monitoring the reconstruction error.

- Filter Renders
- Activation Renders
- Histograms

We use filter renders to see what features are being learned. We use activation renders to understand the quality of the learning with respect to the hidden activation probabilities. We can use histograms of the weights, the visible biases, and the hidden biases to understand how well the weights are converging during training.

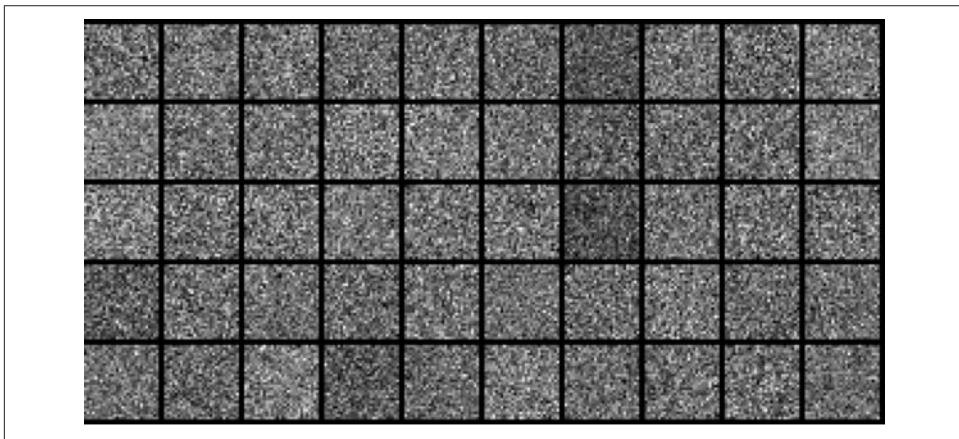
### **Rendering Filters**

In certain domains such as images or speech it is helpful to render an image of the weights for each hidden unit. We do this by rendering a separate image for the

weights per hidden unit where if we had 10 hidden units we'd have 10 rendered images. In some contexts these images are referred to as "receptive fields" and they can give us an idea of what features have been learned by the hidden units. As we can see in the image below renders of receptive fields are typically done in gray scale, as these are more informative than false color renders. The prime reason to render a filter is to get an idea if the feature detectors are picking up relevant features or not.

When we are modeling images as input data this becomes a more relevant exercise as the naked eye can see features such as edges of objects or parts of hand-written digits emerge in rendered filters. To generate a filter for each neuron in a Restricted Boltzmann Machine as an image we perform the process as follows. Visible units correspond to the input data or pixels of the training image data. We want to know what data will drive a feature detector to activate most strongly (the unit would have the value 1). To better understand how this happens we render the hidden unit's connected incoming weights and plot them in the same shape as the input data. Input data that matches this activation filter will drive the hidden unit to have the value 1.

Each column of the weight matrix for the Restricted Boltzmann Machine corresponds to a hidden unit. We also refer to these hidden units in Restricted Boltzmann Machines as feature detectors. This will be clear in a moment when we see specific features emerge in the filter renders. We plot each of these columns separately yet we format that in the shape of the original input data. Each filter is of the same dimension as the input data as it is most applicable to see the filters in the same way the source image data is visualized. Each pixel in the render corresponds to the weight value in the hidden unit. Renders of hidden units can be done many ways but typically we see them lined up in a grid of patches as shown below in figure X.



*Figure 6-3. Filters as Initially Rendered*

As we can see above in the filter render at first the filters look like the static on your television with the signal lost. As training progresses we can see recognizable images

or features emerge from this static as we see below. Input images with these features will tend to drive the hidden unit to have a value of 1.0.

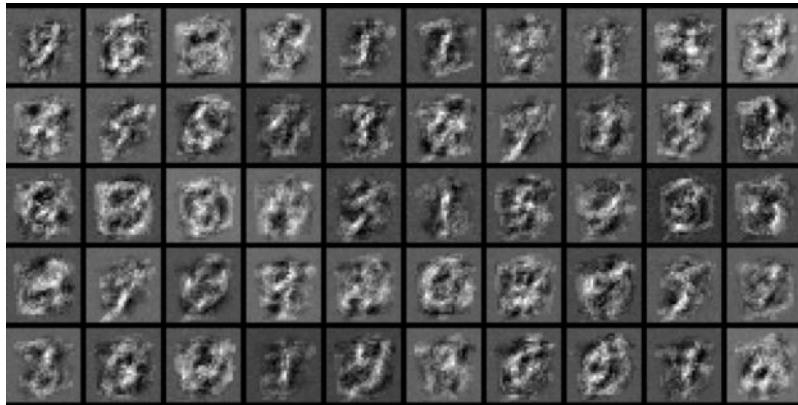


Figure 6-4. Emergence of Learned Features Visible in Rendered Filters



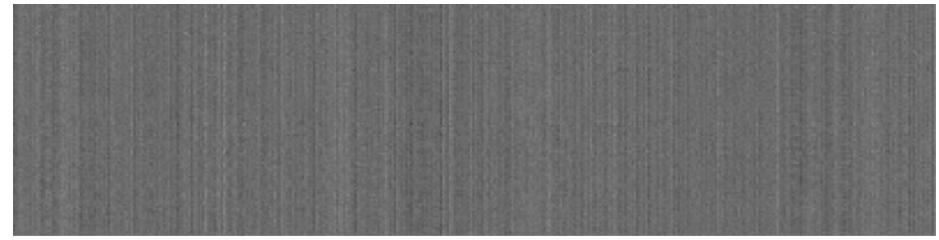
### Interplay of Hidden Units, Learned Filters, and Complex Functions

As we have fewer hidden units we see the filter renders will have more complex features. As we have more hidden units we will see that the generated hidden unit features to be simpler. This illustrates the interplay of more parameters in a neural network being able to model more complex functions.

### Rendering Activations

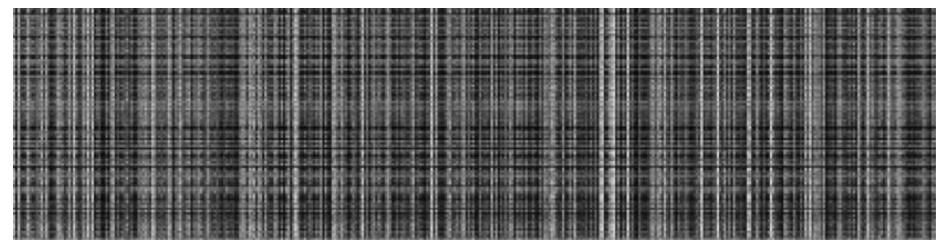
For an input vector each hidden binary neuron has a probability of being set to 1.0 (or being turned on). The value is always either 0.0 or 1.0 yet it is useful to see how the probabilities look behind these binary states. To get a feel for these probabilities we make renders of the probabilities of hidden activations. If we want to see if a hidden unit is never used or if some training cases activate a suspiciously large number or small number of hidden units we would look at the render for a single mini-batch. This also gives us information on how sure certain hidden units are of their state.

The prime reason to render activations are to see if certain hidden units are never used or if some input records drive activation in a suspiciously large or small number of hidden units. To produce these activation renders we take each column in the render represents the weight vector for hidden unit.



*Figure 6-5. Activation Render at Initialization Time*

At the start of training we see very slight vertical lines in an otherwise plain grey render. As training progresses we see vertical and horizontal lines emerge.



*Figure 6-6. Activation Render During Training*

As training lowers the cross entropy towards its convergence point we see these lines disappear and a healthy random black and white static pattern emerges.



*Figure 6-7. Activation Render at 4.9 Cross Entropy*

As we can see in the image above there are no overt horizontal or vertical lines.

### **Rendering Histograms**

Another useful render is plotting the values of the visual biases, the weights matrix, and the hidden biases as histograms. It is pragmatic to understand visually how the weights in the RBMs change during training in the form of a histogram. During the middle training all of the histograms should look roughly Gaussian in shape. In cer-

tain cases the values of the weights will diverge into two separate Gaussian shaped clusters and converge to one cluster later (this does not tend to have adverse effects). At the end of training the histogram of the weights should cluster more values around a smaller range of values as we can see below. The mean magnitudes of the final histogram should be smaller than the starting histogram by a factor of  $10^2$  to  $10^4$ . If the training process doesn't produce weight changes that are large enough then this indicates a need to increase the learning rate (when adagrad is not used). If the change in weights is too large then learning will have issues where the weights diverge to infinity.

In the image below we can see a Restricted Boltzmann Machine's weights plotted at initialization time in a histogram:

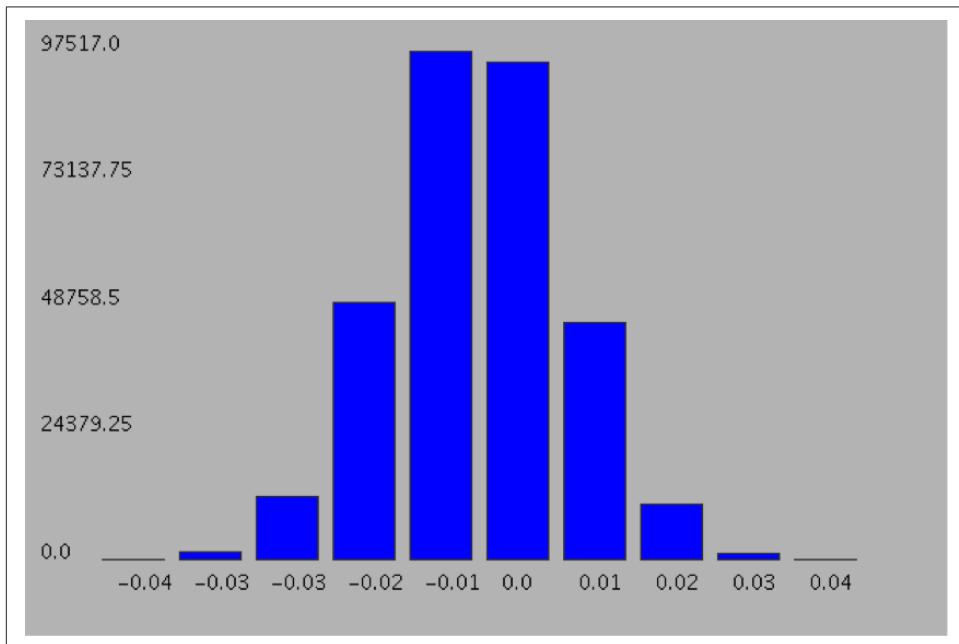


Figure 6-8. Histogram of Restricted Boltzmann Machine Weights Initially

And then as we train we can see how the histogram changes in the image below:

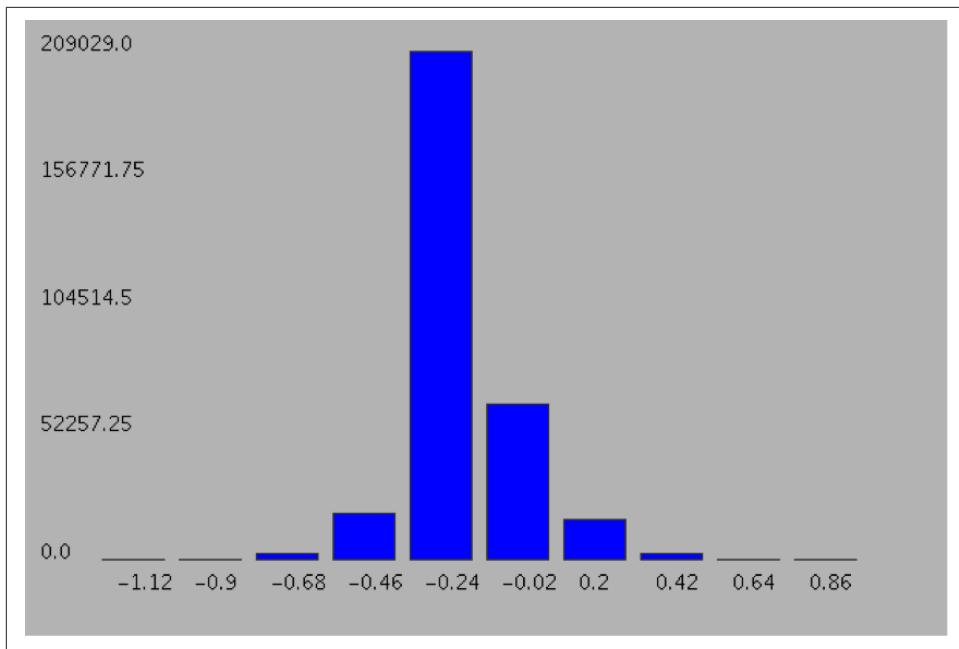


Figure 6-9. Histogram of Restricted Boltzmann Machine at End of Training

The shape of the histogram indicates better training results and provides us with yet another visualization tool to debug training.

# Visually Debugging Deep Belief Networks

Below are some basic notes about triaging issues in Deep Belief Networks.

**Monitoring Overfitting Visually Using Histograms.** If the learning rate is too large the reconstruction error may rapidly rise and the weights become too large [14].

**Using Reconstruction Error.** We want to see reconstruction error fall rapidly early on in learning and then more slowly as we converge [14]. Reconstruction error is a poor measure of the progress of learning in general.

**Visually Analyzing Hidden Unit Weights.** It's useful to look at the weights connecting to the hidden units (per hidden unit) for spatial or temporal domains. The receptive fields of the hidden units can be rendered such that we have a visual clue as to what they are learning

**Visualizing During Training.** It can be useful to visualize activations per mini-batch during training to see if any hidden units are never activating. A healthy visualization should look thoroughly random without any vertical or horizontal lines

## Evaluating a Neural Network Model

The simplest way to evaluate any neural network classifier is to use the F1 score we introduced in chapter 1. As we'd do in many machine learning workflows we hold out a test dataset from the training. We can get a better score for the model if we use N-fold cross validation and average the N F1 scores together. In the next chapter on working with DL4J we'll see this in practice.



### A Caution on N-Fold Cross Validation

Cross validation is costly (train the model separately N times) – for large models it may not be used for that reason.

## Training, Validation, and Test Data

One of the most common ways to train a model is to use training, validation, and test datasets.

**Training Dataset.** We use this part of the dataset to build the model.

**Validation Dataset.** Periodically (typically after each epoch) we will evaluate the trained model and produce a confusion matrix. Evaluating the model on the test dataset biases our training process towards learning to score well on the test dataset and less on generalizing well to the population of potential data. We instead hold out another split of the data called the validation dataset for the purpose of periodically evaluating the model to track our training progress.

**Test Dataset.** This is the held out split of the data that we use to evaluate the model once we determine training is complete.

## 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.



### Overfitting in Machine Learning

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.

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.

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.



