Copyrighted Material, Do not distribute

This book is copyrighted by Heaton Research, Inc. If you obtained this book from a source other than Heaton Research please contact us at support@heatonresearch.com.

Book Title	Programming Neural Network s with Encog 2 in Java
ISBN	1604390077 – Rev 1 (March 2010)
Author	Jeff Heaton
E-Book Price	$19.99 (USD)

Programming Neural Networks with Encog 2 in Java

Programming Neural Networks with Encog 2 in Java

By Jeff Heaton

Heaton Research, Inc.

St. Louis, MO USA

Publisher: Heaton Research, Inc

Programming Neural Networks with Encog 2 in Java
March, 2010
Author: Jeff Heaton
Editor: WordsRU.com
Cover Art: Carrie Spear
ISBN's for all Editions:
1-60439-007-7, Softcover
1-60439-011-5, Adobe PDF e-book

Copyright © 2010 by Heaton Research Inc., 1734 Clarkson Rd. #107, Chesterfield, MO 63017-4976. World rights reserved. The author(s) created reusable code in this publication expressly for reuse by readers. Heaton Research, Inc. grants readers permission to reuse the code found in this publication or downloaded from our website so long as (author(s)) are attributed in any application containing the reusable code and the source code itself is never redistributed, posted online by electronic transmission, sold or commercially exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including, but not limited to photo copy, photograph, magnetic, or other record, without prior agreement and written permission of the publisher.

Heaton Research, Encog, the Encog Logo and the Heaton Research logo are all trademarks of Heaton Research, Inc., in the United States and/or other countries.

TRADEMARKS: Heaton Research has attempted through out this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, so the content is based upon the final release of software whenever possible. Portions of the manuscript may be based upon pre-release versions suppled by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance,

merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America.

SOFTWARE LICENSE AGREEMENT: TERMS AND CONDITIONS

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the "Software") to be used in connection with the book. Heaton Research, Inc. hereby grants to you a license to use and distribute software programs that make use of the compiled binary form of this book's source code. You may not redistribute the source code contained in this book, without the written permission of Heaton Research, Inc. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of Heaton Research, Inc. unless otherwise indicated and is protected by copyright to Heaton Research, Inc. or other copyright owner(s) as indicated in the media files (the "Owner(s)"). You are hereby granted a license to use and distribute the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of Heaton Research, Inc. and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties ("End-User License"), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

SOFTWARE SUPPORT

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material but they are not supported by Heaton Research, Inc.. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate README files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, Heaton Research, Inc. bears no responsibility. This notice concerning support for the Software is provided for your information only. Heaton Research, Inc. is not the agent or principal of the Owner(s), and Heaton Research, Inc. is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

WARRANTY

Heaton Research, Inc. warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from Heaton Research, Inc. in any other form or media than that enclosed herein or posted to www.heatonresearch.com. If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

Heaton Research, Inc.
Customer Support Department
1734 Clarkson Rd #107
Chesterfield, MO 63017-4976

Web: www.heatonresearch.com
E-Mail: support@heatonresearch.com

DISCLAIMER

Heaton Research, Inc. makes no warranty or representation, either expressed or implied, with respect to the Software or its

contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will Heaton Research, Inc., its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, Heaton Research, Inc. further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by Heaton Research, Inc. reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

SHAREWARE DISTRIBUTION

This Software may use various programs and libraries that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

This book is dedicated to my wonderful wife, Tracy. The first year of marriage has been great; I look forward to many more.

.

Table of Contents

Table of Contents **13**

Introduction **19**

The History of Encog 19

Problem Solving with Neural Networks 20

Structure of the Book 22

Chapter 1: Introduction to Encog **27**

What is a Neural Network? 28

Using a Neural Network 32

Chapter 2: Building Encog Neural Networks **47**

What are Layers and Synapses? 47

Understanding Encog Layers 48

Understanding Encog Synapses 54

Understanding Neural Logic 60

Understanding Properties and Tags 63

Building with Layers and Synapses 64

Chapter 3: Using Activation Functions **85**

The Role of Activation Functions 85

Encog Activation Functions 87

Chapter 4: Using the Encog Workbench **101**

Creating a Neural Network 103

Creating a Training Set 107

Training a Neural Network 109

Querying the Neural Network 112
Generating Code 114
Chapter 5: Propagation Training 119
Understanding Propagation Training 119
Propagation Training with Encog 122
Propagation and Multithreading 136
Chapter 6: Obtaining Data for Encog 147
Where to Get Data for Neural Networks 147
What is Normalization? 148
Using the DataNormalization Class 153
Running the Forest Cover Example 171
Understanding the Forest Cover Example 177
Chapter 7: Encog Persistence 207
Using Encog XML Persistence 208
Using Java Serialization 215
Format of the Encog XML Persistence File 218
Chapter 8: More Supervised Training 229
Running the Lunar Lander Example 230
Examining the Lunar Lander Simulator 235
Training the Neural Pilot 247
Using the Training Set Score Class 251
Chapter 9: Unsupervised Training Methods 257
The Structure and Training of a SOM 258
Implementing the Colors SOM in Encog 265
Chapter 10: Using Temporal Data 277

How a Predictive Neural Network Works 277
Using the Encog Temporal Dataset 279
Application to Sunspots 281
Using the Encog Market Dataset 291
Application to the Stock Market 293
Chapter 11: Using Image Data 311
Finding the Bounds 312
Downsampling an Image 313
Using the Encog Image Dataset 315
Image Recognition Example 317
Chapter 12: Recurrent Neural Networks 337
Encog Thermal Neural Networks 338
The Elman Neural Network 359
The Jordan Neural Network 366
Chapter 13: Structuring Hidden Layers 373
Understanding Hidden Layer Structure 373
Using Selective Pruning 374
Using Incremental Pruning 377
Chapter 14: Other Network Patterns 385
Radial Basis Function Networks 386
Adaptive Resonance Theory 393
Counter-Propagation Neural Networks 399
Where to Go from Here 414
Appendix A: Installing and Using Encog 419
Installing Encog 419

Compiling the Encog Core 421
Compiling and Executing Encog Examples 422
Using Encog with the Eclipse IDE 424
Appendix B: Example Locations 433
Appendix C: Encog Patterns 439
Adaline Neural Network 439
ART1 Neural Network 440
Bidirectional Associative Memory (BAM) 441
Boltzmann Machine 442
Counter-Propagation Neural Network 443
Elman Neural Network 445
Feedforward Neural Network 446
Hopfield Neural Network 447
Jordan Neural Network 448
Radial Basis Function Neural Network 450
Recurrent Self-Organizing Map 451
Self-Organizing Map 452
Glossary 455
Index 467

Introduction

Encog is an Artificial Intelligence (AI) Framework for Java and .Net. Though Encog supports several areas of AI outside of neural networks, the primary focus for the Encog 2.x versions is neural network programming. This book was published as Encog 2.3 was being released. It should stay very compatible with later editions of Encog 2. Future versions in the 2.x series will attempt to add functionality with minimal disruption to existing code.

The History of Encog

The first version of Encog, version 0.5 was released on July 10, 2008. However, the code for Encog originates from the first edition of "Introduction to Neural Networks with Java", which I published in 2005. This book was largely based on the Java Object Oriented Neural Engine (JOONE). Basing my book on JOONE proved to be problematic. The early versions of JOONE were quite promising, but JOONE quickly became buggy, with future versions introducing erratic changes that would frequently break examples in my book. As of 2010, with the writing of this book, the JOONE project seems mostly dead. The last release of JOONE was a "release candidate", that occurred in 2006. As of the writing of this book, in 2010, there have been no further JOONE releases.

The second edition of my book used 100% original code and was not based on any neural network API. This was a better environment for my "Introduction to Neural Networks for Java/C#" books, as I could give exact examples of how to implement the neural networks, rather than how to use an API. This book was released in 2008.

I found that many people were using the code presented in the book as a neural network API. As a result, I decided to package it as such. Version 0.5 of Encog is basically all of the book code combined into a package structure. Versions 1.0 through 2.0 greatly enhanced the neural network code well beyond what I would cover in an introduction book.

The goal of my "Introduction to Neural Networks with Java/C#" is to teach someone how to implement basic neural networks of their own. The goal of this book is to teach someone to use Encog to create more complex neural

network structures without the need to know how the underlying neural network code actually works.

These two books are very much meant to be read in sequence, as I try not to repeat too much information in this book. However, you should be able to start with Encog if you have a basic understanding of what neural networks are used for. You must also understand the Java programming language. Particularly, you should be familiar with the following:

- Java Generics
- Collections
- Object Oriented Programming

Before we begin examining how to use Encog, let's first take a look at what sorts of problems Encog might be adept at solving. Neural networks are a programming technique. They are not a silver bullet solution for every programming problem you will encounter. There are some programming problems that neural networks are extremely adept at solving. There are other problems for which neural networks will fail miserably.

Problem Solving with Neural Networks

A significant goal of this book is to show you how to construct Encog neural networks and to teach you when to use them. As a programmer of neural networks, you must understand which problems are well suited for neural network solutions and which are not. An effective neural network programmer also knows which neural network structure, if any, is most applicable to a given problem. This section begins by first focusing on those problems that are not conducive to a neural network solution.

Problems Not Suited to a Neural Network Solution

Programs that are easily written out as flowcharts are examples of problems for which neural networks are not appropriate. If your program consists of well-defined steps, normal programming techniques will suffice.

Another criterion to consider is whether the logic of your program is likely to change. One of the primary features of neural networks is their ability to learn. If the algorithm used to solve your problem is an unchanging business rule, there is no reason to use a neural network. In fact, it might be

detrimental to your application if the neural network attempts to find a better solution, and begins to diverge from the desired process and produces unexpected results.

Finally, neural networks are often not suitable for problems in which you must know exactly how the solution was derived. A neural network can be very useful for solving the problem for which it was trained, but the neural network cannot explain its reasoning. The neural network knows something because it was trained to know it. The neural network cannot explain how it followed a series of steps to derive the answer.

Problems Suited to a Neural Network

Although there are many problems for which neural networks are not well suited, there are also many problems for which a neural network solution is quite useful. In addition, neural networks can often solve problems with fewer lines of code than a traditional programming algorithm. It is important to understand which problems call for a neural network approach.