### Overfitting Rule of Thumb

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

## Finding the Sweet Spot in Training

To find this inflection point, add parameters to the model in successive training runs until the holdout data accuracy begins to diverge. At that point back the parameter count down to the last known good point. N-fold cross validation (where typically  $N = 10$ ) will help get a better score averaged across parts of the held out training dataset and help find the optimal parameter count to make the model generalize more effectively.



### Complex Deep Learning Modeling and 10-Fold Cross Validation

10-fold is often impractical in practice. Complex, large deep networks on large data sets can take days to train.

## Dealing with Class Imbalance

In use cases where the positive class is relatively rare (example: fraud detection or ad click through prediction) dealing with class balance can be difficult for neural networks. To combat this issue we can use multiple variations of “class probability scaling” to improve performance of the model [9].

## Prior Scaling

Scale the weight update on a class based on the relative number of instances in that class for the training dataset. It's based on weighting the loss function and hence the gradients.

## Post Scaling

Train network normally but scale outputs post-training. The results are less consistent compared to the other methods.

## Probabilistic Sampling

Select samples by first randomly (e.g. through some probability) selecting a class then randomly select a sample in the class.

## Balancing Class Membership

Sample both classes such that the frequency of the larger classes are reduced to be similar in size to the other classes.



### Supersampling

We can also duplicate the lower frequency classes to “supersample” them up to the count of the other classes.



# 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:

- Deep Belief Networks
- Convolutional Neural Networks
- Recurrent Neural Networks

Let's now dig into tuning Restricted Boltzmann Machines in the context of tuning Deep Belief Networks.

## Restricted Boltzmann Machines

Restricted Boltzmann Machines have visible and hidden units. These units are used for reconstructing data. Restricted Boltzmann Machines hidden and visible units are used for activations and learning reconstructions.

## Intuition

Typically we're focused on using Restricted Boltzmann Machines for use cases such as:

- data reconstruction
- recommendation engines
- feature learning in Deep Belief Networks

A great way to understand them in practice is to consider the case of modeling the MNIST dataset.

### 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 the cases where we'd use more parameters than training cases we see severe over-fitting in the model constructed.

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.

### Available Information

Using the raw information contained implicitly in the input data we can leverage this information to more effectively extract features and/or allow for more effective dis-

criminative 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.

### Using Different Types of 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:

- Gaussian Visible Units
- Binomial Units
- Rectified Linear Units

Multinomial units are for building a classifier with a Restricted Boltzmann Machine.

**Gaussian Units.** When dealing with images or speech we find that binary 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 [14]. In some cases these units have shown to be unstable during learning.

**Binomial Units.** When dealing with sparse data we want to use binomial units.

**Rectified Linear (ReLU) 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 [14].

Below in table X 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

## Regularization

Regularization is an important topic in machine learning as we want to control the size of the weights in our parameter vector. Reasons for using regularization with Restricted Boltzmann Machines are listed in the following sections.

**Improving Generalization.** The first is to improve generalization to new data and lessen the chance of overfitting on the training dataset.

**Reduce Useless Weights.** Another reason is to make the receptive fields of the hidden units smoother by reducing the useless weights.

**Unsticking Weights.** A third reason 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.

Since the regularization function is applied to every mini-batch, we ideally want to divide the weight cost by the size of the training set. This gives us weight-decay as the effect of a Gaussian weight prior whose variance is independent of the mini-batch size. In the DL4J library this is automatically performed for the user.

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 your training process performs.

### **Summary and Other Notes**

When dealing with missing values in RBMs there exist a few techniques we'll reference [14] but won't list explicitly. Its also been shown in [18] 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.

### **Pre-Training with Restricted Boltzmann Machines**

To train Deep Belief Networks we have to train Restricted Boltzmann Machines. Research indicates we see the advantages of pre-training as the depth of our network increases [15]. It can be a challenge, however, to select the best hyperparameters for the pre-training phase.

### **Initializing Weights**

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 [14]. Hidden bias values of 0 are usually fine.

### **Setting the Learning Rate**

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.

## **Customizing the Learning Rate**

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. As fan-in to the unit becomes large we'd ideally like the updates to be smaller. For biases the updates can be bigger [14].

## **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.

### **Momentum During Pre-Training**

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 [14].

### **Rule of Thumb for Momentum**

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).

## Dropout

When dropout is used for pre-training [17] we want to use a small learning rate with no weight constraints to avoid losing the feature detectors discovered by the pre-training technique [17]. 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 [17]. 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.

## Sparsity

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 [14]. The decay rate of the factor should be set between 0.9 and 0.99.

## 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. The aspects we consider for this are listed in the following sections.

### Decreasing Counts Over Successive Layers

Take the previous number and use a number of parameters an order of magnitude smaller.



### Smaller Sparsity Targets

If we are using a sparsity target that is small then we can use more hidden units.

## Effects of Larger Training Datasets

In larger training datasets we can see records that are highly redundant. In these cases we can lower the number of parameters [14].

# 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.

## Intuition

The typical convolutional network layer structure has 3 stages [15]:

1. Convolution Stage
  - Affine transformations
2. Detector Stage
  - e.g. typically “Rectified Linear”
3. Pooling Stage

We see this pattern repeated in different variants of the Convolutional Neural Network architecture.



### 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.

### Convolution Stage

The convolution stage uses multiple filters to learn different features from the input to the layer as we previously outlined in chapter X. The output of this stage is transformed with an activation function (e.g. rectified linear activation function).

## Pooling Stage

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 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. 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 [15].

The most common pooling functions are:

- Max pooling
- Average of rectangular neighborhood
- L2 norm of a rectangular neighborhood
- Weighted average based on distance from central pixel

## Working with Spatial Arrangement

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 three key hyperparameters are:

- Depth
- Stride
- Zero-padding
- Filter Size

We review each below.

## Depth

The depth hyperparameter controls the number of neurons in the convolutional layer, which connect to same part of the input volume. Different input features may activate different neurons along the depth dimension. A depth column is a set of neurons that are all connected to the same region of the input volume.

## Stride

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.

## Zero-Padding

We want to preserve the spatial size of the input volume. To do we use zero-padding to control the spatial size of the output volumes.

## Formula for Calculating Spatial Size of Output Volume

The formula we use on how to calculate the spatial size of the output volume as a function of the input volume size:

$$\text{Output volume size} = (W - F + 2P)/S + 1$$

Where

*Table 7-1. 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



### Integer Result for Spatial Size

The result of this equation should be an integer and if it is not then the configuration is incorrect.

We might be able to fix it with padding, or different filter size.

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 for this (reasonable currently but continuing to improve it). 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.

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. Using zero-padding improves performance in convolutional networks in addition to keeping spatial sizing constant after convolutional layers. Zero-padding also preserves input information along the borders. Other tricks for improving convolutional network training include whitening the data (eg: PCA) and then using Adagrad for the learning rate.

## Configuring Filters

For our convolutional layers we want to use small filters (3x3 or 5x5) and a stride of 1. We often want to pad the input volume with zeros such that the convolutional layer does not change the spatial dimensions of the input.

### Choosing Number of Filters

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.



### Setting Filter Count in DL4J

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.



## 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.

**Calculating the Filter Count for Convolutional Layers.** The easiest way to get started setting up filters in Convolutional Neural Networks is to start with the design

### Configuring Filter Shape

We want to select a feature size that is obviously smaller than the input volume spatial dimensions yet big enough to capture relevant features.

#### MNIST Input Images

A good example is MNIST input images have a spatial dimension of [28x28] and generally have [5x5] sized filters on the first convolutional layer.

**Configuring Filter Shape.** The easiest way to get going on filter shapes is to start with modern/common architectures like VGG<sup>1</sup> or GoogLeNet.



### A Caution on Filter Sizes

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 you can set your image size after the fact to help things to work better (e.g. “crop or scale to some size”).

### Filter Size in Relation to Convolutional Layer Count.



Smaller filters in multiple stacked convolutional layers tend to perform better as opposed to big filters in fewer convolutional layers.

---

<sup>1</sup> <http://cs231n.github.io/convolutional-networks/>



### A Caution on Filter Size from the v3 Inception Paper

Convolutions with larger spatial filters (e.g.  $5 \times 5$  or  $7 \times 7$ ) tend to be disproportionately expensive in terms of computation. For example, a  $5 \times 5$  convolution with n filters over a grid with m filters is  $25/9 = 2.78$  times more computationally expensive than a  $3 \times 3$  convolution with the same number of filters.<sup>2</sup>

The authors go on to state:

“The above results suggest that convolutions with filters larger  $3 \times 3$  might not be generally useful as they can always be reduced into a sequence of  $3 \times 3$  convolutional layers.”

## Common Convolutional Architectural Patterns

Below we review a few design patterns to achieve specific goals with convolutional neural networks.

### Simple Linear Classifier

Re-implementing a basic neural network in the context of convolutional design is a good way to show a baseline to build from. To build a simple linear classifier we'd stack the following layers:

- Input Layer
- Fully Connected Layer

Now let's evolve this design into a more complex neural network.

### The Common Convolutional Layer Pattern

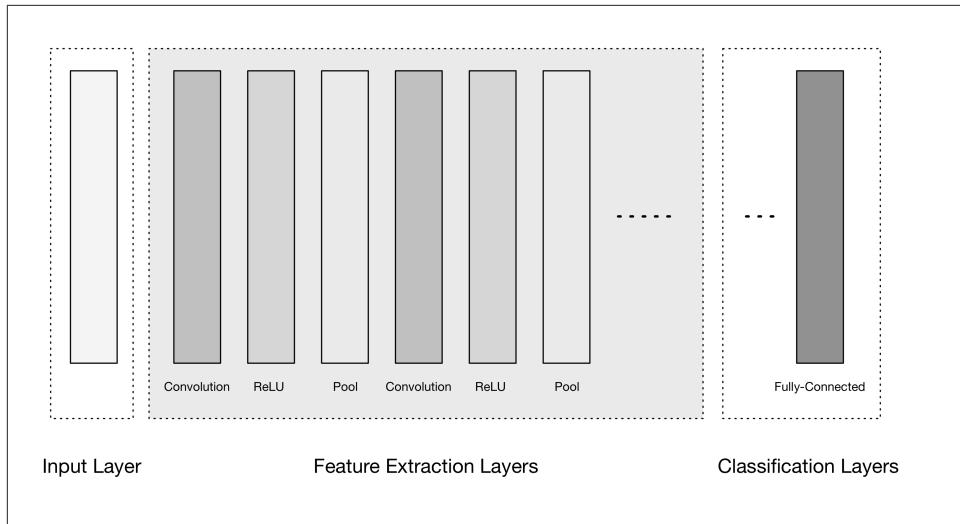
We often see a single convolutional layer between every pool layer. We see a common convolutional network pattern below:

- Input Layer
- Convolutional layer
- Pooling Layer
- Convolutional layer
- Pooling Layer
- Fully Connected Layer
  - ReLU Layer [ *re-check this!* ]
- Fully Connected Layer

We see the above pattern illustrated in the diagram below:

---

<sup>2</sup> <https://arxiv.org/pdf/1512.00567v3.pdf>



*Figure 7-1. General Convolutional Architecture*

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 downsample with the pooling layer.



### Designing Layers and Filters

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.

### Example: Building LeNet

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.

- Input Layer
- Convolutional layer
  - 4 filters [5x5]
- Max Pooling Layer [2x2]
- Convolutional layer

- 6 filters [5x5]
- Max Pooling Layer [2x2]
- Fully Connected Layer

## Configuring Layers

Let's now look at some best practices for setting up input, hidden, and output layers in convolutional neural networks.

### Input Layers

The input layer will have the same number of units as the input image (or vector). We want this input layer size to be divisible by 2 more than once so we have enough information to downsample 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.



#### Some Caveats About Input 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)<sup>3</sup>.

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).

### Hidden Layers

The hidden layers of a Convolutional Neural Network will (generally) have a repeating sequence of convolution and pooling layers.

**Convolution Layers.** We want to setup the spatial arrangement of the convolutional layer and then choose the number of filters. Both were described in a section previously.

**Pooling Layers.** We typically set the max pooling shape (e.g. spatial dimensions: width, height) of the downsampling operation in pooling layers as [2x2].

---

<sup>3</sup> <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>



## Using Max Pooling

If we don't use [2x2] then we won't use max-pooling.

Pooling is still used in conjunction with 3x3 convolutional layers and above in practice. Examples of this include AlexNet and VGG (and inception architectures too)<sup>4</sup>.

If our input data is considered larger than normal then we may increase our spatial dimensions to [4x4] in earlier pooling layers of the network. 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.

## Output Layer

Our output layer is going to follow the patterns set forth in the previous chapter on general network architecture tuning. Common output layers for Convolutional Neural Networks are softmax and sigmoid layers.

# 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.

---

<sup>4</sup> <http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/>



## 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.

## Intuition

There are some basic variations in how we train and tune Recurrent Neural Networks. In this section we'll give the reader a basic intuition on the interplay of some of the different ways we can influence training.

### Mini-Batch Size

Mini-batch size controls how many examples of data are processed at the same time. As mentioned previously in this book, we use mini-batching for training as it allows for batched execution of linear algebra operations and more efficient use of hardware.

### Sequence Length

The sequence length specifies the length of each chunk to train the network on. This tells us how many time steps the gradient signal will travel for training purposes in the model. It's the limit of the number of dependencies along the time dimension that we can model with the recurrent neural network.

### Example

If the sequence length is 20, then the gradient signal will never backpropagate more than 20 time steps.

### Priming

Priming is concerned with influencing the network in a specific direction during prediction of a sequence based on a trained model. We can seed a prediction with a few values in sequence (or characters) to influence the network to continue the sequence during its prediction.

An example would be if we wanted to generate a sentence that started with the word “what” we'd prime the network with the characters in the word “what” (e.g. the letters “w”, “h”, “a”, and “t”) and then allow it to predict the next N letters in a sequence.

## More General Ideas

While it is fair to say that neural network tuning involves a lot of trial and error after a while we begin to see some patterns that can help us tune things faster. We've seen too many people attempting to tune their networks by blindly changing things and it's not the best path.

Some high-level patterns with solutions are reviewed below.

**Ratio of Updates to Parameters Too High.** Reduce the learning rate.

**Connection Weights Too Large.** Increase L2 regularization.

**Connection Weights Too Small.** Might be too much regularization. Simmer down, skillet.

**Gradients Occasionally Very Large.** Sometimes gradients become occasionally very large (visible on mean magnitude graph for updates). This results in loss function values that increase for multiple minibatches. To help with this add gradient clipping or gradient renormalization.