Neural networks are particularly useful for solving problems that cannot be expressed as a series of steps, such as recognizing patterns, classification, series prediction, and data mining.

Pattern recognition is perhaps the most common use for neural networks. For this type of problem, the neural network is presented a pattern. This could be an image, a sound, or any other data. The neural network then attempts to determine if the input data matches a pattern that it has been trained to recognize. There will be many examples in this book of using neural networks to recognize patterns.

Classification is a process that is closely related to pattern recognition. A neural network trained for classification is designed to take input samples and classify them into groups. These groups may be fuzzy, lacking clearly defined boundaries. Alternatively, these groups may have quite rigid boundaries.

Structure of the Book

This book begins with Chapter 1, "Getting Started with Encog". This chapter introduces you to the Encog API and what it includes. You are shown a simple example that teaches Encog to recognize the XOR operator.

The book continues with Chapter 2, "The Parts of an Encog Neural Network". In this chapter, you see how a neural network is constructed using Encog. You will see all of the parts of a neural network that later chapters will expand upon.

Chapter 3, "Using Activation Functions" shows what activation functions are and how they are used in Encog. You will be shown the different types of activation functions Encog makes available, as well as how to choose which activation function to use for a neural network.

Encog includes a GUI neural network editor called the Encog Workbench. Chapter 4, "Using the Encog Workbench" shows how to make use of this application. The Encog Workbench provides a GUI tool that can edit the .EG data files used by the Encog Framework.

To be of any real use, neural networks must be trained. There are several ways to train neural networks. Chapter 5, "Propagation Training" shows how to use the propagation methods built into Encog. Encog supports backpropagation, resilient propagation, the Manhattan update rule, and SCG.

One of the primary tasks for neural networks is to recognize and provide insight into data. Chapter 6, "Obtaining Data for Encog" shows how to process this data before use with a neural network. In this chapter we will examine some data that might be used with a neural network. You will be shown how to normalize this data and use it with a neural network.

Encog can store data in .EG files. These files hold both data and the neural networks themselves. Chapter 7, "Encog Persistence" introduces the .EG format and shows how to use the Encog Framework to manipulate these files. The .EG files are represented as standard XML, so they can easily be used in programs other than of Encog.

Chapter 8, "Other Supervised Training Methods" shows some of the other supervised training algorithms supported by Encog. Propagation training is

not the only way to train a neural network. This chapter introduces simulated annealing and genetic algorithms as training techniques for Encog networks. You are also shown how to create hybrid training algorithms.

Supervised training is not the only training option. Chapter 9, "Unsupervised Training Methods" shows how to use unsupervised training with Encog. Unsupervised training occurs when a neural network is given sample input, but no expected output.

A common use of neural networks is to predict future changes in data. One common use for this is to attempt to predict trends in the stock market. Chapter 10, "Using Temporal Data" will show how to use Encog to predict trends.

Images are frequently used as an input for neural networks. Encog contains classes that make it easy to use image data to feed and train neural networks. Chapter 11, "Using Image Data" shows how to use image data with Encog.

Recurrent neural networks are a special class of neural networks where the layers do not simply flow forward, like the feedforward neural networks that are so common. Chapter 12, "Recurrent Neural Networks" shows how to construct recurrent neural networks with Encog. The Elman and Jordan type neural networks will be discussed.

It can be difficult to determine how the hidden layers of a neural network should be constructed. Chapter 13, "Pruning and Structuring Networks" shows how Encog can automatically provide some insight into the structure of neural networks. Selective pruning can be used to remove neurons that are redundant. Incremental pruning allows Encog to successively tray more complex hidden layer structures and attempt to determine which will be optimal.

Chapter 14, "Common Neural Network Patterns" shows how to use Encog patterns. Often, neural network applications will need to use a common neural network pattern. Encog provides patterns for many of these common neural network types. This saves you the trouble of manually creating all of the layers, synapses and tags necessary to create each of these common neural network types. Using the pattern classes you will be able to simply describe certain parameters of each of these patterns, and then Encog will automatically create such a neural network for you.

As you read though this book you will undoubtedly have questions about the Encog Framework. One of the best places to go for answers is the Encog forums at Heaton Research. You can find the Heaton Research forums at the following URL:

http://www.heatonresearch.com/forum

Chapter 1: Introduction to Encog

- The Encog Framework
- What is a Neural Network?
- Using a Neural Network
- Training a Neural Network

Artificial neural networks are programming techniques that attempt to emulate the human brain's biological neural networks. Artificial neural networks (ANNs) are just one branch of artificial intelligence (AI). This book focuses primarily on artificial neural networks, frequently called simply neural networks, and the use of the Encog Artificial Intelligence Framework, usually just referred to as Encog. Encog is an open source project that provides neural network and HTTP bot functionality.

This book explains how to use neural networks with Encog and the Java programming language. The emphasis is on how to use the neural networks, rather than how to actually create the software necessary to implement a neural network. Encog provides all of the low-level code necessary to construct many different kinds of neural networks. If you are interested in learning to actually program the internals of a neural network, using Java, you may be interested in the book "Introduction to Neural Networks with Java" (ISBN: 978-1604390087).

Encog provides the tools to create many different neural network types. Encog supports feedforward, recurrent, self organizing maps, radial basis function and Hopfield neural networks. The low-level types provided by Encog can be recombined and extended to support additional neural network architectures as well. The Encog Framework can be obtained from the following URL:

http://www.encog.org/

Encog is released under the Lesser GNU Public License (LGPL). All of the source code for Encog is provided in a Subversion (SVN) source code repository provided by the Google Code project. Encog is also available for the Microsoft .Net platform.

Encog neural networks, and related data, can be stored in .EG files. These files can be edited by a GUI editor provided with Encog. The Encog Workbench allows you to edit, train and visualize neural networks. The Encog Workbench can also generate code in Java, Visual Basic or C#. The Encog Workbench can be downloaded from the above URL.

What is a Neural Network?

We will begin by examining what exactly a neural network is. A simple feedforward neural network can be seen in Figure 1.1. This diagram was created with the Encog Workbench. It is not just a diagram; this is an actual functioning neural network from Encog as you would actually edit it.

Figure 1.1: Simple Feedforward Neural Network

Networks can also become more complex than the simple network above. Figure 1.2 shows a recurrent neural network.

Figure 1.2: Simple Recurrent Neural Network

Looking at the above two neural networks you will notice that they are composed of layers, represented by the boxes. These layers are connected by lines, which represent synapses. Synapses and layers are the primary building blocks for neural networks created by Encog. The next chapter focuses solely on layers and synapses.

Before we learn to build neural networks with layers and synapses, let's first look at what exactly a neural network is. Look at Figures 1.1 and 1.2. They are quite a bit different, but they share one very important characteristic. They both contain a single input layer and a single output layer. What happens between these two layers is very different, between the two networks. In this chapter, we will focus on what comes into the input layer and goes out of the output layer. The rest of the book will focus on what happens between these two layers.

Almost every neural network seen in this book will have, at a minimum, an input and output layer. In some cases, the same layer will function as both input and output layer. You can think of the general format of any neural network found in this book as shown in Figure 1.3.

Figure 1.3: Generic Form of a Neural Network

To adapt a problem to a neural network, you must determine how to feed the problem into the input layer of a neural network, and receive the solution through the output layer of a neural network. We will look at the input and output layers in this chapter. We will then determine how to structure the input and interpret the output. The input layer is where we will start.

Understanding the Input Layer

The input layer is the first layer in a neural network. This layer, like all layers, has a specific number of neurons in it. The neurons in a layer all contain similar properties. The number of neurons determines how the input to that layer is structured. For each input neuron, one **double** value is stored. For example, the following array could be used as input to a layer that contained five neurons.

```
double[] input = new double[5];
```

The input to a neural network is always an array of doubles. The size of this array directly corresponds to the number of neurons on this hidden layer. Encog uses the class **NeuralData** to hold these arrays. You could easily convert the above array into a **NeuralData** object with the following line of code.

```
NeuralData data = new BasicNeuralData(input);
```

The interface **NeuralData** defines any "array like" data that may be presented to Encog. You must always present the input to the neural network inside of a **NeuralData** object. The class **BasicNeuralData** implements the **NeuralData** interface. The class **BasicNeuralData** is not the only way to provide Encog with data. There are other implementations of **NeuralData**, as well. We will see other implementations later in the book.

The **BasicNeuralData** class simply provides a memory-based data holder for the neural network. Once the neural network processes the input, a **NeuralData** based class will be returned from the neural network's output layer. The output layer is discussed in the next section.

Understanding the Output Layer

The output layer is the final layer in a neural network. The output layer provides the output after all of the previous layers have had a chance to process the input. The output from the output layer is very similar in format to the data that was provided to the input layer. The neural network outputs an array of doubles.

The neural network wraps the output in a class based on the **NeuralData** interface. Most of the built in neural network types will return a **BasicNeuralData** class as the output. However, future, and third party, neural network classes may return other classes based other implementations of the **NeuralData** interface.

Neural networks are designed to accept input, which is an array of doubles, and then produce output, which is also an array of doubles. Determining how to structure the input data, and attaching meaning to the output, are two of the main challenges to adapting a problem to a neural network. The real power of a neural network comes from its pattern recognition capabilities. The neural network should be able to produce the desired output even if the input has been slightly distorted.

Hidden Layers

As previously discussed, neural networks contain and input layer and an output layer. Sometimes the input layer and output layer are the same. Often the input and output layer are two separate layers. Additionally, other

layers may exist between the input and output layers. These layers are called hidden layers. These hidden layers can be simply inserted between the input and output layers. The hidden layers can also take on more complex structures.

The only purpose of the hidden layers is to allow the neural network to better produce the expected output for the given input. Neural network programming involves first defining the input and output layer neuron counts. Once you have defined how to translate the programming problem into the input and output neuron counts, it is time to define the hidden layers.

The hidden layers are very much a "black box". You define the problem in terms of the neuron counts for the hidden and output layers. How the neural network produces the correct output is performed, in part, by the hidden layers. Once you have defined the structure of the input and output layers you must define a hidden layer structure that optimally learns the problem. If the structure of the hidden layer is too simple it may not learn the problem. If the structure is too complex, it will learn the problem but will be very slow to train and execute.