## 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

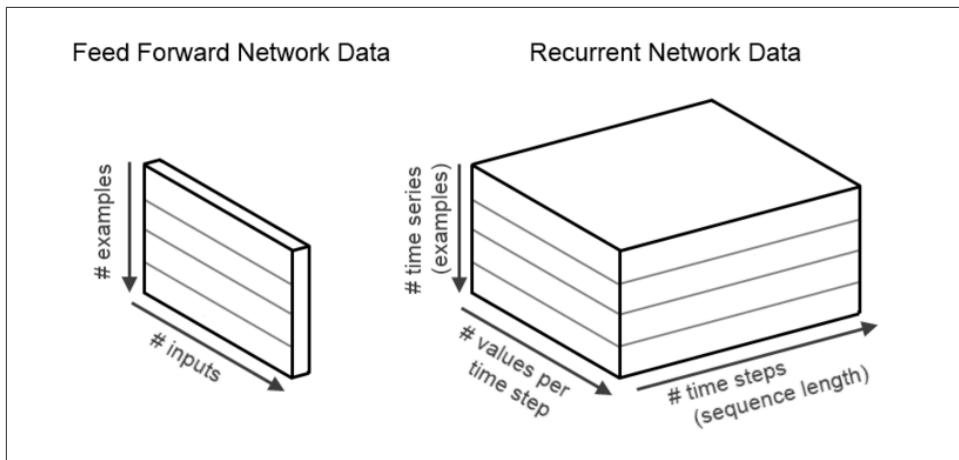


Figure 7-2. Representing 3D Input for Recurrent Neural Networks

Constructing this volumetric input can take some effort so in the following sections we'll illustrate the core concepts involved.

### Constructing Volumetric Input Data

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”.



### 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 [27].

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.

## Output Layer

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

### RnnOutputLayer

In DL4J we use the RnnOutputLayer for the final layer with many recurrent neural networks for the tasks of:

- regression
- 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 7-1. Configuration for the RnnOutputLayer*

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .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 6 where we classified sensor readings.

## 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. 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.

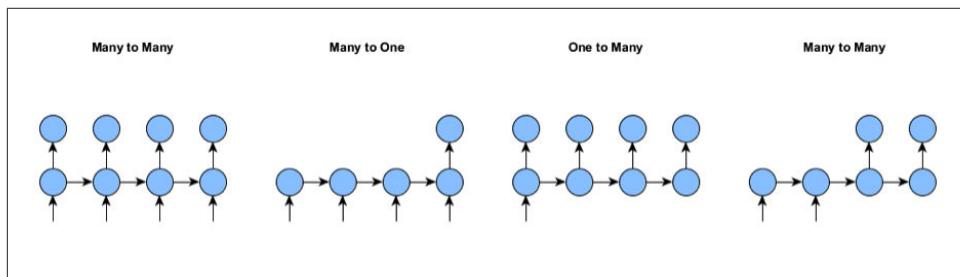


Figure 7-3. Variations of Recurrent Model Input and Output

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 value 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.

## Training the Network

Recurrent neural network training can be computationally expensive. When our recurrent network is dealing with long sequence with many time steps, we find that truncated back-propagation through time (BPTT) works well.

### Initializing Weights

Initializing weights is important in training recurrent neural networks. Random initialization of weights for recurrent neural networks has shown to provide poor training results. 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 [28].

For most Recurrent Neural Network layers in DL4J we recommend to use the Xavier initialization.

### Back Propagation Through Time

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
  - It's most common for them to be the same value

The truncated BPTT time step lengths need to be shorter than or equal to the length of the input time series



### Vanishing Gradient Issue

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].

When the gradient is too small we have the vanishing gradient issue. The opposite case is when the gradient is too large and we have exploding gradient.

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. Other methods to deal with this issue include [15]:

- Combining short and long paths in unfolded flow graph (time-delay neural networks)
- Leaky units and a hierach of different time units
- Gated Recurrent Units (GRUs)
- Better optimization methods
- Gradient clipping (for exploding gradients)
- Regularization to encourage information flow
- Using L1 / L2 penalties [34]

### Optimization Algorithms

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 [28].

## 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 7-2. Evaluation setup*

```
Evaluation.evalTimeSeries(INDArray labels, INDArray predicted, INDArray outputMask)
```

In the above example we have:

*Table 7-2. 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.

# Regularization

Dropout and momentum are common ways to use regularization with Recurrent Neural Networks.

## Dropout

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 [29]. Dropout as implemented for RNNs in DL4J is done on the input connections/ activations only, not the recurrent connections.



### 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.

## Momentum

Recurrent neural networks are more sensitive to learning rate and momentum settings than multi-layer perceptron networks [28].

## 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 preprocessors can be added manually, but in many cases they will be added automatically by the network.



# Vectorization

## Introduction to Vectorization in Machine Learning

This chapter is meant to serve as a set of guidelines for vectorizing different kinds of data used in the machine learning landscape. You 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.

## A Missing Link

Over the years it's become very 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. When attempting to work with the 20newsgroups dataset the user has to parse a lot of raw data in a bunch of directories. This raw text data has to be vectorized and arranged into some new format usable by the intended target tools the user might be focused on. This whole operation 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 vector. In machine learning we touch many types of data (text, timeseries, audio, image, video) but raw text in particular seems to give the average analyst the most problems from a machine learning and vectorization standpoint. Today's analysts and data scientists are working with text more than any other data type.

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. So why can't we just feed raw data to our learning algorithm and just let it handle everything? In some cases we can if the tool was built that way. However, as tools in the ecosystem mature and we want to use more heterogeneous workflows, interoperability becomes more important than ever. Interoperability can mean many things in these workflows (PMML for models, execution environment, etc) but for now we're going to focus on it in terms of vectorization.

```
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. We want our machine learning algorithm's input data to look more like the serialized sparse vector format below:

```
1.0 1:0.7500000000000001 2:0.4166666666666663 3:0.702127659574468 4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.4583333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.3333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0.5652173913043479
1.0 1:0.9166666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0.5652173913043479
0.0 1:0.0833333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149 4:0.5217391304347826
0.0 2:0.7500000000000002
```

## Converting Raw Data Into Vectors

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.

## Understanding Raw Data

When we vectorize a document we typically see a small subset of the words in our total vocabulary used in the corpus. The resultant matrix from this vectorization process will have many feature values containing zeroes --- with each column representing a word in the corpus's vocabulary, even with a small vocabulary most words don't appear in any given document. This is common enough that in a lot of cases we see that 99% of features contain zeros. If we look at the case of vectorizing tweets we see an example case where we'd have 100,000 short text documents. If our vocabulary is 100,000 unique words we'd see a contrast where each tweet only uses around 10 unique words. To give us storage and speed efficiency we want to use a sparse matrix/vector implementation not only for memory efficiency but also disk usage practical-

ity. This helps us speed up algebraic operations in our linear algebra routines and touch far less disk.

Major things to consider when building models from raw data:

1. What kind of source data are we dealing with?
2. What's our approach to vectorizing the data?
  - a. Are we going to hand encode the features or use an algorithmic approach?
  - b. Raw text tends to be more cumbersome to deal with, how do we handle it?

## Strategies For Dealing with Raw Data Attributes

Source data 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.

### 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. Examples of ordinal values include gender (“male” vs “female”) and location (“US”, “EU”, “Africa”). These values end up being converted into either integer values ( US = 0, EU = 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.

## 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. Ratio values are measured based on a zero point and are treated as real numbers. In this context, math operations make sense. Later on in this chapter we'll see a vectorization system that allows us to express column transformations based on the types of data in each column.

## Ratio

scheme defines a zero point and then a distance from this fixed zero point

# Transforming Raw Source Data

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.

## Selecting Features

Datasets such as the Iris dataset (*ref: diagram N above*) have a mixture of raw attribute types and need pre-processing. 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 (or "feature engineering") 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.

## 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).

### Cleaning Data

Data cleaning is a pragmatic step here and takes a good chunk of time for most data scientists. 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.

## Traditional Feature Engineering Techniques

In standard vectorization we create a fixed size n-length vector (where  $n - 1$  is our feature count and the last slot is dedicated to the label value) and then set the values of each indexed cell according to some heuristic that we came up with to represent our data. The process of vectorization is based around selecting attributes and finding a dimension in a feature vector to assign the feature to. 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". Techniques of feature engineering include

- Taking the values directly from the attribute unchanged
- Feature scaling, standardization, or normalizing an attribute to create a feature
- Binarization of features
- Dimensionality reduction

We provide some details below on these techniques.

### Feature Copying

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.

## Feature Scaling

Feature scaling is a technique that involves standardizing the range of the values for this attribute in the dataset. This is sometimes referred to as data “normalization” and is in the domain of the data preparation stage of machine learning, but it is performed after the record has been vectorized. We also want to differentiate between normalization at the feature level and then normalization at the entire vector (or “example”) level. They are similar operations but have different effects on the resulting vector. We’ll refer to the operation at the feature level as “feature normalization” and at the vector level “vector normalization”.

Feature scaling can improve convergence speed of certain algorithms (example: stochastic gradient descent) while preserving the same results. In other cases we see where techniques such as Support Vector Machines see unintended changes in the result when feature scaling is used. Methods of feature scaling include rescaling, standardization, and scaling to unit length. Rescaling a vector is defined as adding, subtracting a constant value from the value in the vector. We then would multiply or divide the value in the vector by a constant. We do this to scale the entries in a vector as you would do to change the units of measurement of the data.

A great example of this would be to convert a value’s units from miles to kilometers. Another example of this would be scaling a feature on the range from 0 to 1, which is a common approach. This class of scaling gives us an effective way to note small standard deviations in certain features while sustaining the zero entries in highly sparse datasets. The argument for not normalizing a feature is that centering some features can degrade sparsity for the value.

## Standardization

When we “standardize” a vector we subtract a measure of location (minimum, maximum, median, etc) and then divide by a measure of scale (variance, standard deviation, range, etc). We’d do something like “standardizing” a vector when we want to obtain the “Standard normal” of a vector containing random values with a Gaussian distribution. Standardization increases the effectiveness of machine learning algorithms in many cases, shaping the data to be more normally distributed (Gaussian) with zero mean and unit variance.

## Binarization

Often as data science practitioners we disregard the shape of the distribution and remove the mean value of each feature to transform the data. Dividing each feature by its standard deviation then scales this value. Some machine learning algorithms,  $\ell_1$  regularizers of linear models for instance, operate under the assumption that all vector features are centered around zero. These algorithms are also sensitive to features

variance having varying orders of magnitude which can cause them to dominate said objective function, altering the learning effects in problematic ways.

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. There are some cases in text modeling where we see practitioners use binary feature values as opposed to TF-IDF valued features. However TF-IDF based techniques have been shown to perform better in practice.

## Normalization

In many cases the attributes need some level of normalization or feature scaling to help the learning algorithm operate more efficiently. 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 normalize at the vector level most of the time this ends up being a division of the vector by a norm (in this case, “length”) of the vector. In linear algebra a norm is a function that assigns a strictly positive length to each vector in a vector space, other than the zero vector (which has zero length assigned to it). Dividing a given vector by its length normalizes the vector. Scaling by norm yields unit norm in the space of that norm. We do this normalization to make the Euclidean norm of the vector equal to 1. Euclidean norm is simply one such norm (L2 norm) and is commonly used. Normalization is the process of scaling each input vector to have unit norm. The Euclidean norm of a vector corresponds to the length of vector  $x$  in a natural way.

In most situations this is the most use norm even though this is just one of many vector norms. In the vector space model we see often in text modeling normalization is routinely applied for text classification and clustering approaches. In the world of neural networks we often normalize an input vector by rescaling by the minimum and the range of the vector. This gives all values in the vector a value between 0 and 1.

## Zero Mean, Unit Variance

When we have sparse data we want to scale features to fall between zero and one to represent probabilities. 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 range of (-1, 1) gives a broader representation of our data than (0, 1) due to more bits of information in the floating point representation.

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*).

## 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 deep autoencoders encoding raw data into shorter vectors of numbers. This smaller representation then can be used in classification or similarity search algorithms as input. Later in this chapter we’ll also see how Keogh’s SAX decomposition acts as a low pass filter on timeseries data to achieve dimensionality reduction.

## Deep Learning and Feature Learning

Deep Learning has shown that we can do less of these specific feature engineering pre-processing steps and let the dynamics of deep belief networks of recurrent neural networks discover in an unsupervised fashion the best representation of the data. Deep Learning is also adept at performing dimensionality reduction in the process of learning structure in the dataset.

We still have to produce the initial raw input vector but we can relax the need to hand produce specific features and let the deep learning algorithm do more of the feature selection for us as we’ll see in later chapters. We’ve now seen many ways we can produce vectors for our algorithms, now let’s take a look at some common vector formats we’ll use later to serialize the vectors.

## Well Known Vector File Formats

We want to be able to serialize our vectorization output into a format that is well-known and portable across multiple tools. To do this we want to use open-standard vector file formats. Some of the well-known vector formats in machine learning are:

- SVMLight
- libSVM
- ARFF

There is much debate over sparse vs dense formats in the land of vectorization. The sparse format gives us a nice and simple text based serialization strategy and for now

it has worked well for practitioners. SVMLight and libSVM both save vectors in a sparse format such that they only write out entries with non-zero values.

## SVMLight

SvmLight is an implementation of Support Vector Machines (SVMs) in C developed by Thorsten Joachims (Cornell University).

<http://svmlight.joachims.org/>

The svmLight format is well-known as a way to store sparse vectors on disk. We see below the basic schema for the text file format. Any first lines may contain comments and are ignored if they start with a pound sign (#). Each of the following lines in the vector file format represents one training example and are of the following format:

```
<label> <feature>:<value> <feature>:<value> ... <feature>:<value> #comment
```

A space character separates the target value and each of the feature/value pairs. The format also imposes the constraint that feature/value pairs must be ordered by increasing feature number. Given that we only want to store feature values that are non-zero, features with value zero can be skipped. When we are serializing vectors for a binary classification task the target value denotes the class of the example. For this case we have +1 as the target value marks a positive example, and -1 a negative example respectively. Below we have an example line based on the schema for svmLight above:

```
-1 1:0.43 3:0.12 9284:0.2 # abcdef
```

svmLight is a great example of a well-known text file format for sparse vectors and is used by many tools as an open input format for machine learning.

## libSVM

The library “LibSVM” was developed at the National Taiwan University (along with LIBLINEAR) written in C++. LibSVM implements the SMO algorithm for kernalized support vector Machines (SVMs). The vector file format documented for use with LibSVM is another well-known vector file format just like svmLight. LibSVM has been ported to other languages and toolkits, most notably WEKA. We mention it here as its vector file format is well known and worth discussion. The schema of a libSVM file:

```
<label> <index1>:<value1> <index2>:<value2> ...
```

Data is stored in a sparse array/matrix form in the row oriented schema of libSVM which gives us only the non-zero data being stored on disk. Any missing data would be considered by default to have a 0.0 value.

Example serialized vectors in LibSVM format:

```
-1 3:1 11:1 14:1 19:1 39:1 42:1 55:1 64:1 67:1 73:1 75:1 76:1 80:1 83:1  
-1 3:1 6:1 17:1 27:1 35:1 40:1 57:1 63:1 69:1 73:1 74:1 76:1 81:1 103:1  
-1 4:1 6:1 15:1 21:1 35:1 40:1 57:1 63:1 67:1 73:1 74:1 77:1 80:1 83:1  
-1 5:1 6:1 15:1 22:1 36:1 41:1 47:1 66:1 67:1 72:1 74:1 76:1 80:1 83:1  
-1 2:1 6:1 16:1 22:1 36:1 40:1 54:1 63:1 67:1 73:1 75:1 76:1 80:1 83:1  
-1 2:1 6:1 14:1 20:1 37:1 41:1 47:1 64:1 67:1 73:1 74:1 76:1 82:1 83:1
```

Like many line-oriented formats each line represents a record with the line (and record) terminated by a '\n' character. The first entry on the line is the label or the regression value. In the case that the record is for (multi-class) classification training the first entry represents the class id and will be an integer value. In the case that the record is for modeling a regression then the value is any real number. In the case of a training algorithm that does not use labels then the value will be ignored and can be anything (most likely 0.0).

We then see a series of <index>:<value> entries where the index represents the vector location of the value in the deserialized record. For this format the indexes start at 1 and value is a real number. Precomputed kernels are an exception here as their index starts at 0. This format expects the indices to be in ascending order but depending on the deserializing algorithm the effects of out of order indices may not be an issue. For files containing test data in this format we'll have labels included with each record yet we would only be concerned with the indexes and values to feed to our model. We'd then use the labels to score the effectiveness of our model. Again if any label is unknown in the testing file we can fill it in with any arbitrary real number.

Another interesting aspect of the libSVM format is that we do not see every index in the format as it is a "sparse" format where zero values do not need to be stored. This allows us to store less data in the serialized text format. Given that libSVM only supports real numbers if your data contains other types of data (strings, lists, etc) you will need to use some of the discussed techniques of vectorization as appropriate. Comments are supported in the libSVM format by the way of a pound sign at the end of a record before the null line terminator. The comment cannot contain any semi-colons, however.

```
2 2:1 4:9 # this is my comment
```

LIBSVM and svmLight are both well known and easy to use text file formats for sparse vector serialization.

## ARFF

The ARFF vector file format is an ASCII text file format that allows us to be more descriptive than libSVM or SVMLight about the vectors contained in the file. The file format was created as part of the Weka machine learning project which was developed by the Department of Computer Science of The University of Waikato.

<http://www.cs.waikato.ac.nz/ml/weka/arff.html>

Over time this format has evolved as Weka's needs and use-cases have evolved. In this context we consider the version (3.3) that is referenced by Ian Witten and Eibe Frank in their classic book on data mining. As we'll see the format has evolved to take into account string attributes, data attributes, and sparse instances. ARFF files contain two regions each representing a specific part of the data. The first section is the "header" information (or the metadata), followed by the actual training or testing records in the "data" section of the file. The header region of the ARFF file holds the name of the relation, a list of the columns in the data (or the "attributes"), and then each column's type. Below we see an example of what the canonical Iris dataset looks like when encoded in an ARFF file header:

```
@RELATION iris

@ATTRIBUTE sepalength    REAL
@ATTRIBUTE sepalwidth    REAL
@ATTRIBUTE petallength   REAL
@ATTRIBUTE petalwidth    REAL
@ATTRIBUTE class          {Iris-setosa,Iris-versicolor,Iris-virginica}
```

If we were to see some of the Iris data in the data area of an ARFF file it would look similar to the below section:

```
@DATA
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
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

The ARFF format permits for comments in files by prepending a percentage sign as the first character in the line. The declarations for @RELATION, @ATTRIBUTE, and @DATA are all case sensitive. ARFF is another option for vector serialization and storage and is most well known for being the primary format of the WEKA machine learning toolkit. Now that we've looked at some very common ways to serialize vectors to disk, let's move on to working with text during the vectorization process.

## Working with Text in Vectorization

We've now looked at why we need to vectorize raw data into vectors and some approaches on how to do it, but now we need to better understand the techniques used for specific data types. 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. Let's start out by looking at the fundamentals of free text and the vector space model.

## Free Text and Vector Space Model

The vector space model (VSM) is a common way of vectorizing text documents. In this model every possible word is mapped to a specific integer. If we have a large enough array 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. Lemmatization – finding the base or stem of words
4. Removing Stop Words
5. 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 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. 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).

	$T_1$	$T_2$	...	$T_t$
$D_1$	$w_{11}$	$w_{21}$	...	$w_{t1}$
$D_2$	$w_{12}$	$w_{22}$	...	$w_{t2}$
...	...	...	...	...
$D_n$	$w_{1n}$	$w_{2n}$	...	$w_{tn}$

In this example 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” 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 above matrix becomes the A in the  $Ax = b$  equation mentioned previously in this chapter.

### The Term Frequency Vector

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. TF-IDF multiplies the frequency of the word within a document by the rarity with which it

appears in the corpus overall. We'll discuss calculating TF-IDF shortly. 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. 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. We'll talk more about this later in the chapter when we take a look at "kernel hashing".

### Drawbacks of Bag of Words

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, which we'll talk about later in this chapter. In the next section we take a look at how we can better represent how often words occur relative to how important they are.

## Term Frequency Inverse Document Frequency (TF-IDF)

Term Frequency inverse document frequency (TF-IDF) is a great way to fix some inherent issues with the bag of words model. Words do not occur with the same frequency across all documents 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 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.

### Leveraging Relative Document Information

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 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 latter (IDF). To produce our TF-IDF number we start out by calculating the "term frequency" portion of the measurement.

## Weighting Words in Documents

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

$$[ \text{formula: } tf = (\text{count}(t) / \text{count(all terms in document)}) ]$$

### 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). 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. The inverse document frequency was original introduced in 1972 by Karen Sparck Jones as "term specificity".

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.

Dft document frequency – number of documents in the corpus that contain a term t

N total number of document in corpus

$$idfi = N / dft$$

$$wi = TF_i * IDF_i = TF_i * N / DF_i$$

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:

$$IDF_i = \log(N / DF_i)$$

### Computing the Full TF-IDF Score

Thus the TF-IDF weight for a word  $w_i$  becomes:

[ *eq here* ]

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.

### Summary of TF-IDF

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.

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”.

## 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 identifiers 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.

### N-grams in Practice

To understand the basic usage of n-grams let’s look at how it plays out in practice. A group of words in an arbitrary sequence is called an n-gram. A unigram is considered a single word. “Wall Street” can be considered a single unit and also be called a bigram. Trigrams would consist of three consecutive words, and so on. The “n” in n-gram indicates how many consecutive words we’re considering in our vectorization operation. Earlier in the chapter we saw how TF-IDF assumed that the words occurred independently of other words. The vectors created via this method usually lack the ability to identify key features of documents where dependency in phrases compose key information.

### Extracting Features with N-grams

In some cases using n-grams can pick up the good features (“Wall Street”) in text for compiling document vectors. In other cases we see n-grams picking up groups of words (“then we”) that yield little extra value in the document vector. Some imple-

mentations of machine learning algorithms pass the n-grams through a filter like a “log-likelihood” test. The Log-likelihood test can give us an indication as to how random the occurrence of the two words together are. The least likely n-grams are then filtered away leaving us with sets of words that provide more valuable information for our resulting vector.

### Applications of N-grams

N-Gram uses include applications in the fields of:

- Computational linguistics
- Speech recognition
- Machine translation
- Spelling correction
- Language identification
- And, in an extension of the idea of n-grams into genomics:
- DNA Sequencing

In cases where we have ambiguous input that is noisy n-grams give us a way to select better features from the noise. For example, in tasks such as speech recognition certain sounds, when isolated, sound extremely similar to one another. When we are able to generate sequence of sounds the contextual information in the preceding and/or succeeding sounds give us the extra information that helps classify it. We can use a model of sequences of how sounds tend to occur in conjunction with n-grams of these audio sounds to get a better prediction on what the individual words translate to.

We can also use a generative model of sequences of commonly occurring words in text to help confirm the predictions of the speech translation tool. N-grams are an example of a pre-processing phase that can add more passes over the input data, which can add up with respect to processing time. Now let's look at a method that uses time that is directly proportional to the input size but gives us a compact yet effective vector representation.

## 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.

We come up with a fixed sized vector that is typically smaller than the total possible words that we could index or vectorize and then we use a hash function to create an index into the vector. In comparison to TF-IDF the vector size would not be the unique word count (“input space”), but some length considerably smaller. We want to choose an array size that balances computational efficiency with modeling accuracy. Hashing data values into the fixed sized feature vector is based on similar concepts of hash functions in the canonical HashTable. A hash function is a function that maps input data to a table of fixed size. The values produced by a hash function are called keys, hashes, or hash values. We use a hash function to incorporate the data into feature vector in such a way that any kind of data maps to the fixed feature space. A simple hash function example:

h hash key

z input integer

n hash table size

$$h = z \bmod n$$

Although there is a downside with entry collisions via the hash function in kernel hashing, the reality is that these collisions occur very infrequently and don't have a noticeable impact on learning performance. Kernel hashing is known to be very effective as a single pass vectorization technique in the context of machine learning and can be coupled with pre-processing techniques in a similar fashion to TF-IDF.

## Image Vectorization

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.

	Col1	Col2	...	Colm
Row1	p11	p21	...	pm1
Row2	p12	p22	...	pm2
...	...	...	...	...
Rown	p1n	p2n	...	pmn

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.

Each format can store the image data different and perform varying levels of compression on the raw data. Images are different from text but can be more straightforward in how we vectorize them.

With text we're looking to synthesize the vector with sometimes exotic strategies to infer a text passage's meaning into an array of numbers. That can be tricky and not always straightforward. 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  $M \times N$  image and "flatten" this rectangle into a  $1 \times (m \times n)$  array.

```
Col1  Col2  ...  Colmxn  
Row1  pixel1  pixel2  ...  pixelmxn
```

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  $M \times N$  grid of pixel intensities. To feed this image into any linear algebra vector we need to arrange these pixel intensities into a vector of values more suitable for machine learning algorithms. We need to "flatten" this rectangle into a  $1 \times (N \times M)$  array per image record in our matrix.

```
Col1  Col2  ...  Colmxn  
Image1  pixel1  pixel2  ...  pixelmxn  
Image2  pixel1  pixel2  ...  pixelmxn  
Image3  pixel1  pixel2  ...  pixelmxn
```

A major part of the conversion is getting the raw pixel data out of the file format container and translated into the vector object that our learning algorithms understand. Our target output vector length needs to match up to  $M \times N$  image size (or whatever

derivative size we decided on through our feature engineering process). 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 methods of image processing sections of the image is parsed out and used as a vector by itself. 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). Later on in this chapter we'll look at general image vectorization in practice and then specifically vectorizing a custom image file format with the MNIST dataset.

## Working with Timeseries in Vectorization

Time series data is defined as a sequence of data points measured typically at successive times spaced at uniform time intervals. In the world of finance we see timeseries data in the form of the daily-adjusted close price of a stock at the NYSE. Another example would be sensor readings from phasor measurement units devices (PMUs) on the smart grid measuring “phase angle” and voltage 30 times a second.

Another interesting use case in timeseries is where we look at genetic text data (A, C, G, and T can be 1, 2, 3, and 4) through the lens of timeseries tools. Genome and Proteomic work is very interested in applications where we can find fuzzy yet similar matches.

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). Timeseries typically are a challenge to vectorize but for different reasons than raw text. We have a lot of values that can be very noisy and we need to perform a low-pass filter to keep the structure but remove the noise. In the next section we'll look at a way to deal with noisy timeseries data.

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/>

### Symbolic Aggregate Approximation (SAX)

A great way we found to deal with noisy timeseries data is to run a low pass filter over it as a pre-processing step. Symbolic Aggregated Approximation (SAX) 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”. At first glance the aspect of timeseries data being so noisy may not seem a great obstacle, but in the example story below we take a look at how this played out in practice.

## DataVec and Vectorization

For more on how to vectorize specific data types check out the appendix chapter on “Using DataVec”.



# 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 Spark and Hadoop

Two key datacenter technologies that have emerged in the past decade are both Hadoop and 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 scaleout of network training on spark – we can use this to significantly reduce the time required to train our networks. Or train larger networks with more data, in a given amount of time.



### 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.

## What is Spark?

Apache Spark is a general parallel processing engine that can execute on its own 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.

## Key Spark Concepts to Understand

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).** 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.

There are two ways we can create a RDD in a Spark program:

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, or any other system that is compatible with a Hadoop InputFormat. In practice, this is most often HDFS.

## What is Hadoop?

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 and Google File System (GFS) papers (Dean, Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”) and originally constructed to parallelize inverted index construction for the Lucene search engine as part of the Apache Nutch project. The Hadoop project was started by Doug Cutting and Mike Cafarella and ultimately kicked off the democratization of search engine infrastructure, later revolutionizing how data warehouses 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). By 2009 many companies such as last.fm, Facebook, The New York Times and even the Tennessee Valley Authority saw Hadoop as a practical way to use commodity hardware to parallelize processing on large amounts of data. 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 and Hadoop

DL4J was designed from day 1 to be a first-class citizen in the Hadoop environment.

## 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 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
- 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.

## Providing Additional Configuration

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.

## 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
```

or on HDP:

```
/hdp/todo/...
```



### 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 does not have 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

Valid starting     Expires            Service principal
07/05/16 20:39:08  07/06/16 20:39:08  krbtgt/HORTONWORKS.COM@HORTONWORKS.COM
```

# 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 which should be familiar to the enterprise user of either Cloudera's CDH or Hortonwork's HDP distributions.

## Understanding Spark Execution Modes

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.

### The Spark Driver Process

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. We can quit the client session for a background client and allow the spark job to continue to run. This active driver is used by spark to manage the job flow and schedule tasks for Spark.

**Client Mode.** The spark driver process is running on the local machine it was executed on. When you run Spark in client mode, the driver process runs locally as a master daemon which coordinates the workers. This mode typically cannot be used with secure (e.g. Kerberos-enabled) Hadoop clusters.

**Cluster mode.** The spark driver process is run on a remote machine in the cluster itself.

### 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. 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.

**yarn-cluster.** The spark driver process runs remotely on an ApplicationMaster in a YARN-based Hadoop cluster. This same container runs the Spark Application Master to coordinate the job. 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.