Later chapters in this book will discuss many different hidden layer structures. You will learn how to pick a good structure, based on the problem that you are trying to solve. Encog also contains some functionality to automatically determine a potentially optimal hidden layer structure. Additionally, Encog also contains functions to prune back an overly complex structure. Chapter 13, "Pruning and Structuring Networks" shows how Encog can help create a potentially optimal structure.

Some neural networks have no hidden layers. The input layer may be directly connected to the output layer. Further, some neural networks have only a single layer. A single layer neural network has the single layer self-connected. These connections permit the network to learn. Contained in these connections, called synapses, are individual weight matrixes. These values are changed as the neural network learns. We will learn more about weight matrixes in the next chapter.

Using a Neural Network

We will now look at how to structure a neural network for a very simple problem. We will consider creating a neural network that can function as an XOR operator. Learning the XOR operator is a frequent "first example" when

demonstrating the architecture of a new neural network. Just as most new programming languages are first demonstrated with a program that simply displays "Hello World", neural networks are frequently demonstrated with the XOR operator. Learning the XOR operator is sort of the "Hello World" application for neural networks.

The XOR Operator and Neural Networks

The XOR operator is one of three commonly used Boolean logical operators. The other two are the AND and OR operators. For each of these logical operators, there are four different combinations. For example, all possible combinations for the AND operator are shown below.

```
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
```

This should be consistent with how you learned the AND operator for computer programming. As its name implies, the AND operator will only return true, or one, when both inputs are true.

The OR operator behaves as follows.

```
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
```

This also should be consistent with how you learned the OR operator for computer programming. For the OR operator to be true, either of the inputs must be true.

The "exclusive or" (XOR) operator is less frequently used in computer programming, so you may not be familiar with it. XOR has the same output as the OR operator, except for the case where both inputs are true. The possible combinations for the XOR operator are shown here.

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

As you can see the XOR operator only returns true when both inputs differ. In the next section we will see how to structure the input, output and hidden layers for the XOR operator.

Structuring a Neural Network for XOR

There are two inputs to the XOR operator and one output. The input and output layers will be structured accordingly. We will feed the input neurons the following **double** values:

```
0.0,0.0
1.0,0.0
0.0,1.0
1.0,1.0
```

These values correspond to the inputs to the XOR operator, shown above. We will expect the one output neuron to produce the following **double** values:

```
0.0
1.0
1.0
0.0
```

This is one way that the neural network can be structured. This method allows a simple feedforward neural network to learn the XOR operator. The feedforward neural network, also called a perceptron, is one of the first neural network architectures that we will learn.

There are other ways that the XOR data could be presented to the neural network. Later in this book we will see two examples of recurrent neural networks. We will examine the Elman and Jordan styles of neural networks. These methods would treat the XOR data as one long sequence. Basically concatenate the truth table for XOR together and you get one long XOR sequence, such as:

```
0.0,0.0,0.0,
0.0,1.0,1.0,
1.0,0.0,1.0,
1.0,1.0,0.0
```

The line breaks are only for readability. This is just treating XOR as a long sequence. By using the data above, the network would have a single input neuron and a single output neuron. The input neuron would be fed one value from the list above, and the output neuron would be expected to return the next value.

This shows that there is often more than one way to model the data for a neural network. How you model the data will greatly influence the success of

your neural network. If one particular model is not working, you may need to consider another. For the examples in this book we will consider the first model we looked at for the XOR data.

Because the XOR operator has two inputs and one output, the neural network will follow suit. Additionally, the neural network will have a single hidden layer, with two neurons to help process the data. The choice for 2 neurons in the hidden layer is arbitrary, and often comes down to trial and error. The XOR problem is simple, and two hidden neurons are sufficient to solve it. A diagram for this network can be seen in Figure 1.4.

Figure 1.4: Neuron Diagram for the XOR Network

Usually, the individual neurons are not drawn on neural network diagrams. There are often too many. Similar neurons are grouped into layers. The Encog workbench displays neural networks on a layer-by-layer basis. Figure 1.5 shows how the above network is represented in Encog.

Figure 1.5: Encog Layer Diagram for the XOR Network

The code needed to create this network is relatively simple.

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(2));
network.addLayer(new BasicLayer(2));
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
network.reset();
```

In the above code you can see a **BasicNetwork** being created. Three layers are added to this network. The first layer, which becomes the input layer, has two neurons. The hidden layer is added second, and it has two neurons also. Lastly, the output layer is added, which has a single neuron. Finally, the **finalizeStructure** method must be called to inform the network that no more layers are to be added. The call to **reset** randomizes the weights in the connections between these layers.

Neural networks frequently start with a random weight matrix. This provides a starting point for the training methods. These random values will be tested and refined into an acceptable solution. However, sometimes the initial random values are too far off. Sometimes it may be necessary to reset the weights again, if training is ineffective.

These weights make up the long-term memory of the neural network. Additionally, some layers have threshold values that also contribute to the

long-term memory of the neural network. Some neural networks also contain context layers which give the neural network a short-term memory as well. The neural network learns by modifying these weight and threshold values. We will learn more about weights and threshold values in Chapter 2, "The Parts of an Encog Neural Network".

Now that the neural network has been created, it must be trained. Training is discussed in the next section.

Training a Neural Network

To train the neural network, we must construct a **NeuralDataSet** object. This object contains the inputs and the expected outputs. To construct this object, we must create two arrays. The first array will hold the input values for the XOR operator. The second array will hold the ideal outputs for each of 115 corresponding input values. These will correspond to the possible values for XOR. To review, the four possible values are as follows:

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

First we will construct an array to hold the four input values to the XOR operator. This is done using a two dimensional **double** array. This array is as follows:

```
public static double XOR_INPUT[][] = {
{ 0.0, 0.0 },
{ 1.0, 0.0 },
{ 0.0, 1.0 },
{ 1.0, 1.0 } };
```

Likewise, an array must be created for the expected outputs for each of the input values. This array is as follows:

```
public static double XOR_IDEAL[][] = {
{ 0.0 },
{ 1.0 },
{ 1.0 },
{ 0.0 } };
```

Even though there is only one output value, we must still use a two-dimensional array to represent the output. If there had been more than one output neuron, there would have been additional columns in the above array.

Now that the two input arrays have been constructed a **NeuralDataSet** object must be created to hold the training set. This object is created as follows.

```
NeuralDataSet trainingSet = new BasicNeuralDataSet(XOR_INPUT,
XOR_IDEAL);
```

Now that the training set has been created, the neural network can be trained. Training is the process where the neural network's weights are adjusted to better produce the expected output. Training will continue for many iterations, until the error rate of the network is below an acceptable level. First, a training object must be created. Encog supports many different types of training.

For this example we are going to use Resilient Propagation (RPROP). RPROP is perhaps the best general-purpose training algorithm supported by Encog. Other training techniques are provided as well, as certain problems are solved better with certain training techniques. The following code constructs a RPROP trainer.

```
final Train train = new ResilientPropagation(network,
trainingSet);
```

All training classes implement the **Train** interface. The RPROP algorithm is implemented by the **ResilientPropagation** class, which is constructed above.

Once the trainer has been constructed the neural network should be trained. Training the neural network involves calling the **iteration** method on the **Train** class until the error is below a specific value.

```
int epoch = 1;
do {
  train.iteration();
  System.out.println("Epoch #" + epoch + " Error:"
    + train.getError());
  epoch++;
} while(train.getError() > 0.01);
```

The above code loops through as many iterations, or epochs, as it takes to get the error rate for the neural network to be below 1%. Once the neural network has been trained, it is ready for use. The next section will explain how to use a neural network.

Executing a Neural Network

Making use of the neural network involves calling the **compute** method on the **BasicNetwork** class. Here we loop through every training set value and display the output from the neural network.

```
System.out.println("Neural Network Results:");

for(NeuralDataPair pair: trainingSet ) {
  final NeuralData output =
    network.compute(pair.getInput());

  System.out.println(pair.getInput().getData(0)
    + "," + pair.getInput().getData(1)
    + ", actual=" + output.getData(0) + ",ideal=" +

  pair.getIdeal().getData(0));
}
```

The **compute** method accepts a **NeuralData** class and also returns a **NeuralData** object. This contains the output from the neural network. This output is displayed to the user. With the program is run the training results are first displayed. For each Epoch, the current error rate is displayed.

```
Epoch #1 Error:0.5604437512295236
Epoch #2 Error:0.5056375155784316
Epoch #3 Error:0.5026960720526166
Epoch #4 Error:0.4907299498390594
...
Epoch #104 Error:0.01017278345766472
Epoch #105 Error:0.010557202078697751
Epoch #106 Error:0.011034965164672806
Epoch #107 Error:0.009682102808616387
```

The error starts at 56% at epoch 1. By epoch 107 the training has dropped below 1% and training stops. Because neural network was initialized with random weights, it may take different numbers of iterations to train each time the program is run. Additionally, though the final error rate may be different, it should always end below 1%.

Finally, the program displays the results from each of the training items as follows:

```
Neural Network Results:
0.0,0.0, actual=0.002782538818034049,ideal=0.0
1.0,0.0, actual=0.9903741937121177,ideal=1.0
0.0,1.0, actual=0.9836807956566187,ideal=1.0
1.0,1.0, actual=0.0011646072586172778,ideal=0.0
```

As you can see, the network has not been trained to give the exact results. This is normal. Because the network was trained to 1% error, each of the results will also be within generally 1% of the expected value.

Because the neural network is initialized to random values, the final output will be different on second run of the program.