## Why Run Spark on YARN?

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:

- 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 to control how many executors are being used at any given time on the cluster for a single Spark application
- Kerberos support

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 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.

## Summary Table Comparing Spark Execution Modes

Below is a quick summary table to understand the effects of running Spark in different modes. This table was originally created by Sandy Ryza for Cloudera's Engineering Blog<sup>1</sup>.

*Table 9-1. Understanding the different Spark Execution Modes*

	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 and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and workers
Supports Spark Shell?	No	Yes	Yes

<sup>1</sup> <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/>

# 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.

## 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:

```
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 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 cgroups during the shuffle process in Spark. Defaults to 0.2 and we recommend staying with this setting at least early on.

**spark.storage.memoryFraction.** The parameter to control how Spark manages the total size of the cached RDD. The default value is 0.6 and controls the fraction of Java heap memory to use for Spark's memory cache. This makes sure the executor doesn't 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  $(\text{executorMemory} * 0.10)$  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. 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.memoryOverhead` together to get the full YARN memory request.

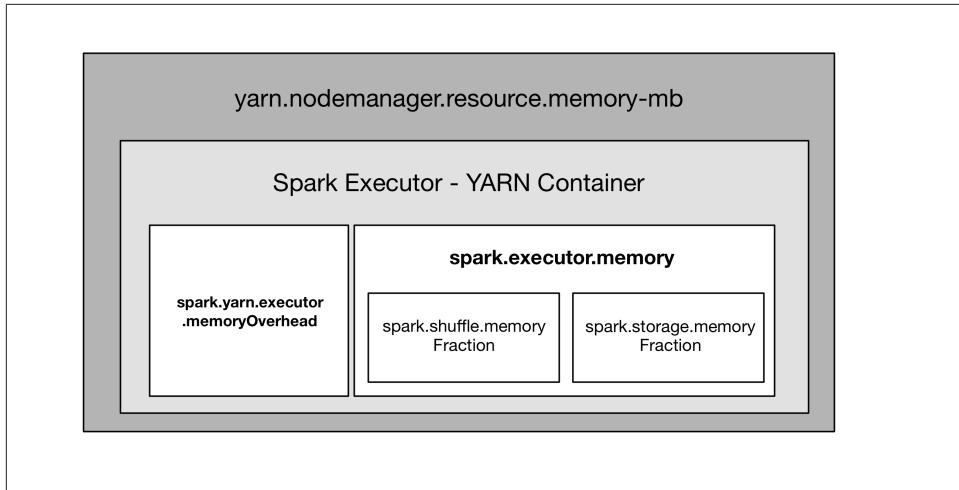


Figure 9-1. Hierarchy of Memory Configurations of Spark on YARN

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 on Cloudera's Engineering Blog for more Spark tuning tricks:

<http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/>

<http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/>

## 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 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.

## 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.

## 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 learning but also how much memory the executor will need.

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):  $28 \times 28 = 784 \rightarrow \sim 3\text{kB}$  (plus labels – 10 values) per example.

Time series with 256 inputs, 1000 time steps  $\rightarrow 1\text{MB}$  per example for features (plus labels)

## Understanding Parallel Performance

Parallel training of iterative algorithms has some minor effects on convergence. Below we list a few of the effects and discuss how to deal with them.

### Parameter Averaging and Spark

Users of DL4J's distributed Deep Learning can choose the mini-batch size and the software will optimally distribute the workload across workers in Spark. This is an optimal input split similar to Hadoop's. The system will maximize the number of processing units available to it, automatically scaling horizontally. Running DL4J on runtimes such as yarn directly or Spark on Hadoop will run a worker on 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.

## Effects of Training in Parallel on Spark

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. In this section we’ll look at some best practices when building DL4J POM files.

## Major Dependencies for a DL4J Spark Job

The major dependencies for a Spark DL4J 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.

### nd4j-native-platform

If building on one platform, deploying on another: use “nd4j-native-platform” dependency instead. Includes native binaries for all platforms.

## Understanding Versions

Depending on the version of Spark we’ll need to set the pom.xml file up differently.

### Hadoop-Common

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>

```

**hadoop.version.** Hadoop.version is going to be set in the properties area of the pom.xml file depending on the Hadoop distribution.

## Spark Dependencies Versions

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}` should be `_2.10` or `_2.11` and should match the version of Spark you are using.

**spark.version.** Spark.version is going to be dependent on the Hadoop distribution or Spark distribution we're running on.

**scala.binary.version.** This variable is dependent on what version of Spark we're using.

## 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>
    </dependencies>
</dependencyManagement>

```

```

        </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.scala-lang</groupId>
        <artifactId>scala-reflect</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>
```

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.

### Key Version Settings

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>0.5.0</nd4j.version>
    <dl4j.version>0.5.0</dl4j.version>
    <datavec.version>0.5.0</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>

```

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.

### Key Version Settings for HDP 2.4

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>0.5.0</nd4j.version>
<dl4j.version>0.5.0</dl4j.version>
<datavec.version>0.5.0</datavec.version>

<scala.binary.version>2.10</scala.binary.version>
<scala.version>2.10.4</scala.version>

</properties>
```

## 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>
```

## 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.

### Common Issues with ND4J

Below we list a few common issues encountered when using ND4J on Spark and their resolutions.

#### ND4J and Kryo Serialization

Kryo is a serialization library commonly used with Apache Spark. It proposes to increase performance by reducing the amount of time taken to serialize objects.



## SerDe

The term “SerDe” is 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. 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.

## Common Issues with Spark

In this section we list a few notes about things the practitioner may encounter in Spark usage.

### Not Enough Resources to Allocate YARN Containers

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.

## List of Key Ports for Spark and YARN

Below are some of the key ports for viewing and debugging Spark activity.

*Table 9-2. 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

## DL4J Parallel Execution on Spark

Deeplearning4j supports training neural networks on a Spark cluster, in order to accelerate network training.

Similar to DL4J's MultiLayerNetwork and ComputationGraph classes, DL4J defines two classes for training neural networks on Spark:

- SparkDl4jMultiLayer, a wrapper around MultiLayerNetwork
- SparkComputationGraph, a wrapper around ComputationGraph

Because these two classes are wrappers around the standard single-machine classes, the network configuration process (i.e., 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**. This is functionally equivalent to running Spark in local mode, though has lower overhead (and hence provides better training performance).

## Distributed Network Training

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 Training-Master
3. Iterate over the data splits. For each split of the training data:
  - Distribute the configuration, parameters (and if applicable, network updater state for momentum/rmsprop/adagrad) from the master to each worker
  - Fit each worker on its portion of the split

- Average the parameters (and if applicable, update state) and return the averaged results to the master
4. Training is complete, with the master having a copy of the trained network

## 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.

```
JavaSparkContext sc = ...;
JavaRDD<DataSet> trainingData = ...;
MultiLayerConfiguration networkConfig = ...;

//Create the TrainingMaster instance
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster = new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
    .(other configuration options)
    .build();

//Create the SparkDl4jMultiLayer instance
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, networkConfig, trainingMaster);

//Fit the network using the training data:
sparkNetwork.fit(trainingData);
```

Let's now take a closer look at how we work with the concept of DL4J's TrainingMaster.

## Working with the TrainingMaster

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> through a number of steps) then this will usually be 1

## **batchSizePerWorker**

This controls the minibatch size for each worker. This is analogous 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 asynchronously 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 `mapPartitions` operation; consequently, the number of partitions (and, the values in each partition) matters a lot for proper cluster utilization. However, repartitioning is not a free operation, 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 standard 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

Below we list a few key points for getting the most out of your Spark+Hadoop cluster.

## Slim Down the Jar

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.

## A Well-Tuned Cluster

Use Spark on a well-tuned cluster with a good distributed filesystem.



### A Note About What We Mean by “Good Distributed Filesystem”

We mean HDFS. Very plainly: don’t mess around here.

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.

## Efficient Vectorization Pipelines

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.

## A Well-Tuned JVM Goes a Long Way

Make sure the JVM is tuned such that we’re not having long GC pauses.

## Multi-Layer Perceptron Spark Example

For this example we’ll revisit our Multi-Layer Perceptron from Chapter 5 where we modeled the non-linear synthetic Saturn dataset.

## Since We Last Spoke

The major thing that has changed in this chapter is how we’re approaching parallel training on Spark. Most everything else for this example will remain the same. We’re

now working with data in HDFS and our ETL and vectorization 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 we'll demonstrate how to use anonymous functions in Spark to do generalized CSV parsing to produce NDArray vectors.



### 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.

## Spark Code

We list the code for the Saturn Spark example below.

*Example 9-1. MLP Spark Example for Saturn Dataset*

```
public static void main( String[] args) throws Exception {  
  
    Logger.getLogger("org").setLevel(Level.ERROR);  
    Logger.getLogger("akka").setLevel(Level.WARN);  
  
    Nd4j.ENFORCE_NUMERICAL_STABILITY = true;  
    int iterations = 10;  
    int seed = 123;  
    int listenerFreq = iterations/5;  
    double learningRate = 0.005;  
    //Number of epochs (full passes of the data)  
    int nEpochs = 10;  
  
    int numInputs = 2;  
    int numOutputs = 2;  
    int numHiddenNodes = 20;  
  
    org.apache.hadoop.conf.Configuration hadoopConfig = new org.apache.hadoop.conf.Configuration()  
    FileSystem hdfs = FileSystem.get(hadoopConfig);  
    String hdfsPathString = args[0];  
  
    // .setMaster("local[*]")  
    SparkConf sparkConf = new SparkConf();  
    sparkConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");  
    sparkConf.set("spark.kryo.registrator", "org.nd4j.Nd4jRegistrator");  
    sparkConf.setAppName("Skymind_MLP_BuildSaturnModel");
```

```

JavaSparkContext sc = new JavaSparkContext( sparkConf ); //new SparkConf());

String hdfsFilesToTrainModel = hdfsPathString; //"hdfs:///user/cloudera/svmlight/*";

String hdfsFilesToTestModel = args[ 1 ];

String cliLearningRate = args[ 2 ];

String cliEpochs = args[ 3 ];

learningRate = Double.parseDouble(cliLearningRate);

nEpochs = Integer.parseInt( cliEpochs );

int worker_count = Integer.parseInt( args[ 4 ].toString() );

int batchSizePerWorker = Integer.parseInt( args[ 5 ] );

final int parsedSVMLightFeatureColumns = 2;
String hdfsPathToSaveModel = "/user/skymind/models/mlp_saturn/dl4j_mlp.model";

JavaRDD<String> rawCSVRecords_trainingRecords = sc.textFile( hdfsFilesToTrainModel );

JavaRDD< LabeledPoint > trainingRecords = rawCSVRecords_trainingRecords.map(new Function< String, LabeledPoint > {
    @Override
    public LabeledPoint call(String rawRecordString) throws Exception {
        String[] parts = rawRecordString.split(",");
        String label = parts[ 0 ]; //SVMLightUtils.getLabel( svmLight );
        double dLabel = Double.parseDouble( label );
        Vector csvVector = Utils.convert_CSV_To_Dense_Vector( rawRecordString, parsedSVMLightFeatureColumns );
        return new LabeledPoint( dLabel, csvVector );
    }
}).cache();

JavaRDD<String> rawCSVRecords_testRecords = sc.textFile( hdfsFilesToTestModel );

JavaRDD< LabeledPoint > testRecords = rawCSVRecords_testRecords.map(new Function< String, LabeledPoint > {
    @Override
    public LabeledPoint call(String rawRecordString) throws Exception {
        String[] parts = rawRecordString.split(",");
        String label = parts[ 0 ]; //SVMLightUtils.getLabel( svmLight );
        double dLabel = Double.parseDouble( label );
        Vector csvVector = Utils.convert_CSV_To_Dense_Vector( rawRecordString, parsedSVMLightFeatureColumns );
        return new LabeledPoint( dLabel, csvVector );
    }
});

```

```

        }
}).cache();

// Build the model

StopWatch watch = new StopWatch();
long start = System.currentTimeMillis();

MultiLayerConfiguration dl4j_network_conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations)
.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("relu")
.build())
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
.weightInit(WeightInit.XAVIER)
.activation("sigmoid").weightInit(WeightInit.XAVIER)
.nIn(numHiddenNodes).nOut(numOutputs).build())
.pretrain(false).backprop(true).build());

MultiLayerNetwork networkModel = new MultiLayerNetwork( dl4j_network_conf );
networkModel.init();
networkModel.setUpdater( null );
networkModel.setListeners(Collections.singletonList((IterationListener) new ScoreIterationLister()));

TrainingMaster tm = new ParameterAveragingTrainingMaster( true, worker_count, 1, batchSizePerWorker );

SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer( sc, networkModel, tm );
sparkNetwork.setCollectTrainingStats(true);

long totalRecs = trainingRecords.count();

MultiLayerNetwork inProcessNetwork = null;

//fit on the training set
for ( int n = 0; n < nEpochs; n++ ) {

    start = System.currentTimeMillis();
    System.out.println( "\n\n> ----- Epoch" + n + " ! ----- \n\n" );

    inProcessNetwork = sparkNetwork.fitLabeledPoint( trainingRecords );

    long end = System.currentTimeMillis();
    long epochSeconds = Math.abs(end - start) / 1000; // se
    long epochMin = epochSeconds / 60; // se
}

```

```

        double recordsPerSecond = (double)batchSizePerWorker / (double)epochSeconds;

        System.out.println("----- Epoch " + n + " complete -----");

        System.out.println("\nTime for Epoch Training " + Math.abs(end - start) + " ms, (" + epoch

        System.out.println( "Avg Worker Records/Sec: " + recordsPerSecond );

    }

final MultiLayerNetwork scoreNetwork = inProcessNetwork.clone();

System.out.println( "\n\n> ----- Training Complete! ----- \n\n" )

long end = System.currentTimeMillis();
System.out.println("\n\nTime for training " + Math.abs(end - start) + "\n\n");
watch.reset();

System.out.println( "\n\n> ----- [ Calculating F1 Score ] -----"

// Compute raw scores on the test set.
JavaRDD<Tuple2<Object, Object>> predictionAndLabels = testRecords.map(
    new Function<LabeledPoint, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(LabeledPoint p) {

            INDArray ndArray = MLLibUtil.toVector( p.features() );
            INDArray pred_tmp = scoreNetwork.output( ndArray );

            Vector prediction = MLLibUtil.toVector( pred_tmp );
                //trainedNetworkWrapper.predict(p.features());
            double max = 0;
            double idx = 0;
            for(int i = 0; i < prediction.size(); i++) {
                if(prediction.apply(i) > max) {
                    idx = i;
                    max = prediction.apply(i);
                }
            }

            return new Tuple2<Object, Object>(idx, p.label());
        }
    );
);

// Get evaluation metrics.
MulticlassMetrics multi_class_metrics = new MulticlassMetrics( JavaRDD.toRDD( predictionAndLab

double precision = multi_class_metrics.fMeasure();
double recall = multi_class_metrics.recall();

```

```

BinaryClassificationMetrics metrics = new BinaryClassificationMetrics(JavaRDD.toRDD( predictions );
double auROC = metrics.areaUnderROC();

System.out.println( "\n> AUC: " + auROC + " " );
System.out.println( "\n> Precision: " + precision + " " );
System.out.println( "\n> Recall: " + recall + " " );

SparkTrainingStats stats = sparkNetwork.getSparkTrainingStats();
StatsUtils.exportStatsAsHtml(stats, "hdfs:/user/skymind/stats/SparkStats.html", sc);

sc.stop();

System.out.println( "Skymind Cert > MLP-Saturn: Job Complete" );

}

```

In the following sections we'll break out sections of the code to highlight key concepts.

## Building the Spark Job Jar

To build our spark job we 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.

## Setting Up Network Architecture

In the code above we can see our network architecture is nearly identical to the one we build for the single process version of this job back in Chapter 5. We highlight this code below.

```

MultiLayerConfiguration dl4j_network_conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations)
    .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("relu")
        .build())
    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)

```

```

        .weightInit(WeightInit.XAVIER)
        .activation("sigmoid").weightInit(WeightInit.XAVIER)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();
}

```

We covered the aspects of this network architecture back in Chapter 5.



### 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.

## Tracking Progress and Understanding Results

In this example we're actually using some of Spark's MLLib library's evaluation code to demonstrate interoperability between DL4J and MLLib. We highlight this section of the code below.

```

// Compute raw scores on the test set.
JavaRDD<Tuple2<Object, Object>> predictionAndLabels = testRecords.map(
    new Function<LabeledPoint, Tuple2<Object, Object>>() {
        public Tuple2<Object, Object> call(LabeledPoint p) {

            //org.apache.spark.mllib.linalg.Vector in = MLLibUtil.toVector(scoreNetwork.
            //Vector prediction = scoreNetwork.predict( MLLibUtil.toVector( p.features() );
            //scoreNetwork.output(MLLibUtil.toMatrix( p.features() ));

            INDArray ndArray = MLLibUtil.toVector( p.features() );

            INDArray pred_tmp = scoreNetwork.output( ndArray );

            Vector prediction = MLLibUtil.toVector( pred_tmp );
            //trainedNetworkWrapper.predict(p.features());
            double max = 0;
            double idx = 0;
            for(int i = 0; i < prediction.size(); i++) {
                if(prediction.apply(i) > max) {
                    idx = i;
                    max = prediction.apply(i);
                }
            }

            return new Tuple2<Object, Object>(idx, p.label());
        }
    );
}

```

```

// Get evaluation metrics.
MulticlassMetrics multi_class_metrics = new MulticlassMetrics( JavaRDD.toRDD( predictionA

double precision = multi_class_metrics.fMeasure();
double recall = multi_class_metrics.recall();

BinaryClassificationMetrics metrics = new BinaryClassificationMetrics(JavaRDD.toRDD( predi
double auROC = metrics.areaUnderROC();

System.out.println( "\n> AUC: " + auROC + " " );
System.out.println( "\n> Precision: " + precision + " " );
System.out.println( "\n> Recall: " + recall + " " );

```

DL4J includes a class called “MLLibUtil” to convert from an NDArray vector to a MLLib vector. This is a great example showing how we would do this and how MLLib’s evaluation metrics can be generated with NDArray vectors output from our model.

## Recurrent Neural Network Spark Example

*[ to be completed for final production ]*

### Loading and Vectorizing Data

Section content goes here

### Setting Up Network Architecture

Section content goes here

### Tracking Progress and Understanding Results

Section content goes here

## Modeling MNIST with a Convolutional Neural Network on Spark Local

For this example we’ll take the Convolutional Neural Network example from chapter 5 and update it to run on Spark but with the local filesystem.

### Java Code Listing for Spark Local LeNet Example

In the example below we can see the code for the building a Convolutional Neural Network model with Spark.

### *Example 9-2. Spark MNIST Example*

```
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.Builder builder = new NeuralNetConfiguration.Builder()  
            .seed(seed)  
            .iterations(iterations)  
            .regularization(true).l2(0.0005)  
            .learningRate(0.1)  
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)  
            .updater(Updater.ADAGRAD)  
            .list()  
            .layer(0, new ConvolutionLayer.Builder(5, 5)  
                .nIn(nChannels)  
                .stride(1, 1)  
                .nOut(20)  
                .weightInit(WeightInit.XAVIER)  
                .activation("relu")  
                .build())  
            .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)  
                .kernelSize(2, 2)  
                .build())  
    }  
}
```

```

    .layer(2, new ConvolutionLayer.Builder(5, 5)
        .nIn(20)
        .nOut(50)
        .stride(2,2)
        .weightInit(WeightInit.XAVIER)
        .activation("relu")
        .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .build())
    .layer(4, new DenseLayer.Builder().activation("relu")
        .weightInit(WeightInit.XAVIER)
        .nOut(200).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .weightInit(WeightInit.XAVIER)
        .activation("softmax")
        .build())
    .backprop(true).pretrain(false);
new ConvolutionLayerSetup(builder,28,28,1);

MultiLayerConfiguration conf = builder.build();
MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();

//Create Spark multi layer network from configuration
ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster.Builder(examplesPer
    .workerPrefetchNumBatches(0)
    .saveUpdater(true)
    .averagingFrequency(5)                                //Do 5 minibatch fit operations per
    .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 in Code

To use the Spark execution framework in Java we need to create a `SparkConf` object and a `JavaSparkContext` object instance. Below we see the portion of the code example above where we set these objects up.

```
//Create spark context, and load data into memory
SparkConf sparkConf = new SparkConf();
sparkConf.setMaster("local[*]");
sparkConf.setAppName("MNIST");
JavaSparkContext sc = new JavaSparkContext(sparkConf);
```

We're also telling the system that we want to run Spark in local mode as opposed to on a Hadoop cluster. We set the name of the job as "MNIST" and then pass our Spark configuration to our `JavaSparkContext` object constructor.

## Loading and Vectorizing MNIST Data

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 `JavaRDD`s.

## Setting Up the Convolutional Network Architecture

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.Builder builder = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(iterations)
    .regularization(true).l2(0.0005)
    .learningRate(0.1)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(Updater.ADAGRAD)
    .list()
    .layer(0, new ConvolutionLayer.Builder(5, 5)
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .weightInit(WeightInit.XAVIER)
        .activation("relu")
        .build())
    .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .build())
    .layer(2, new ConvolutionLayer.Builder(5, 5)
        .nIn(20)
        .nOut(50)
        .stride(2,2)
        .weightInit(WeightInit.XAVIER)
        .activation("relu")
        .build())
    .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .build())
    .layer(4, new DenseLayer.Builder().activation("relu")
        .weightInit(WeightInit.XAVIER)
        .nOut(200).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .weightInit(WeightInit.XAVIER)
        .activation("softmax")
        .build())
    .backprop(true).pretrain(false);
new ConvolutionLayerSetup(builder,28,28,1);

```

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 the Convolutional Network on Spark

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(example
    .workerPrefetchNumBatches(0)
    .saveUpdater(true)
    .averagingFrequency(5)                                //Do 5 minibatch fit operations
    .batchSizePerWorker(examplesPerDataSetObject)        //Number of examples that each w
    .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:

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.

Below we detail why the code uses these special class for Spark training.

### Using the SparkDl4jMultiLayer Class

The SparkDl4jMultiLayer class is the way we can control executors in nearly the same fashion as we performed model training on a local machine.

```

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);
}

```

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.

## Building the Spark Job Jar

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.

## Executing the Spark Job

Once we've copied the job jar to our intended machine to execute on Spark, we type the command:

```
spark-submit --class io.skymind.[todo] --num-executors 3 --properties-file ./spark_extra.props ./[
```

This will output a lot of training information to the console as the model reports back how the training is progressing.

## 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<sup>1</sup> 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

---

<sup>1</sup> 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 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 or 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". 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:

Like the Internet of Things, Web 2.0, and big data, AI is discussed and debated in many different contexts by people with all sorts of motives and backgrounds: academics, business types, journalists, and technologists. As with these other nebulous technologies, it's no wonder the meaning of AI can be hard to pin down; everyone sees what they want to see.

Part of the problem with viewpoints and defining intelligence is that we weave liberally into defining consciousness and then by extension more philosophical (e.g. “what is consciousness”) and religious topics (e.g. “what is a soul?”). We’re touching on some complex territory at this point and any 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 recently spoke at a conference on 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<sup>2</sup>:

“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<sup>3</sup>:

Part of the problem is that some companies & journalists are hyping it up, blurring the line between sci-fi and reality. Because it sells.

---

<sup>2</sup> <https://twitter.com/fchollet/status/734919468908875776>

<sup>3</sup> <https://twitter.com/fchollet/status/734919888876097536>

Chollet goes on to say that we should “define” what we’re talking<sup>4</sup>:

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.

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 today.

**Moving the Goal Posts.** Psychologists habitually have had disdain for the metaphor of the human brain as a computer. In the 2016 article<sup>5</sup> 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”.

---

<sup>4</sup> <https://twitter.com/fchollet/status/734920203478274048>

<sup>5</sup> <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<sup>6</sup> 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.

---

<sup>6</sup> <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 unfulfilled, however, and the Trough of disillusionment looms.

Winter kills the weak.

A few interesting applications slowly emerge from the trough, are rebranded (“voice recognition”) and are integrated into other projects as features, generally under the “latent intelligence” class of improvements. We’ve seen this with:

- 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

### The Big Jump in Computer Vision

In 2006 Geoff Hinton and team at the University of Toronto published a key paper on Deep Belief Networks<sup>7</sup>. 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.

### Advancement in Applications of Deep Learning

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.

### The Wave of Big Data

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 bringing online large distributed systems in the early 2010's to hold their growing transactional datasets. These enterprise

---

<sup>7</sup> <https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>

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.

## A Confluence of Coverage

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.

## Tidal Forces

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.

## A Repeating Season

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.



### Jeopardy

Jeopardy does seem to be a “solved problem”, however.

Marketing departments are setting up similar scenarios as in the previous two Artificial Intelligence winters, unfortunately.

The coals of the real advances in the domain, as in previous winters, will keep the true enthusiasts and hardcore researchers warm during the cold.

## APPENDIX B

# Numbers Everyone Should Know

Also known as “Jeff Dean’s 12 Numbers”.

*Table B-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.



# Setting Up DL4J Projects in Maven

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.

## ND4J

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 D

# 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 averaging-Frequency(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 E

# Using the ND4J API

ND4J is a scientific computing library for the JVM. It is designed to run fast in production environments. Its 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. Activation functions are the exception, because they take strings such as "relu" or "tanh".

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 they take a single argument and perform an operation on it. Absolute value is a transform that takes the argument `x` like `soabs(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]  
]
```

[ TODO: re-check the column / row ordering! ]



## 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](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 it is possible to use a specific ordering for a given array, should the need arise.

ND4J defaults to [todo]

**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:

- Row vector: Nd4j.create(float[]) or Nd4j.create(double[])
- Column vector: Nd4j.create(float[],new int[]{length,1}) or Nd4j.create(double[],new int[]{length,1})

For 2d arrays, use Nd4j.create(float[][])) or Nd4j.create(double[][])).

For creating NDArrays from Java primitive arrays with 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 *i*th dimension: `size(i)`
- Get the size of all dimensions, as an `int[]`: `shape()`
- Determine the total number of elements in array: `arr.length()`
- See also: `isMatrix()`, `isVector()`, `isRowVector()`, `isColumnVector()`

## DataSet

The `org.nd4j.linalg.dataset.DataSet` class represents a data transform with input and output data. We can see its full java doc listing at:

<http://nd4j.org/doc/org/nd4j/linalg/dataset/DataSet.html>

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 labels = 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.



### 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”).

[ *is this value scaled from 0.0 to 1.0 or the actual value?* ]

## 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 );  
  
public int[] predict(INDArray d)
```

Section content goes here

a



### 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.

## Evaluating Regular Predictions

Section content goes here

## Evaluating Timeseries Predictions

Section content goes here

## Building Confusion Matrices

Section content goes here

## 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 set of abstractions for adding in new data formats or data sources.



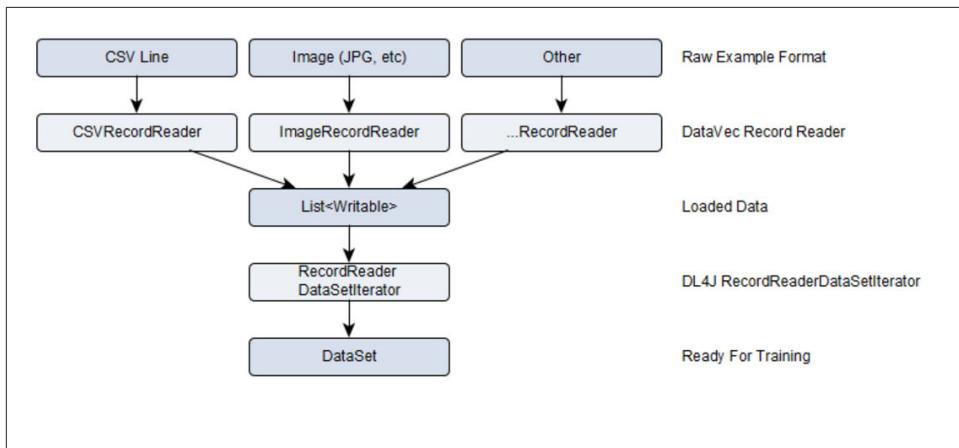
## ETL, Preprocessing and Vectorization

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.