```
Neural Network Results:
0.0,0.0, actual=0.005489822214926685,ideal=0.0
1.0,0.0, actual=0.985425090860287,ideal=1.0
0.0,1.0, actual=0.9888064742994463,ideal=1.0
1.0,1.0, actual=0.005923146369557053,ideal=0.0
```

Above, you see a second run of the program. The output is slightly different. This is normal.

This is the first Encog example. You can see the complete program in Listing 1.1. All of the examples contained in this book are also included with the examples downloaded with Encog. For more information on how to download these examples and where this particular example is located, refer to Appendix A, “Installing Encog”.

Listing 1.1: Solve XOR with RPROP

```
package org.encog.examples.neural.xorresilient;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.
  resilient.ResilientPropagation;
import org.encog.util.logging.Logging;

public class XORResilient {
```

```
  public static double XOR_INPUT[][] =
    { { 0.0, 0.0 },
      { 1.0, 0.0 },
       { 0.0, 1.0 },
      { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
      { 0.0 },
      { 1.0 },
      { 1.0 },
      { 0.0 } };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();

    BasicNetwork network = new BasicNetwork();
    network.addLayer(
      new BasicLayer(new ActivationSigmoid(),
        false,2));
    network.addLayer(
      new BasicLayer(new ActivationSigmoid(),
      false,4));
    network.addLayer(new BasicLayer(
      new ActivationSigmoid(), false,1));

    network.getStructure().finalizeStructure();
    network.reset();

    NeuralDataSet trainingSet =
      new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

// train the neural network
    final Train train = new ResilientPropagation(
      network,
      trainingSet);


    int epoch = 1;

    do {
      train.iteration();
      System.out.println("Epoch #"
        + epoch
        + " Error:"
        + train.getError());
      epoch++;
```

```
      } while(train.getError() > 0.01);

// test the neural network
  System.out.println("Neural Network Results:");
  for(NeuralDataPair pair: trainingSet ) {
    final NeuralData output =
      network.compute(pair.getInput());
    System.out.println(
      pair.getInput().getData(0)
      + ","
      + pair.getInput().getData(1)
      + ", actual="
      + output.getData(0)
      + ",ideal="
      + pair.getIdeal().getData(0));
    }
  }
}
```

Chapter Summary

Encog is a framework that allows you to create neural networks or bot applications. This chapter focused on using Encog to create neural network applications. This book focuses on the overall layout of a neural network. In this chapter, you saw how to create an Encog application that could learn the XOR operator.

Neural networks are made up of layers. These layers are connected by synapses. The synapses contain weights that make up the memory of the neural network. Some layers also contain threshold values that also contribute to the memory of the neural network. Together, thresholds and weights make up the long-term memory of the neural network. Networks can also contain context layers. Context layers are used to form a short-term memory.

There are several different layer types supported by Encog. However, these layers fall into three groups, depending on where they are placed in the neural network. The input layer accepts input from the outside. Hidden layers accept data from the input layer for further processing. The output layer takes data, either from the input or final hidden layer, and presents it on to the outside world.

The XOR operator was used as an example for this chapter. The XOR operator is frequently used as a simple “Hello World” application for neural networks. The XOR operator provides a very simple pattern that most

neural networks can easily learn. It is important to know how to structure data for a neural network. Neural networks both accept and return an array of floating point numbers.

This chapter introduced layers and synapses. You saw how they are used to construct a simple neural network. The next chapter will greatly expand on layers and synapses. You will see how to use the various layer and synapse types offered by Encog to construct neural networks.

Questions for Review

1. Explain the role of the input layer, the output layer and hidden layers.

2. What form does the input to a neural network take? What form is the output from a neural network?

3. How does a neural network implement long-term memory? How does a neural network implement short-term memory?

4. Where does Encog store the weight matrix values? Where does Encog store the threshold values?

5. What is the best "general purpose" training method for an Encog neural network?

Terms

Artificial Intelligence

Artificial Neural Network

Biological Neural Network

Black Box

Context Layer

Encog

Encog File

Encog Workbench

Error Rate

Feedforward Neural network

Hidden Layer

Input Layer

Iteration

Layer

LGPL

Long Term Memory

Neural Network

Output Layer

Recurrent Neural Network

Resilient Propagation

Short Term Memory

Synapse

Training

Training Set

XOR Operator

Chapter 2: Building Encog Neural Networks

- What are Layers and Synapses?
- Encog Layer Types
- Encog Synapse Types
- Neural Network Properties
- Neural Network Logic
- Building with Layers and Synapses

Encog neural networks are made up of layers, synapses, properties and a logic definition. In this chapter we will examine the various types of layers and synapses supported by Encog. You will see how the layer and synapse types can be combined to create a variety of neural network types.

What are Layers and Synapses?

A layer is a collection of similar neurons. All neurons in the layer must share exactly the same characteristic as other neurons in this layer. A layer accepts a parameter that specifies how many neurons that layer is to have. Layers hold an array of threshold values. There is one threshold value for each of the neurons in the layer. The threshold values, along with the weight matrix form the long-term memory of the neural network. Some layers also hold context values that make up the short-term memory of the neural network.

A synapse is used to connect one layer to another. The synapses contain the weight matrixes used by the neural network. The weight matrixes hold the connection values between each of the neurons in the two layers that are connected by this synapse.

Every Encog neural network contains a neural logic class. This class defines how a neural network processes its layers and synapses. A neural logic class must implement the **NeuralLogic** interface. Every Encog neural network must have a **NeuralLogic** based logic class. Without such a class the network would not be able to process incoming data. **NeuralLogic** classes allow Encog to be compatible with a wide array of neural network types.

Some **NeuralLogic** classes require specific layer types. For the **NeuralLogic** classes to find these layers, the layers must be tagged. Tagging allows a type to be assigned to any layer in the neural network. Not all layers need to be tagged.

Neural network properties are stored in a collection of name-value pairs. They are stored in a simple **Map** structure. Some **NeuralLogic** classes require specific parameters to be set for them to operate. These parameters are stored in the neural network properties.

Neural networks are constructed of layers and synapses. There are several different types of layers and synapses, provided by Encog. This chapter will introduce all of the Encog layer types and synapse types. We will begin by examining the Encog layer types.

Understanding Encog Layers

There are a total of three different layer types used by Encog. In this section we will examine each of these layer types. All three of these layer types implement the **Layer** interface. As additional layer types are added to Encog, they will support the **Layer** interface as well. We will begin by examining the **Layer** interface.

Using the Layer Interface

The **Layer** interface defines many important methods that all layers must support. Additionally, most Encog layers implement a constructor that initializes that unique type of layer. Listing 2.1 shows the **Layer** interface.

Listing 2.1: The Layer Interface

```
public interface Layer extends EncogPersistedObject {
  void addNext(Layer next);
  void addNext(Layer next, SynapseType type);
  void addSynapse(Synapse synapse);
  NeuralData compute(final NeuralData pattern);
  ActivationFunction getActivationFunction();
  int getNeuronCount();
  List<Synapse> getNext();
  Collection<Layer> getNextLayers();
  double[] getThreshold();
```

```
  double getThreshold(int index);
  int getX();
  int getY();
  boolean hasThreshold();
  boolean isConnectedTo(Layer layer);
  void process(final NeuralData pattern);
  NeuralData recur();
  void setNeuronCount(int neuronCount);
  void setThreshold(double[] d);
  void setThreshold(int index, double d);
  void setX(int x);
  void setY(int y);
  void setActivationFunction(
    ActivationFunction activationFunction);
}
```

As you can see there are a number of methods that must be implemented to create a layer. We will now review some of the more important methods.

The **addNext** method is used to connect another layer to this one. The next layer is connected with a **Synapse**. There are two overloads to the **addNext** method. The first allows you to simply specify the next layer and a **WeightedSynapse** is automatically created to connect the new layer. The second allows you to specify the next layer and use the **SynapseType** enumeration to specify what type of synapse you would like to connect the two layers. Additionally, the **addSynapse** method allows you to simply pass in an already created **Synapse**.

The **getNext** method can be called to get a **List** of the **Synapse** objects used to connect to the next layers. Additionally the **getNextLayers** method can be used to determine which layers this **Layer** is connected to. To see if this **Layer** is connected to another specific **Layer** call the **isConnectedTo** method.

The **setThreshold** and **getThreshold** methods allow access to the threshold values for this layer. The threshold values are numeric values that change as the neural network is trained, together with the weight matrix values; they form the long-term memory of the neural network. Not all layers have threshold values, the **hasThreshold** method can be used to determine if a layer has threshold values.

The **setActivation** and **getActivation** methods allow access to the activation function. Activation functions are mathematical functions that scale the output from a neuron layer. Encog supports many different activation functions. Activation functions will be covered in much greater detail in the next chapter.

Finally, the **compute** method is provided that applies the activation function and does any other internal processing necessary to **compute** the output from this layer. You will not usually call **compute** directly, rather you will call the **compute** method on the **Network** that this layer is attached to, and it will call the appropriate **compute** functions for its various layers.

Using the Basic Layer

The **BasicLayer** implements the **Layer** interface. The **BasicLayer** class has two primary purposes. First, many types of neural networks can be built completely from **BasicLayer** objects, as it is a very useful layer in its own right. Second, the **BasicLayer** provides the basic functionality that some other layers require. As a result, some of the other layers in Encog subclass are base on the **BasicLayer** class.

The most basic form of the **BasicLayer** constructor accepts a single integer parameter that specifies how many neurons this layer will have. This constructor creates a layer that uses threshold values and the hyperbolic tangent function as the activation function. For example, the following code creates three layers, with varying numbers of neurons.

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(2));
network.addLayer(new BasicLayer(3));
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
```

If you would like more control over the layer, you can use a more advanced constructor. The following constructor allows you to specify the activation function, as well as if threshold values should be used.

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,2));
```

```
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,3));
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,1));
network.getStructure().finalizeStructure();
```

The above code creates the same sort of network as the previous code segment; however, a sigmoid activation function is used. The **true** parameter means that threshold values should be used. Some neural network architectures do not use threshold values, while others do. As you progress through this book you will see both networks that use, as well as those that do not use threshold values.

The **BasicLayer** class is used for many neural network types in this book.

Using the Context Layer

The **ContextLayer** class implements a contextual layer. This layer allows the neural network to have a short-term memory. The context layer always remembers the last input values that were fed to it. This causes the context layer to always output what it originally received on the previous run of the neural network. This allows the neural network to remember the last data that was fed to it on the previous run. The context layer is always one iteration behind.

Context layers are usually used with a recurrent neural network. Recurrent neural networks do not feed the layers just forward. Layers will be connected back into the flow of the neural network. Chapter 12, "Recurrent Neural Networks", will discuss recurrent neural networks in greater detail. Two types of neural network that make use of the **ContextLayer** are the Elman and Jordan neural networks. Elman and Jordan neural networks will also be covered in Chapter 12. The following code segment shows how to create a **ContextLayer**.

```
final Layer context = new ContextLayer(2);
final BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(1));
network.addLayer(hidden = new BasicLayer(2));
hidden.addNext(context, SynapseType.OneToOne);
context.addNext(hidden);
```

```
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
```

The above code shows a **ContextLayer** used with regular **BasicLayer** objects. The output from the hidden layer in the above neural network not only goes to the output layer. The output from the hidden layer also is fed into the **ContextLayer**. A **OneToOneSynapse** is used to feed the **ContextLayer**. We simply want the context layer to remember the output from the hidden layer; we do not want any processing. A **WeightedSynapse** is fed out of the **ContextLayer** because we do want additional processing. We want the neural network to learn from the output of the **ContextLayer**.

These features allow the **ContextLayer** to be very useful for recognizing sequences of input data. The patterns are no longer mutually exclusive when you use a **ContextLayer**. If "Pattern A" is presented to the neural network, followed by "Pattern B", it is much different than "Pattern B" being presented first. Without a context layer, the order would not matter.

Using the Radial Basis Function Layer

The **RadialBasisFunctionLayer** object implements a radial basis function (RBF) layer. This layer type is based on one or more RBF functions. A radial basis function reaches a peak and decreases quickly on both sides of the graph. One of the most common radial basis functions is the Gaussian function. The Gaussian function is the default option for the **RadialBasisFunctionLayer** class. You can see the Gaussian function in Figure 2.1.

Figure 2.1: The Gaussian Function

The above figure shows a graph of the Gaussian function. Usually several Gaussian functions are combined to create a **RadialBasisFunctionLayer**. Figure 2.2 shows a **RadialBasisFunctionLayer** being edited in the Encog Workbench. Here you can see that this layer is made up of multiple Gaussian functions.

Figure 2.2: An RBF Layer in Encog Workbench

The following code segment shows the **RadialBasisFunctionLayer** as part of a RBF neural network.

```
RadialBasisFunctionLayer rbfLayer;
final BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(new ActivationLinear(),
false, 2));
network.addLayer(rbfLayer = new RadialBasisFunctionLayer(4),
SynapseType.Direct);
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
network.reset();
rbfLayer.randomizeGaussianCentersAndWidths(0, 1);
```

As you can see from the above code, the RBF layer is used as a hidden layer between two **BasicLayer** objects. RBF layers will be discussed in greater detail in Chapter 14, "Common Neural Network Patterns".

Understanding Encog Synapses

In the previous section you saw how neural networks are made up of layers. Synapses are used to connect these layers together. Encog supports a variety of different synapse types. Synapses are used to carry information between the levels of a neural network. The synapses differ primarily in how the neurons are connected and what processing should be done on the information as it flows between levels.

Some of the synapse types supported by Encog make use of weight matrixes. A weight matrix allows the connections between each of the source layer neurons to have its connection weighted to the target neuron layer. By adjusting each of these weights, the neural network can learn.

In the next section you will learn about the synapse types that Encog supports. Any synapse that Encog makes use of must support the **Synapse** interface. This interface is discussed in the next section.

The Synapse Interface

The **Synapse** interface defines all of the essential methods that a class must support to function as a synapse. The **Synapse** interface is shown in Listing 2.2.

Listing 2.2: The Synapse Interface

```
public interface Synapse extends EncogPersistedObject {
  Object clone();
  NeuralData compute(NeuralData input);
  Layer getFromLayer();
  int getFromNeuronCount();
  Matrix getMatrix();
  int getMatrixSize();
  Layer getToLayer();
  int getToNeuronCount();
  SynapseType getType();
  boolean isSelfConnected();
  boolean isTeachable();
  void setFromLayer(Layer fromLayer);
  void setMatrix(final Matrix matrix);
  void setToLayer(Layer toLayer);
}
```

As you can see there are a number of methods that must be implemented to create a synapse. We will now review some of the more important methods.

The **getFromLayer** and **getToLayer** methods can be used to find the source and target layers for the neural synapse. The **isSelfConnected** can also be used to determine if the synapse creates a self-connected layer. Encog also supports self-connected layers. A layer is self-connected if it has a self-connected synapse. A self-connected synapse is a synapse where the "from layer" and "to layer" are the same layer.

The **getMatrix** and **setMatrix** methods allow access to the weight matrix for the neural network. A neural network that has a weight matrix is "teachable", and the **isTeachable** method will return **true**. The **getMatrixSize** method can also be called to determine the size of the weight matrix.

Finally, the **compute** method is provided that applies any synapse specific transformation, such as weight matrixes. You will not usually call **compute** directly, rather you will call the **compute** method on the **Network** that this layer is attached to, and it will call the appropriate **compute** functions for its various synapses.

Constructing Synapses

Often the synapses are simply created in the background and the programmer is not really aware of what type of synapse is even being created. The **addLayer** method of the **BasicNetwork** class automatically creates a new **WeightedSynapse** every time a new layer is added to the neural network.

The **addLayer** method of the **BasicNetwork** class hides quite a bit of complexity. However, it is useful to see what is actually going on, and how the synapses are created. The following lines of code will show how to create a neural network "from scratch", where every object that is needed to create the neural network is created by hand. The first step is to create a **BasicNetwork** object to hold the layers.

```
network = new BasicNetwork();
```

Next, we create three layers. Hidden, input and output layers are all created.

```
Layer inputLayer = new BasicLayer(new ActivationSigmoid(),
  true,2);
Layer hiddenLayer = new BasicLayer(
  new ActivationSigmoid(), true,2);
Layer outputLayer = new BasicLayer(
  new ActivationSigmoid(), true,1);
```

Two synapses are needed to connect these three layers together. One synapse holds the input to the hidden layer. The second synapse holds the hidden to the output layer. These synapses are created by the following lines of code.

```
Synapse synapseInputToHidden = new
WeightedSynapse(inputLayer,hiddenLayer);
Synapse synapseHiddenToOutput = new
WeightedSynapse(hiddenLayer,outputLayer);
```

These synapses can then be added to the two layers they originate from.

```
inputLayer.getNext().add(synapseInputToHidden);
hiddenLayer.getNext().add(synapseHiddenToOutput);
```

The **BasicNetwork** object should be informed what the input and output layer. Finally, the network structure should be finalized and the weight matrix and threshold values reset.

```
network.tagLayer(BasicNetwork.TAG_INPUT, inputLayer);
network.tagLayer(BasicNetwork.TAG_OUTPUT, outputLayer);
network.getStructure().finalizeStructure();
network.reset();
```

This section will discuss the different types of synapses supported by Encog. We will begin with the weighted synapse.

Using the WeightedSynapse Class

The weighted synapse is perhaps the most commonly used synapse type in Encog. The **WeightedSynapse** class is used by many different neural network architectures. Any place that a learning synapse is needed, the **WeightedSynapse** class is a good candidate. The **WeightedSynapse** connects every neuron in the source layer with every neuron in the target layer. Figure 2.3 shows a diagram of the weighted synapse.

Figure 2.3: The Weighted Synapse

This is the default synapse type for Encog. To create a weighted synapse object usually you will simply add a layer to the network. The default synapse type is the weighted synapse. You can also construct a weighted synapse object with the following line of code.

```
Synapse synapse = new WeightedSynapse(from,to);
```

Once the weighted synapse has been created, it can be added to the next collection on the source layer's target.

Using the Weightless Synapse

The weightless synapse works very similar to the weighted synapse. The primary difference is that there are no weights in the weightless synapse. It provides a connection between each of the neurons in the source layer to every other neuron in the target layer. Figure 2.4 shows the weightless synapse.

Figure 2.4: The Weightless Synapse

The weightless synapse is implemented inside of the **WeightlessSynapse** class. The following line of code will construct a weightless synapse.

```
Synapse synapse = new WeightlessSynapse(from,to);
```

The weightless synapse is used when you would like to fully connect two layers, but want the information to pass through to the target layer untouched. The weightless synapse is unteachable.

Using the OneToOne Synapse

The one to one synapse works very similar to the weightless synapse. Like the weightless synapse, the one to one synapse does not include any weight

values. The primary difference is that every neuron in the source layer is connected to the corresponding neuron in the target layer. Each neuron is connected to only one other neuron. Because of this, the one to one synapse requires that the source and target layers have the same number of neurons. Figure 2.5 shows the one to one synapse.

Figure 2.5: The One to One Synapse

The following code segment shows how to construct a neural network that makes use of a one to one layer. The one to one layer is used in conjunction with a context layer.

```
final Layer context = new ContextLayer(2);
final BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(1));
network.addLayer(hidden = new BasicLayer(2));
hidden.addNext(context, SynapseType.OneToOne);
context.addNext(hidden);
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
```

The one to one synapse is generally used to directly feed the values from the output of a layer to a context layer. However, it can serve any purpose where you would like to send a copy of the output of one layer to another similarly sized layer.

Using the Direct Synapse

The direct synapse is useful when you want to send a complete copy of the output from the source to every neuron in the target. Most layers are not designed to accept an array from every source neuron, so the number of layers that the direct synapse can be used with is limited. Currently, the only

Encog layer type that supports the **DirectSynapse** is the **RadialBasisFunctionLayer** class. Figure 2.6 shows how the direct synapse works.

Figure 2.6: The Direct Synapse

The following code segment shows how to use the **DirectSynapse**.