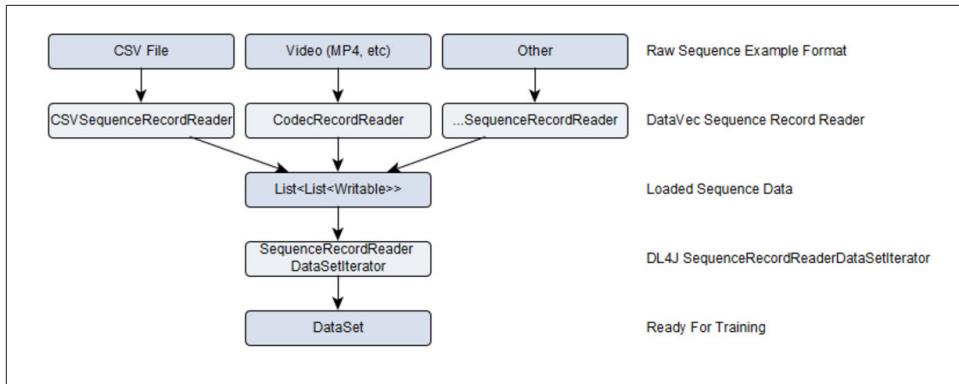
Writable is an interface that represents a piece of data. For example, a DoubleWritable can be used to represent double-precision numerical values, and Text instances are used for text data \footnote{Additional types include IntWritable, LongWritable, FloatWritable and NullWritable}. DataVec also provides 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 as a list of (or sequence of) time steps. That is, the outer list is a list of time steps, and the inner list is a list of values at each time step. Put another way, the code `mySequence.get(i).get(j)` would return the *j*th value at the *i*th time step. 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, numClasses );  
  
//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 flexibility 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, labelMa
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, numClas

//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 CropImageTransform() );

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);

DataSetIterator iterator = new SequenceRecordReaderDataSetIterator( reader, minibatchSize, labelIndex);

//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 Wrangling 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 date time column, and extract out the hour of the day as a new feature column

These operations can all be specified 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 pre-processing 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

# RL4J and Reinforcement Learning

Ruben Fiszel<sup>1</sup>

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.

## Preliminaries

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  $S$  (eg: in chess, a state is the board configuration)
- A set of possible actions  $A$  (In chess, all the moves that could be possible in every configuration possible, eg: e4-e5)
- The conditional distribution  $P(s'|s,a)P(s'|s,a)$  of next state given a current state and an action. (In a deterministic environment like chess, there is only one state

---

<sup>1</sup> <http://rubenfiszel.github.io/>

$s'$  with probability 1, and all the other with probability 0. Nevertheless, in a stochastic (involving randomness, eg: a coin toss) environment, the distribution is not as simple.)

- The reward function of transitionning from state  $s$  to  $s'$ :  $R(s,s')P(s,s')$  (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:  $\gamma$ . This is the preference for present rewards compared to future rewards. (A concept very common in **finance**.)

Note: It is usually more convenient to use the set of Action  $A_{s,A}$ , which is the set of available move from a given state, than the complete set  $A$ .  $A_{s,A}$  is simply the elements  $\in A \in A$  such that  $P(s'|s,a) > 0$ .

[ diagram: schema of a MDP ]

Final states: The states that have no available actions are final states.

Episode: An episode is a complete play from one of the initial state to a final state.

$s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_n$

Cumulative reward: The cumulative reward is the discounted sum of reward accumulated throughout an episode:

$$R = \sum_{t=0}^n \gamma^t r_t$$

Policy: A Policy is the agent's strategy to choose an action at each state. It is noted by  $\pi$ .

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

*“He who can do the greater things, can do the lesser things”*

### Model-Free

The conditional distribution and the reward function are the model of the environment. Some reinforcement learning algorithms can work without being given the model. Thus, in order to learn the best strategy, they also 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. Model-free reinforcement learning 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 be given access to a partial observation of the state only. This is the difference between the partial and fully observed setting. For instance, our field of vision is a very partial observation of the full state of the universe (the position and energy of every particle in the universe). Like model-free, partial observation setting is a superset of fully observed setting.

Fortunately for us, DQN is a model-free reinforcement learning that can learn from the partially observed setting.

## 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 informations that happen during a transition. An action is all the available command at the disposal of the player (In doom, go up, right, left, shoot).

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** 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, where if given a board configuration, a move would be good against a chess master, it would still be a good move against a beginner [^1]. Whatever is the current level of the agent, by playing against himself, he still get information about the quality of his previous moves (good moves if he won, bad moves if he lost on average).

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 magic 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<sup>2</sup>. The policy was bootstrapped on a dataset of master moves, then they used reinforcement learning and self play to increase furthermore the level of their agent (and in the end gets better than the original dataset). They used their policy gradient in combination with a Monte-Carlo Search Tree on a huge amount of computation power.

---

<sup>2</sup> <https://deepmind.com/alpha-go>

This setting is a bit different than learning from pixels. Firstly, because the input is not as high-dimensional. The **manifold** is a lot closer to its embedding space. Nevertheless, a convolutional layer was still used in this case to 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 everything, in principle, everything ironed and the true problems concealed. (Erwin Schrödinger)

### From Policy to Neural Network

Our goal is learn the optimal policy  $\pi^*$  that

$$E[R_0] = E\left[\sum_{t=0}^n \gamma^t r_{t+1}\right]$$

Let's introduce an auxiliary function:

$$V_\pi(s) = E\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_n \mid s_t = s, \text{policy followed at each state is } \pi\}$$

which is the expected cumulative reward from a state  $s$  following the policy  $\pi$ .

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 action, choose the action that the expectation of  $V\pi^*(s)$ . This is a greedy behavior. The optimal policy is the greedy policy w.r.t to  $V\pi^*$ .

$\pi^*(s)$  chooses a s.t  $a = \arg \max_a [E_\pi(r_t + \gamma V(s_{t+1}) \mid s_t = s, a_t = a)]$

If you were extremely 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.

To solve this very annoying issue, we are gonna use another auxiliary function, the Q function:

$$Q_{\pi^*}(s, a) = E_{\pi}[r_t + \gamma V_{\pi^*}(s_{t+1}) \mid s_t, a_t = a]$$

In a greedy setting, we have the 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  $\pi^*$ .

$$\pi^*(s) \text{ chooses a s.t } a = \max_a [Q_{\pi^*}(s, a)]$$

No more uncomputable expectations, Neat

Nevertheless, we have only moved the expectation from outside to inside the oracle. And unfortunately, oracles do not exist in the real world.

The trick here is that we have reduced an abstract notion that is a policy into a numerical function. Fortunately for us, there is one weapon at our disposal to approximate complex functions: Neural networks.

Neural networks are universal function approximator. They can approximate any differentiable function. Although they can get stuck in local extrema and many proofs are not valid anymore when throwing neural networks in the equation. This is because their learning is not as deterministic or boundable than a linear function. Nonetheless, in most case, with the right hyperparameters, they are unreasonably powerful. 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 do not exist in the real world, so we will have to 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](#)).

If we play randomly from a given state, the better state should get better reward on average. So without knowing anything about the environment, you can get some information about the expected value of a state. For example, at poker, better hands will win more often on average than lesser hands when every decision is taken randomly. Monte Carlo Search Tree are based on this property. This is a phase of exploration that lead to unsupervised learning and enable us to extract meaningful label.

More formally,

Given a policy  $\pi$ , a state  $s$  and an action  $a$ , to get an approximation of  $Q(s,a)$  we sample it according to its definition:

$$\begin{aligned} Q_\pi(s, a) &= E[r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_n | s_t = s, a_t = a] \end{aligned}$$

In plain english, we can get a label for  $Q\pi(s,a)$  by playing a sufficient number of time from  $s$  according to the policy  $\pi$ .

From an aggregate of signals:

[ diagram: "one signal" ]

We use the Mean-Square Error loss function with a learning rate of  $\alpha$  and apply Stochastic Gradient Descent (On a batch of size 1):

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)]$$

$(s_t, a_t)$  is the input  $Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)]$  is the label aka 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  $\pi$

$$Q_\pi(s_t, a_t) \leftarrow E_\pi[R_t] = E_{s_t, a_t, \dots, s_n \sim \pi} \left[ \sum_{i=t}^n \gamma^{i-t} r_i \right]$$

We can converge to the rightful expectation

[ diagram: "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 G-1. Hello World in Python*

```
val neuralNet: NeuralNet

def UnsupervisedLearningpoch() = {
    val state = randomState
    val action = randomAction(state)
    var (new_state, accuReward) = transition(state, action)
    accuReward += playRandomly(state)
    fit((state, action), accuReward)
}

def transition(state: State, action: Action): (State, Double) =
    // MDP specific
    (new_state, reward)
```

```

def randomState: State

def playRandomly(state: State): Double = {
    var s = state
    var accuReward = 0
    var k = 0
    while (!s.isFinal) {
        val action = randomAction(s)
        val (state, reward) = transition(s, action)
        k += 1
        accuReward += Math.pow(gamma, k) * reward
        s = state
    }
    accuReward
}

def randomAction(state: State): Action =
    oneOf(state.available_action)

def oneOf(seq: Seq[Action]): Action =
    seq.get(Random.nextInt(seq.size))

def fit(input: (State, Action), label: Double) =
    neuralNet.fit(toTensor(input), toTensor(label))

def toTensor(obj: Object): Tensor =
    //magic
    return tensor

```

There is 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 ... after an almost infinite time: you will have to visit every possible trajectories (a trajectory is the state visited and action chosen during an episode) at least once. 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 that we are gonna use is  $\epsilon$ -greedy exploration.  $\epsilon$ -greedy exploration is fairly simple. It is a policy that choose an action at random with odd ( $1 - \epsilon$ ) the best action as deemed by our current policy. Usually  $\epsilon$  is annealed over time to privilege exploitation over 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 path. 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 takes 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.

[ diagram: “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). On a sidenote, this is possible in some case, see [Giraffe](#) that uses TD-Lambda.

Fortunately, some rearranging and optimisations are possible:

## Bellman Equation

We can transform the Q equation into a Bellman equation:

$$\begin{aligned} Q_{\pi}(s, a) &= E[r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_n | s_t = s, a_t = a] \\ &= E[r_t + \gamma r_{t+1} + V(s_{t+1}) | s_t = s, a_t = a] \\ &= E[r_t + \gamma \max_a Q(s_{t+1}, a) | s_t = s, a_t = a] \end{aligned}$$

As in the Monte-Carlo method, we can do many updates of Q.

MSE:

$$Q_{\pi}(s_t, a_t) \leftarrow Q_{\pi}(s_t, a_t) + \alpha \left[ \underbrace{r_t + \max_a Q_{\pi}(s_{t+1}, a)}_{\text{target}} - Q_{\pi}(s_t, a_t) \right]_{\text{TD-error}}$$

TD-error is the “Temporal difference error”. Indeed we are actually calculating the difference between what the Q approximation expects in the future plus the realized reward and its present value as evaluated by the neural net.

That bellman equation only make sense with some boundary conditions. if s is terminal:

$$V(s) = 0$$

and

$$Q(s_{t-1}, a) = r_t$$

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 this excellent post from [kofzor](#).

## 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:

```

def epoch() = {

    //sample among the initial state space
    //(often unique state)
    var state = initState

    //while the state is not terminal,
    //play an episode and do a Q-update at each transition
    while(!state.isTerminal) {

        //sample action from eps-greedy policy
        val action = epsilonGreedyAction(state)

        //interaction with the environment
        val (nextState, reward) = transition(state, action)

        //Q-update
        update(state, action, reward, nextState)

        state = nextState
    }
}

//Our Q-update as explained above
def update(state: State, action: Action, reward: Double, nextState: State) = {
    val target = reward + maxQ(nextState)
    fit((state, action), target)
}

//the eps-greedy policy implementation
def epsilonGreedyAction(state: State) = {
    if (Random.float() < epsilon)
        randomAction(state)
    else
        maxQAction(state)
}

//Retrive max Q value
def maxQ(state: State) =
    actionsWithQ(state).maxBy(_._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  $a\alpha$  as an additional input of the neural net combined with the state, the state is the only input and the output contain the Q value of every output possible. This make sense only when the available actions are consistent across the full episode.

[ diagram here: Q modeling ]

## 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 optimize your learning based on the last episode.

The Google DeepMind research team used experience replay, which is a windowed buffer of the last N transitions (N being a million in the original paper) with DQN and greatly improved their 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-greedy policy  
        val action = epsilonGreedyAction(state)  
  
        //interaction with the environment  
        val (nextState, reward) = transition(state, action)  
  
        //store transition (Exp Replay is just a Ring buffer)  
        expReplay.store(state, action, reward, nextState)  
  
        //Q update in batch
```

```
updateFromBatch(expReplay.getBatch())

state = nextState
}
}
```

## Compression

nd4j, the tensor library of dl4j, does not support as first class type uint8. However, pixels in grayscaling are encoded with that precision. To avoid wasting too much space on memory, INDArray were compressed to uint8.

## Convolutional Layers and Image Preprocessing

Convolutional layers are layers that are excellent 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 so it would seem reasonable to accept that preprocessing.

## Resizing

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 paper, only 1 in 4 frame 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.

## Double dqn

The network is frozen every M 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 instability.

## Clipping

The TD-error is clipped such that no outlier update can have too much impact on the learning.

## Policy Gradient

Policy gradient work by directly learning the stochastic policy from the softmax distribution and its log gradient. To learn more about it, see this excellent post: [Karpathy blog](#)

## Asynchronous Methods for Deep Reinforcement Learning

A3C and Asynd NStep Q learning are a WIP in rl4j. It bypass the need for experience replay by using multiple agents exploring in parallel the environment. The original [paper](#) uses Hogwild!. In RL4J, a workaround is to use a central thread and accumulate gradient from slave agents.

## Example in RL4J

Here is a working example with RL4J.

*Example G-2. Hello World in Python*

```
public static QLearning.QLConfiguration DOOM_QL =
    new QLearning.QLConfiguration(
        123, //seed
        10000, //maxEpochStep
        8000000, //maxStep
        1000000, //expRepMaxSize
        32, //batchSize
        10000, //targetDqnUpdateFreq
        50000, //updateStart
        0.99, //gamma
        100.0, //errorClamp
        0.1f, //minEpsilon
        1f / 1000000f, //epsilonDecreaseRate
        true //doubleDQN
    );

public static DQNFactoryStdConv.Configuration DOOM_NET =
    new DQNFactoryStdConv.Configuration(0.00025, 0.000, 0.99);

public static HistoryProcessor.Configuration DOOM_HP =
    new HistoryProcessor.Configuration(4, 84, 84, 84, 84, 0, 0, 4);
```

```
public static void main(String[] args) {  
    DataManager manager = new DataManager(true);  
    VizDoom mdp = new DeadlyCorridor(true);  
  
    QLearningDiscreteConv<VizDoom.GameScreen> dql = new QLearningDiscreteConv(mdp, DOOM_NET, DOOM_  
    dql.train();  
  
    dql.getPolicy().save("end.model");  
    mdp.close();  
}
```

---

# Other Deep Learning Libraries

Deeplearning4j is not the first open-source deep-learning project, but it is distinguished from its predecessors in both programming language and intent. DL4J is a JVM-based, industry-focused, commercially supported, distributed deep-learning framework intended to solve problems involving massive amounts of data in a reasonable amount of time. It integrates with Hadoop and Spark using an arbitrary number of GPUs or CPUs. There is also enterprise-support available for the project from vendors.