```
RadialBasisFunctionLayer rbfLayer;
final BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(new ActivationLinear(), false,
2));
network.addLayer(rbfLayer = new RadialBasisFunctionLayer(4),
SynapseType.Direct);
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
network.reset();
rbfLayer.randomizeGaussianCentersAndWidths(0, 1);
```

As you can see, the **DirectSynapse** is being used to feed a **RadialBasisFunctionLayer**.

Understanding Neural Logic

Every Encog neural network must contain a neural logic class. The **NeuralLogic** classes define how a neural network will process its layers and synapses. All neural logic classes must implement the **NeuralLogic** interface. By default a **BasicNetwork** class will make use of the **SimpleRecurrentLogic** logic class. This class can be used for both feedforward and simple recurrent networks. Because these are some of the

most common neural network types in use, the **SimpleRecurrentLogic** class was chosen as the default.

The next few sections summarize the network logic classes provided by Encog.

The ART1Logic Class

The **ART1Logic** class is used to implement an adaptive resonance theory neural network. Adaptive Resonance Theory (ART) is a form of neural network developed by Stephen Grossberg and Gail Carpenter. There are several versions of the ART neural network, which are numbered ART-1, ART-2 and ART-3. The ART neural network is trained using either a supervised or unsupervised learning algorithm, depending on the version of ART being used. ART neural networks are used for pattern recognition and prediction. Encog presently supports ART1.

To create an ART1 neural network with Encog you should make use of the **ART1Logic** class. An example of an ART1 neural network will be provided in Chapter 14, "Common Neural Network Patterns".

The BAMLogic Class

The **BAMLogic** class is used to implement a Bidirectional Associative Memory (BAM) network. The BAM network is a type of neural network developed by Bart Kosko in 1988. The BAM is a recurrent neural network that works similarly to and allows patterns of different lengths to be mapped bidirectionally to other patterns. This allows it to act as almost a two-way hash map. During its training, the BAM network is fed pattern pairs. The two halves of each pattern do not have to be of the same length. However, all patterns must be of the same overall structure. The BAM network can be fed a distorted pattern on either side and will attempt to map to the correct value.

The BoltzmannLogic Class

The **BoltzmannLogic** class is used to implement a Boltzmann machine neural network. A Boltzmann machine is a type of neural network developed by Geoffrey Hinton and Terry Sejnowski. It appears identical to a Hopfield

neural network except it contains a random nature to its output. A temperature value is present that influences the output from the neural network. As this temperature decreases so does the randomness. This is called simulated annealing. Boltzmann networks are usually trained in an unsupervised mode. However, supervised training can be used to refine what the Boltzmann machine recognizes.

To create a Boltzmann machine neural network with Encog you should make use of the **BoltzmannLogic** class. An example of a Boltzmann neural network will be provided in Chapter 12, “Recurrent Neural Networks”.

The FeedforwardLogic Class

To create a feedforward with Encog the **FeedforwardLogic** class should be used. It is also possible to use the **SimpleRecurrentLogic** class as in place of the **FeedforwardLogic** class; however, the network will run slower. If there are no recurrent loops, the more simple **FeedforwardLogic** class should be used.

The feedforward neural network, or perceptron, is a type of neural network first described by Warren McCulloch and Walter Pitts in the 1940s. The feedforward neural network, and its variants, is the most widely used form of neural network. The feedforward neural network is often trained with the backpropagation training technique, though there are other more advanced training techniques, such as resilient propagation. The feedforward neural network uses weighted connections from an input layer to zero or more hidden layers, and finally to an output layer. It is suitable for many types of problems. Feedforward neural networks are used frequently in this book.

The HopfieldLogic Class

To create a Hopfield neural network with Encog, you should use the **HopfieldLogic** class. The Hopfield neural network was developed by Dr. John Hopfield in 1979. The Hopfield network is a single layer recurrent neural network. The Hopfield network always maintains a "current state" which is the current output of the neural network. The Hopfield neural network also has an energy property, which is calculated exactly the same as the temperature property of the Boltzmann machine. The Hopfield network is

trained for several patterns. The state of the Hopfield network will move towards the closest pattern, thus "recognizing" that pattern. As the Hopfield network moves towards one of these patterns, the energy lowers.

To create a Hopfield neural network with Encog you should make use of the **HopfieldLogic** class. An example of a Hopfield neural network will be provided in Chapter 12, “Recurrent Neural Networks”.

The SimpleRecurrentLogic Class

To create a neural network where some layers are connected to context layers that connect back to previous layers, you should use the **SimpleRecurrentLogic** class. The Elman and Jordan neural networks are examples of the sort of networks where the **SimpleRecurrentLogic** class can be used. The **SimpleRecurrentLogic** class can also be used to implement a simple feedforward neural network, however, the **FeedforwardLogic** class will execute faster.

To create either an Elman or Jordan type of neural network with Encog you should make use of the **SimpleRecurrentLogic** class. Several examples of recurrent neural networks will be provided in Chapter 12, “Recurrent Neural Networks”.

The SOMLogic Class

To create a Self Organizing Map with Encog the **SOMLogic** class should be used. The Self Organizing Map (SOM) is a neural network type introduced by Teuvo Kohonen. SOM's are used to classify data into groups.

To create a SOM neural network with Encog you should make use of the **SOMLogic** class. An example of a SOM neural network will be provided in Chapter 9, “Unsupervised Training Methods”.

Understanding Properties and Tags

The **BasicNetwork** class also provides properties and tags to address the unique needs of different neural network logic types. Properties provide a set of name-value pairs that the neural logic can access. This is how you set

properties about how the neural network should function. Tags allow individual layers to be identified. Some of the neural network logic types will affect layers differently. The layer tags allow the neural network logic to know which layer is which.

The following code shows several properties being set for an ART1 network.

```
BasicNetwork network = new BasicNetwork();

network.setProperty(ARTLogic.PROPERTY_A1, 1);
network.setProperty(ARTLogic.PROPERTY_B1, 2);
network.setProperty(ARTLogic.PROPERTY_C1, 3);
network.setProperty(ARTLogic.PROPERTY_D1, 4);
```

The first parameter specifies the name of the property. The neural network logic classes will define constants for properties that they require. The name of the property is a string.

The following code shows two network layers being tagged.

```
network.tagLayer(BasicNetwork.TAG_INPUT, layerF1);
network.tagLayer(BasicNetwork.TAG_OUTPUT, layerF2);
network.tagLayer(ART1Pattern.TAG_F1, layerF1);
network.tagLayer(ART1Pattern.TAG_F2, layerF2);
```

Here multiple tags are being applied to the **layerF1** and **layerF2** layers. One layer can have multiple tags; however, a single tag can only be applied to one layer.

The **BasicNetwork** class does not keep a list of layers. The only way that layers actually "join" the neural network is either by being tagged, or linked through a synapse connection to a layer that is already tagged.

Building with Layers and Synapses

You are now familiar with all of the layer and synapse types supported by Encog. You will now be given a brief introduction to building ANNs with these neural network types. You will see how to construct several neural network types. They will be used to solve problems related to the XOR operator. For now, the XOR operator is a good enough introduction to several

neural network architectures. We will see more interesting examples, as the book progresses. We will begin with the feedforward neural network.

Creating Feedforward Neural Networks

The feedforward neural network is one of the oldest types of neural networks still in common use. The feedforward neural network is also known as the perceptron. The feedforward neural network works by having one or more hidden layers sandwiched between an input and output layer. Figure 2.7 shows an Encog Workbench diagram of a feedforward neural network.

Figure 2.7: The Feedforward Neural Network

Listing 2.3 shows a simple example of a feedforward neural network learning to recognize the XOR operator.

Listing 2.3: Simple XOR Feedforward Neural Network

```
package org.encog.examples.neural.xorresilient;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
```

```
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.
  resilient.ResilientPropagation;
import org.encog.util.logging.Logging;

public class XORResilient {

  public static double XOR_INPUT[][] =
    { { 0.0, 0.0 },
      { 1.0, 0.0 },
       { 0.0, 1.0 },
      { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
      { 0.0 },
      { 1.0 },
      { 1.0 },
      { 0.0 } };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();

    BasicNetwork network = new BasicNetwork();
    network.addLayer(
      new BasicLayer(new ActivationSigmoid(),
      false,2));
    network.addLayer(
      new BasicLayer(new ActivationSigmoid(),
      false,4));
    network.addLayer(new BasicLayer(
      new ActivationSigmoid(), false,1));

    network.getStructure().finalizeStructure();
    network.reset();

    NeuralDataSet trainingSet =
      new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

// train the neural network
    final Train train = new ResilientPropagation(
      network,
      trainingSet);
```

```
    int epoch = 1;

    do {
      train.iteration();
      System.out.println("Epoch #"
        + epoch
        + " Error:"
        + train.getError());
      epoch++;
      } while(train.getError() > 0.01);

// test the neural network
  System.out.println("Neural Network Results:");
  for(NeuralDataPair pair: trainingSet ) {
    final NeuralData output =
      network.compute(pair.getInput());
    System.out.println(
      pair.getInput().getData(0)
      + ","
      + pair.getInput().getData(1)
      + ", actual="
      + output.getData(0)
      + ",ideal="
      + pair.getIdeal().getData(0));
    }
  }
}
```

As you can see from the above listing, it is very easy to construct a three-layer, feedforward neural network. Essentially, three new **BasicLayer** objects are created and added to the neural network with calls to the **addLayer** method. Because no synapse type is specified, the three layers are connected together using the **WeightedSynapse**.

You will notice that after the neural network is constructed, it is trained. There are quite a few ways to train a neural network in Encog. Training is the process where the weights and thresholds are adjusted to values that will produce the desired output from the neural network. This example uses resilient propagation (RPROP) training. RPROP is the best choice for most neural networks to be trained with Encog. For certain special cases, some of the other training types may be more efficient.

```
// train the neural network
final Train train = new ResilientPropagation(network,
trainingSet);
```

With the trainer setup we must now cycle through a bunch of iterations, or epochs. Each of these training iterations should decrease the "error" of the neural network. The error is the difference between the current actual output of the neural network and the desired output.

```
int epoch = 1;
do {
train.iteration();
System.out.println("Epoch #" + epoch + " Error:" +
train.getError());
epoch++;
```

Continue training the neural network so long as the error rate is greater than one percent.

```
} while(train.getError() > 0.01);
```

Now that the neural network has been trained, we should test it. To do this, the same data that the neural network was trained with is presented to the neural network. The following code does this.