### Deeplearning4j and Goals

Deeplearning4j intends to be the equivalent of Scikit-learn in the Deep Learning space. It aims to automate as many knobs as possible in a scalable fashion on parallel GPUs or CPUs, integrating as needed with Hadoop and Spark.

Let's take a look at the other Deep Learning projects in the space.

## TensorFlow

TensorFlow is the open source python Deep Learning library from Google. It is written with a Python API over a C++ backend engine for speed. Like Theano (described in another section below), TensorFlow generates a computational graph (e.g. a series of matrix operations such as  $z = \text{sigmoid}(x)$  where  $x$  and  $z$  are matrices) and performs automatic differentiation.



## Automatic Differentiation

Automatic differentiation is important because we don't want to have to hand-code a new variation of backpropagation every time we're experimenting with a new arrangement of neural networks.

In Google's ecosystem, the computational graph is then used by Google Brain for the heavy lifting, but Google hasn't open-sourced those tools yet. TensorFlow is one half of Google's in-house Deep Learning solution. TensorFlow is commercially supported in a SaaS setup through Google Cloud.

## Downpour and Sandblaster

There are other internal Google Deep Learning systems besides TensorFlow including Downpour SGD and Sandblaster L-BFGS.

## Caffe

Caffe is a well-known and widely used machine-vision library that ported Matlab's implementation of fast convolutional neural networks to C and C++. Caffe is not intended for other Deep Learning applications such as text, sound or timeseries data. Like other frameworks mentioned here, Caffe has chosen Python for its API.

Both Deeplearning4j and Caffe perform image classification with convolutional neural networks. In contrast to Caffe, Deeplearning4j offers parallel GPU support for an arbitrary number of chips as well as many features that make Deep Learning run more smoothly on multiple GPU clusters in parallel. Caffe has a suite of pre-trained models hosted on its Model Zoo site.



## The Caffe Model Zoo

The Caffe Model Zoo has many Convolutional Neural Network pre-trained models available for use. Check them out at:

[http://caffe.berkeleyvision.org/model\\_zoo.html](http://caffe.berkeleyvision.org/model_zoo.html)

## Keras

Keras is a Deep Learning library in python by Francois Chollet for TensorFlow and Theano. Keras is minimalist and highly modular with a focus on enabling fast experimentation through a layer-oriented architecture. Major features include:

- fast prototyping
- support for Convolutional and Recurrent Neural Networks (and combinations)

- flexible connection schemes
- GPU and CPU support



### The Keras Homepage

The reader can find more information about the Keras project at:  
<http://keras.io/>

## Torch

Torch is a computational framework written in Lua that supports machine learning algorithms. Some version of Torch is used by large tech companies such as Google DeepMind and Facebook, which devote in-house teams to customizing their Deep Learning platforms.

### What is Lua?

Lua is a multi-paradigm scripting language that was developed in Brazil in the early 1990s.

## Theano

Most academic researchers in the field of Deep Learning rely on Theano. Theano is one of the earliest Deep Learning frameworks and is written in Python. Theano is well suited to data exploration and intended for research.

Numerous open source Deep Learning libraries have been built on top of Theano, including Keras, Lasagne and Blocks. These libraries add an abstraction layer API on top of Theano's API to simplify the programmer's job when building and training neural networks.

## Other Machine Learning Libraries

The Deep Learning frameworks listed above are more specialized than general machine-learning frameworks. Other general purpose machine learning libraries of note are listed below:

- R
- SAS
- sci-kit learn - the default open-source machine-learning framework for Python.
- Apache Mahout - The first machine-learning framework for Hadoop



## APPENDIX I

# Evaluating Deep Learning Platforms

Below we look at a few important facets in how the practitioner may compare and evaluate Deep Learning libraries.

## Licensing

Licensing is an important distinction among open-source projects: Theano, Torch and Caffe employ a BSD License, which does not address patents or patent disputes. Deeplearning4j and ND4J are distributed under an Apache 2.0 License



### Litigation and Software Licenses

The Apache 2.0 License contains both a patent grant and a litigation retaliation clause. That is, anyone is free to make and patent derivative works based on Apache 2.0 licensed code, but if they sue someone else over patent claims regarding the original code (DL4J in this case), they immediately lose all patent claim to it.

In other words, you are given resources to defend yourself in litigation, and discouraged from attacking others. The BSD license doesn't typically address this issue.

## Speed

Deeplearning4j has been optimized to run on various chips including x86 and GPUs with CUDA C. Torch7 and DL4J both employ parallelism but there are differences in how parallelism is used. DL4J uses the parameter averaging strategy for parallelizing computation on neural networks. The parallelism is largely transparent to the practitioner as DL4J has a unified engine for multiple single or parallel execution contexts.



## ND4J Benchmarks

Deeplearning4j's underlying linear algebra computations, performed with ND4J, have been shown to run at least twice as fast as Numpy on very large matrix multiplies as shown on the page below:

<http://nd4j.org/benchmarking>

# Arguments for the JVM

We're often asked why we chose to implement an open-source Deep Learning project in Java, when so much of the Deep Learning community is focused on Python. After all, Python has great syntactic elements that allow you to add matrices together without creating explicit classes, as Java requires you to do. Likewise, Python has an extensive scientific computing environment with native extensions like Theano and Numpy. Below we list some of the competitive aspects that make the JVM a great place to practice Deep Learning.

## Widely Deployed

Many major companies use Java or a JVM-based system. It remains the most widely used language in enterprise. The JVM is the language of Hadoop, Hive, Lucene and Pig, which happen to be useful for machine learning problems. Many programmers solving real-world problems could benefit from deep learning, but they are separated from it by a language barrier. We want to make Deep Learning more usable to a large new audience that can put it to immediate use.

## The JVM Ecosystem

Java's popularity is only strengthened by its ecosystem. Hadoop is implemented in Java and Spark is implemented in Scala (JVM language). Spark can also execute within Hadoop's Yarn API. Java boasts a highly tested infrastructure for pretty much any application and Deep Learning networks written in Java can execute in multiple environments with DL4J (e.g. locally or in parallel on Spark).

Java can also be used natively from other popular languages like Scala, Clojure, Python and Ruby. By choosing Java, we excluded the fewest major programming communities possible. While Java is not as fast as C or C++, it is much faster than

many believe, and DL4J can execute in a distributed fashion that can accelerate training. That is, if the reader wants speed, they just throw more boxes at it.

## Scala

Special attention was paid to Scala in building Deeplearning4j and ND4J, because it is believed Scala has the potential to become an important language in data science. Writing numerical computing, vectorization and Deep Learning libraries for the JVM with a Scala API moves the community toward that goal.

## Java and Speed

Java is inherently faster than Python. Anything written in Python by itself, disregarding its reliance on Cython, will be slower. Admittedly, most computationally expensive operations are written in C or C++. Most Deep Learning projects that are initially written in Python will have to be rewritten if they want to be more competitive in terms of execution speed. Deeplearning4j relies on JavaCPP to call pre-compiled native C++ from Java, substantially accelerating the speed of training.

## Solvable Issues

ND4J solves the issue of Java not having a robust scientific computing library. ND4J runs on distributed CPUs or GPUs, and can be interfaced via a Java or Scala API.

## Security as a First Class Citizen

Java is a secure, network language that inherently works cross-platform on Linux servers, Windows and OSX desktops, Android phones and in the low-memory sensors of the Internet of Things via embedded Java.



## APPENDIX J

# 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/Canova  
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, Canova and Deep-learning4j 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, Canova and Deeplearning4j, in that order, is a good way to get the most recent bug fixes and features.

---

# 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 program 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 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 uber-jar 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 OutofMemory error that looks like this, fork join is the problem:

```
java.util.concurrent.ExecutionException: java.lang.OutOfMemoryError ...java.util.concurrent.ForkJoin
```

### 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 than 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** 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 libopenblas.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/>





## APPENDIX L

---

# References

Appendix content

## Reference Papers

- [1] LeCun et al. 1998, Efficient BackProp, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [2] Orr and Muller, 1998, Regularization Techniques to Improve Generalization, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [3] Prechelt 1998: Early Stopping – But When?, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [4] Rognvaldsson 1998, A Simple Trick for Estimating the Weight Decay Parameter, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [5] Larsen et al. 1998, Adaptive Regularization in Neural Network Modeling, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [6] Horn et al. 1998, Large Ensemble Averaging, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [7] Flake 1998, Square Unit Augmented, Radially Extended, Multilayer Perceptrons, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [8] Caruana 1998, A Dozen Tricks with Multitask Learning, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [9] Lawrence et al. 1998, Neural Network Classification and Prior Class Probabilities, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [10] Moody 1998, Forecasting the Economy with Neural Nets: A Survey of Challenges and Solutions, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [11] Neuneier and Zimmermann 1998, How to Train Neural Networks, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [12] Bottou 2012, Stochastic Gradient Descent Tricks, in Muller et al. 2012: Neural

Networks: Tricks of the Trade 2Ed.

- [13] Bengio 2012, Practical Recommendations for Gradient-Based Training of Deep Architectures, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [14] Hinton 2012, A Practical Guide to Training Restricted Boltzmann Machines, in Muller et al. 2012: Neural Networks: Tricks of the Trade 2Ed.
- [15] Bengio et al. 2015, Deep Learning (In Preparation), <http://www.iro.umontreal.ca/~bengioy/dlbook/>
- [16] Bengio 2007, Learning Deep Architectures for AI, to appear in Foundations and Trends in Machine Learning, <http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf>
- [17] Hinton et al. 2012, Improving neural networks by preventing co-adaption of feature detectors, <http://arxiv.org/pdf/1207.0580v1.pdf>
- [18] Srivastava et al. 2014: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>
- [19] Goodfellow et al. 2013, Maxout Networks, <http://arxiv.org/pdf/1302.4389.pdf>
- [20] Schaul, Zhang, LeCun 2013, No More Pesky Learning Rates, <http://jmlr.org/proceedings/papers/v28/schaul13.pdf>
- [21] Vincent et al. 2010, Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion  
<http://jmlr.csail.mit.edu/papers/volume11/vincent10a/vincent10a.pdf>
- [22] Goodfellow et al. 2015, Explaining and Harnessing Adversarial Examples, <http://arxiv.org/pdf/1412.6572v3.pdf>
- [23] Goodfellow et al. 2014, Generative Adversarial Nets, <http://arxiv.org/pdf/1406.2661.pdf>
- [24] Mirza, Osindero 2014, Conditional Generative Adversarial Nets, <http://arxiv.org/pdf/1411.1784.pdf>
- [25] Alain, et al. 2015, GSNs: Generative Stochastic Networks, <http://arxiv.org/pdf/1503.05571.pdf>
- [26] Andrej Karpathy 2015, The Unreasonable Effectiveness of Recurrent Neural Networks, <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- [27] Graves 2012, Supervised Sequence Labelling with Recurrent Neural Networks (PhD Thesis), <http://www.cs.toronto.edu/~graves/phd.pdf>
- [28] Sutskever 2013, Training Recurrent Neural Networks (PhD Thesis), [http://www.cs.utoronto.ca/~ilya/pubs/ilya\\_sutskever\\_phd\\_thesis.pdf](http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf)
- [29] Zaremba, Sutskever, Vinyals 2014, Recurrent Neural Network Regularization, <http://arxiv.org/pdf/1409.2329v4.pdf>
- [30] Martens 2010, Deep learning via Hessian-free optimization, [http://machine-learning.wustl.edu/mlpapers/paper\\_files/icml2010\\_Martens10.pdf](http://machine-learning.wustl.edu/mlpapers/paper_files/icml2010_Martens10.pdf)
- [31] Le et al. 2010, On Optimization Methods for Deep Learning, [http://machine-learning.wustl.edu/mlpapers/paper\\_files/ICML2011Le\\_210.pdf](http://machine-learning.wustl.edu/mlpapers/paper_files/ICML2011Le_210.pdf)
- [32] Graves, Wayne, Danihelka 2014, Neural Turing Machines, <http://arxiv.org/abs/1410.5401>
- [33] Zaremba, Sutskever 2015, Reinforcement Learning Neural Turing Machines,

- <http://arxiv.org/pdf/1505.00521v1.pdf>
- [34] Pascanu, Mikolov, Bengio 2013, On the difficulty of training Recurrent Neural Networks, <http://arxiv.org/pdf/1211.5063.pdf>
- [35] Mikolov 2012, Statistical Language Models based on Neural Networks (PhD Thesis), <http://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf>
- [36] Zaremba, Sutskever 2015, Learning to Execute, <http://arxiv.org/pdf/1410.4615v2.pdf>
- [37] 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](http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf)
- [38] Bengio et al. 2009, Curriculum Learning, <http://www.machinelearning.org/archive/icml2009/papers/119.pdf>
- [39] Graves 2014, Generating Sequences with Recurrent Neural Networks, <http://arxiv.org/pdf/1308.0850v5.pdf>
- [40] Pascanu, Gulcehr, Cho, Bengio 2014, How to Construct Deep Recurrent Neural Networks, <http://arxiv.org/pdf/1312.6026.pdf>
- [41] Hermans, Schrauwen 2013, Training and Analyzing Deep Recurrent Neural Networks <http://papers.nips.cc/paper/5166-training-and-analysing-deep-recurrent-neural-networks.pdf>
- [42] Chung, Gulcehr, Cho, Bengio 2015, Gated Feedback Recurrent Neural Networks, <http://arxiv.org/pdf/1502.02367.pdf>
- [43] Cho et al. 2014, Learning phrase representations using RNN encode-decoder for statistical machine translation, <http://arxiv.org/pdf/1406.1078.pdf>
- [44] Chung et al. 2014, Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, <http://arxiv.org/pdf/1412.3555.pdf>
- [45] Joulin, Mikolov 2015, Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets, <http://arxiv.org/pdf/1503.01007.pdf>
- [46] Weston, Chopra, Bordes 2015, Memory Networks, <http://arxiv.org/pdf/1410.3916v10.pdf>
- [47] Mikolov et al 2015, Learning Longer Memory in Recurrent Neural Networks, <http://arxiv.org/pdf/1412.7753.pdf>
- [48] Li, Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition (Course Notes), <http://cs231n.stanford.edu/>, <http://cs231n.github.io/>
- [49] He et al 2015: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, <http://arxiv.org/pdf/1502.01852v1.pdf>
- [50] Bergstra, Bengio 2012, Random Search for Hyper-Parameter Optimization, <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>
- [51] Le, Jaitly, Hinton 2015, A Simple Way to Initialize Recurrent Networks of Rectified Linear Units, <http://arxiv.org/abs/1504.00941>



## 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 runs 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.