```
System.out.println("Neural Network Results:");
for(NeuralDataPair pair: trainingSet ) {
final NeuralData output = network.compute(pair.getInput());

System.out.println(pair.getInput().getData(0) + "," +
pair.getInput().getData(1)                              + ",
actual=" + output.getData(0) + ",ideal=" +
pair.getIdeal().getData(0));
```

This will produce the following output:

```
Epoch #1 Error:0.9902997764512583
Epoch #2 Error:0.6762359214192293
Epoch #3 Error:0.49572129129302844
Epoch #4 Error:0.49279160045197135
Epoch #5 Error:0.5063357328001542
Epoch #6 Error:0.502484567412553
Epoch #7 Error:0.4919515177527043
Epoch #8 Error:0.49157058621332506
Epoch #9 Error:0.48883664423510526
```

```
Epoch #10 Error:0.48977067420698456
Epoch #11 Error:0.4895238942630234
Epoch #12 Error:0.4870271073515729
Epoch #13 Error:0.48534672846811844
Epoch #14 Error:0.4837776485977757
Epoch #15 Error:0.48184530627656685
Epoch #16 Error:0.47980242878514856
Epoch #17 Error:0.47746641141708474
Epoch #18 Error:0.4748474362926616
Epoch #19 Error:0.47162728117571795
Epoch #20 Error:0.46807640808835427
...
Epoch #495 Error:0.010583637636670955
Epoch #496 Error:0.010748859630158925
Epoch #497 Error:0.010342203029249158
Epoch #498 Error:0.00997945501479827
Neural Network Results: 0.0,0.0,
actual=0.005418223644461675,ideal=0.0 1.0,0.0,
actual=0.9873413174817033,ideal=1.0 0.0,1.0,
actual=0.9863636878918781,ideal=1.0 1.0,1.0,
actual=0.007650291171204077,ideal=0.0
```

As you can see the error rate starts off high and steadily decreases. Finally, the patterns are presented to the neural network. As you can see, the neural network can handle the XOR operator. It does not produce the exact output it was trained with, but it is very close. The values 0.0054 and 0.0076 are very close to zero, just as 0.987 and 0.986 are very close to one.

For this network, we are testing the neural network with exactly the same data that the neural network was trained with. Generally, this is a very bad practice. You want to test the neural network on data that it was not trained with. This lets you see how the neural network is performing with new data that it has never processed before. However, the XOR function only has four possible combinations, and they all represent unique patterns that the network must be trained for. Neural networks presented later in this book will not use all of their data for training. Rather, they will be tested on data it has never been presented with before.

Creating Self-Connected Neural Networks

We will now look at self-connected neural networks. The Hopfield neural network is a good example of a self-connected, neural network. The Hopfield neural network contains a single layer of neurons. This layer is connected to

itself. Every neuron on the layer is connected to every other neuron on the same layer. However, no two neurons are connected to themselves. Figure 2.8 shows a Hopfield neural network diagramed in the Encog Workbench.

Figure 2.8: The Hopfield Neural Network

Listing 2.4 shows a simple example of a Hopfield neural network learning to recognize various patterns.

Listing 2.4: Hopfield Neural Network

```
package org.encog.examples.neural.hopfield;

import org.encog.neural.data.bipolar.BiPolarNeuralData;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.logic.HopfieldLogic;
import org.encog.neural.pattern.HopfieldPattern;

public class HopfieldAssociate {

  final static int HEIGHT = 10;
  final static int WIDTH = 10;

  /**
    * The neural network will learn these patterns.
    */
  public static final String[][] PATTERN  = { {
    "O O O O O ",
    " O O O O O",
    "O O O O O ",
    " O O O O O",
    "O O O O O ",
    " O O O O O",
    "O O O O O ",
    " O O O O O",
    "O O O O O ",
    " O O O O O"  },

  { "OO  OO  OO",
    "OO  OO  OO",
```

```
    "  OO  OO  ",
    "  OO  OO  ",
    "OO  OO  OO",
    "OO  OO  OO",
    "  OO  OO  ",
    "  OO  OO  ",
    "OO  OO  OO",
    "OO  OO  OO"  },

  { "OOOOO     ",
    "OOOOO     ",
    "OOOOO     ",
    "OOOOO     ",
    "OOOOO     ",
    "     OOOOO",
    "     OOOOO",
    "     OOOOO",
    "     OOOOO",
    "     OOOOO"  },

  { "O  O  O  O",
    " O  O  O  ",
    "  O  O  O ",
    "O  O  O  O",
    " O  O  O  ",
    "  O  O  O ",
    "O  O  O  O",
    " O  O  O  ",
    "  O  O  O ",
    "O  O  O  O"  },

  { "OOOOOOOOOO",
    "O        O",
    "O OOOOOO O",
    "O O    O O",
    "O O OO O O",
    "O O OO O O",
    "O O    O O",
    "O OOOOOO O",
    "O        O",
    "OOOOOOOOOO"  } };

  /**
    * The neural network will be tested on these
    * patterns, to see
    * which of the last set they are the closest to.
```

```
  */
public static final String[][] PATTERN2 = { {
  "           ",
  "           ",
  "           ",
  "           ",
  "           ",
  " O O O O O",
  "O O O O O ",
  " O O O O O",
  "O O O O O ",
  " O O O O O"  },

{ "OOO O    O",
  " O  OOO OO",
  "  O O OO O",
  " OOO   O  ",
  "OO  O  OOO",
  " O OOO   O",
  "O OO  O  O",
  "   O OOO  ",
  "OO OOO  O ",
  " O  O  OOO"  },

{ "OOOOO     ",
  "O   O OOO ",
  "O   O OOO ",
  "O   O OOO ",
  "OOOOO     ",
  "     OOOOO",
  " OOO O   O",
  " OOO O   O",
  " OOO O   O",
  "     OOOOO"  },

{ "O  OOOO  O",
  "OO  OOOO  ",
  "OOO  OOOO ",
  "OOOO  OOOO",
  " OOOO  OOO",
  "  OOOO  OO",
  "O  OOOO  O",
  "OO  OOOO  ",
  "OOO  OOOO ",
  "OOOO  OOOO"  },
```

```
{ "OOOOOOOOOO",
  "O        O",
  "O        O",
  "O        O",
  "O   OO   O",
  "O   OO   O",
  "O        O",
  "O        O",
  "O        O",
  "OOOOOOOOOO"  } };

public BiPolarNeuralData convertPattern(
  String[][] data, int index) {

  int resultIndex = 0;
  BiPolarNeuralData result =
    new BiPolarNeuralData(WIDTH*HEIGHT);

  for(int row=0;row<HEIGHT;row++) {

    for(int col=0;col<WIDTH;col++) {
      char ch = data[index][row].charAt(col);
      result.setData(resultIndex++, ch=='O');
      }
  }
return result;
}

public void display(BiPolarNeuralData pattern1,
  BiPolarNeuralData pattern2) {

  int index1 = 0;
  int index2 = 0;

  for(int row = 0;row<HEIGHT;row++) {

    StringBuilder line = new StringBuilder();

    for(int col = 0;col<WIDTH;col++) {

      if(pattern1.getBoolean(index1++))
        line.append('O');
      else
        line.append(' ');
      }
```

```
    line.append("   ->   ");

    for(int col = 0;col<WIDTH;col++) {
      if(pattern2.getBoolean(index2++))
        line.append('O');
      else
        line.append(' ');
      }
    System.out.println(line.toString());
    }
  }

  public void evaluate(BasicNetwork hopfield,
    String[][] pattern) {

    HopfieldLogic hopfieldLogic =
      (HopfieldLogic)hopfield.getLogic();
    for(int i=0;i<pattern.length;i++) {

      BiPolarNeuralData pattern1 =
        convertPattern(pattern,i);

      hopfieldLogic.setCurrentState(pattern1);

      int cycles =  hopfieldLogic.runUntilStable(100);
      BiPolarNeuralData pattern2 =
(BiPolarNeuralData)hopfieldLogic.getCurrentState();

      System.out.println(
        "Cycles until stable(max 100): "
        + cycles
        + ", result=");
        display( pattern1, pattern2);
      System.out.println(
        "----------------------");
    }
  }

  public void run()
  {
    HopfieldPattern pattern = new HopfieldPattern();
    pattern.setInputNeurons(WIDTH*HEIGHT);
    BasicNetwork hopfield = pattern.generate();
    HopfieldLogic hopfieldLogic =
    (HopfieldLogic)hopfield.getLogic();
```

```
    for(int i=0;i<PATTERN.length;i++) {
      hopfieldLogic.addPattern(
      convertPattern(PATTERN,i));
    }

    evaluate(hopfield,PATTERN);
    evaluate(hopfield,PATTERN2);
  }

  public static void main(String[] args) {
    HopfieldAssociate program =
      new HopfieldAssociate();
            program.run();
  }

}
```

The Hopfield example begins by creating a **HopfieldPattern** class. The pattern classes allow for common types of neural networks to be constructed automatically. You simply provide the parameters about the type of neural network you wish to create, and the pattern takes care of setting up layers, synapses, parameters and tags.

```
HopfieldPattern pattern = new HopfieldPattern();
```

This Hopfield neural network is going to recognize graphic patterns. These graphic patterns are mapped on to grids. The number of input neurons will be the total number of cells in the grid. This is the width times the height.

```
pattern.setInputNeurons(WIDTH*HEIGHT);
```

The Hopfield pattern requires very little input, just the number of input neurons. Other patterns will require more parameters. Now that the **HopfieldPattern** has been provided with all that it needs, the **generate** method can be called to create the neural network.

```
BasicNetwork hopfield = pattern.generate();
```

The logic object is obtained for the Hopfield network.

```
HopfieldLogic hopfieldLogic = (HopfieldLogic)hopfield.getLogic();
```

The logic class is used to add the patterns that the neural network is to be trained on. This is similar to the training seen in the last section, except it happens much faster for the simple Hopfield neural network.

```
for(int i=0;i<PATTERN.length;i++){
  hopfieldLogic.addPattern(convertPattern(PATTERN,i));
}
```

Now that the network has been "trained" we will test it. Just like in the last section, we will evaluate the neural network with the same data with which it was trained.

```
evaluate(hopfield,PATTERN);
```

However, in addition to the data that the network has already been presented with, we will also present new data. This new data are distorted images of the data that the network was trained on. The network should be able to still recognize the patterns, even though they were distorted.

```
evaluate(hopfield,PATTERN2);
```

The following shows the output of the Hopfield neural network. As you can see the Hopfield neural network is first presented with the patterns that it was trained on. The Hopfield network simply echoes these patterns. Next, the Hopfield neural network is presented with distorted versions of the patterns with which it was trained. As you can see from the code snippet below, the Hopfield neural network still recognizes the values.

```
Cycles until stable(max 100): 1, result=
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
----------------------
Cycles until stable(max 100): 1, result=
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
  OO  OO     ->     OO  OO
```

```
  OO  OO     ->     OO  OO
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
  OO  OO     ->     OO  OO
  OO  OO     ->     OO  OO
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
----------------------
Cycles until stable(max 100): 1, result=
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
----------------------
Cycles until stable(max 100): 1, result=
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
----------------------
Cycles until stable(max 100): 1, result=
OOOOOOOOOO   ->   OOOOOOOOOO
O        O   ->   O        O
O OOOOOO O   ->   O OOOOOO O
O O    O O   ->   O O    O O
O O OO O O   ->   O O OO O O
O O OO O O   ->   O O OO O O
O O    O O   ->   O O    O O
O OOOOOO O   ->   O OOOOOO O
O        O   ->   O        O
OOOOOOOOOO   ->   OOOOOOOOOO
----------------------
Cycles until stable(max 100): 2, result=
             ->   O O O O O
```

```
              ->    O O O O O
              ->   O O O O O
              ->    O O O O O
              ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
----------------------
Cycles until stable(max 100): 2, result=
OOO O    O   ->   OO  OO  OO
 O  OOO OO   ->   OO  OO  OO
  O O OO O   ->     OO  OO
 OOO   O     ->     OO  OO
OO  O  OOO   ->   OO  OO  OO
 O OOO   O   ->   OO  OO  OO
O OO  O  O   ->     OO  OO
   O OOO     ->     OO  OO
OO OOO  O    ->   OO  OO  OO
 O  O  OOO   ->   OO  OO  OO
----------------------
Cycles until stable(max 100): 2, result=
OOOOO        ->   OOOOO
O   O OOO    ->   OOOOO
O   O OOO    ->   OOOOO
O   O OOO    ->   OOOOO
OOOOO        ->   OOOOO
     OOOOO   ->        OOOOO
 OOO O   O   ->        OOOOO
 OOO O   O   ->        OOOOO
 OOO O   O   ->        OOOOO
     OOOOO   ->        OOOOO
----------------------
Cycles until stable(max 100): 2, result=
O  OOOO  O   ->   O  O  O  O
OO  OOOO     ->    O  O  O
OOO  OOOO    ->     O  O  O
OOOO  OOOO   ->   O  O  O  O
 OOOO  OOO   ->    O  O  O
  OOOO  OO   ->     O  O  O
O  OOOO  O   ->   O  O  O  O
OO  OOOO     ->    O  O  O
OOO  OOOO    ->     O  O  O
OOOO  OOOO   ->   O  O  O  O
----------------------
```

```
Cycles until stable(max 100): 2, result=
OOOOOOOOOO   ->   OOOOOOOOOO
O        O   ->   O        O
O        O   ->   O OOOOOO O
O        O   ->   O O    O O
O   OO   O   ->   O O OO O O
O   OO   O   ->   O O OO O O
O        O   ->   O O    O O
O        O   ->   O OOOOOO O
O        O   ->   O        O
OOOOOOOOOO   ->   OOOOOOOOOO
----------------------
```

As you can see, the neural network can recognize the distorted values as well as those values with which it was trained. This is a much more comprehensive test than was performed in the previous section. This is because the network is evaluated with data that it has never seen before.

When the Hopfield neural network recognizes a pattern, it returns the pattern that it was trained with. This is called autoassociation.

The program code for the **evaluate** method will now be examined. This shows how to present a pattern to the neural network.

```
public void evaluate(
   BasicNetwork hopfield, String[][] pattern)
{
```

First the logic object is obtained.

```
  HopfieldLogic hopfieldLogic =
(HopfieldLogic)hopfield.getLogic();
```

Loop over all of the patterns and present each to the neural network.

```
  for(int i=0;i<pattern.length;i++)
  {
    BiPolarNeuralData pattern1 = convertPattern(pattern,i);
```

The pattern is obtained from the array and converted to a form that can be presented to the neural network. The graphic patterns are binary, either the pixel is on or it is off. To convert the image all displayed pixels are converted to the numbers. We are using bipolar numbers, so a display pixel is 1, a hidden pixel is -1.

The Hopfield neural network has a current state. The neurons will be at either 1 or -1 level. The current state of the Hopfield network is set to the pattern that we want to recognize.

```
    hopfieldLogic.setCurrentState(pattern1);
```

The Hopfield network will be run until it stabilizes. A Hopfield network will adjust its pattern until it no longer changes. At this point it has stabilized. The Hopfield neural network will stabilize on one of the patterns that it was trained on. The following code will run the Hopfield network until it stabilizes, up to 100 iterations.

```
    int cycles = hopfieldLogic.runUntilStable(100);
    BiPolarNeuralData pattern2 =
(BiPolarNeuralData)hopfieldLogic.getCurrentState();
```

Once the network's state has stabilized it is displayed.

```
    System.out.println("Cycles until stable(max 100): " + cycles
+ ", result=");
    display( pattern1, pattern2);
    System.out.println("----------------------");
}
```

These are just a few of the neural network types that can be constructed with Encog. As the book progresses, you will learn many more.

Chapter Summary

Encog neural networks are made up of layers, synapses, properties and a neural logic class. This chapter reviewed each of these. A layer is a collection of similar neurons. A synapse connects one layer to another. Properties define unique qualities that one neural network type might have. The neural logic class defines how the output of the neural network should be calculated.

Activation functions are very important to neural networks. Activation functions scale the output from one layer before it reaches the next layer. The next chapter will discuss how Encog makes use of activation functions.

Questions for Review

1. What is the purpose of an Encog layer.

2. What is autoassociation? Wheat Encog neural network type makes use of it.

3. Should the same data that was used to train a neural network be used to evaluate it? Why or why not?

4. What is the role of the neural logic class?

5. What are properties used for in an Encog neural network?

Terms

Activation Function

Adaptive Resonance Theory

Autoassociation

Basic Layer

Bidirectional Associative Memory

Boltzmann Machine

Direct Synapse

Elman Neural Network

Hopfield Neural Network

Hyperbolic Tangent Activation Function

Jordan Neural Network

Layer Tag

Network Pattern

Neural Logic

Neural Network Logic

Neural Network Properties

One-to-One Synapse

Pattern

Radial Basis Activation Function

Radial Basis Function

Radial Basis Function Layer

Self-Connected Layer

Sigmoid Activation Function

Threshold Value

Weight Matrix

Weighted Synapse

Weightless Synapse

Chapter 3: Using Activation Functions

- Activation Functions
- Derivatives and Propagation Training
- Choosing an Activation Function

Activation functions are used by many neural network architectures to scale the output from layers. Encog provides many different activation functions that can be used to construct neural networks. In this chapter you will be introduced to these activation functions.

The Role of Activation Functions

Activation functions are attached to layers. They are used to scale data output from a layer. Encog applies a layer's activation function to the data that the layer is about to output. If you do not specify an activation function for **BasicLayer**, the hyperbolic tangent activation will be the defaulted. The following code creates several **BasicLayer** objects with a default hyperbolic tangent activation function.

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(2));
network.addLayer(new BasicLayer(3));
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
network.reset();
```

If you would like to use an activation function other than the hyperbolic tangent function, use code similar to the following:

```
ActivationSigmoid a = new ActivationSigmoid();
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(a,true,2));
network.addLayer(new BasicLayer(a,true,3));
network.addLayer(new BasicLayer(a,true,1));
network.getStructure().finalizeStructure();
network.reset();
```

The sigmoid tangent activation function is assigned to the variable **a** and passed to each of the **addLayer** calls. The **true** value, that was also introduced, specifies that the **BasicLayer** should also have threshold values.

The ActivationFunction Interface

All classes that are to serve as activation functions must implement the **ActivationFunction** interface. This interface is shown in Listing 3.1.

Listing 3.1: The ActivationFunction Interface

```
public interface ActivationFunction extends EncogPersistedObject
{
  void activationFunction(double[] d);
  void derivativeFunction(double[] d);
  boolean hasDerivative();
}
```

The actual activation function is implemented inside of the **activationFunction** method. The **ActivationSIN** class is a very simple activation function that implements the sine wave. You can see the **activationFunction** implementation below.

```
public void activationFunction(final double[] d) {
  for (int i = 0; i < d.length; i++) {
    d[i] = BoundMath.sin(d[i]);
  }
}
```

As you can see, the activation simply applies the sine function to the array of provided values. This array represents the output neuron values that the activation function is to scale. It is important that the function be given the entire array at once. Some of the activation functions perform operations, such as averaging, that require seeing the entire output array.

You will also notice from the above code that a special class, named **BoundMath**, is used to calculate the sine. This causes "not a number" and "infinity" values to be removed. Sometimes, during training, unusually large or small numbers may be generated. The **BoundMath** class is used to eliminate these values by binding them to either a very large or a very small number. The sine function will not create an out-of-bounds number, and **BoundMath** is used primarily for completeness.

However, we will soon see other functions that could produce out of bound numbers. Exponent and radical functions can be particularly prone to this. Once a "not a number" (NaN) is introduced into the neural network, the

neural network will no longer produce useful results. As a result, bounds checking must be performed.

Derivatives of Activation Functions

If you would like to use propagation training with your activation function, then the activation function must have a derivative. Propagation training will be covered in greater detail in Chapter 5, "Propagation Training". The derivative is calculated by a function named **derivativeFunction.**

```
public void derivativeFunction(final double[] d) {
  for (int i = 0; i < d.length; i++) {
    d[i] = BoundMath.cos(d[i]);
  }
}
```

The **derivativeFunction** works very similar to the **activationFunction**, an array of values is passed in to calculate.

Encog Activation Functions

The next sections will explain each of the activation functions supported by Encog. There are several factors to consider when choosing an activation function. Firstly, the type of neural network you are using may dictate the activation function you must use. Secondly, you should consider if you would like to train the neural network using propagation. Propagation training requires an activation function that provides a derivative. You must also consider the range of numbers you will be dealing with. This is because some activation functions deal with only positive numbers or numbers in a particular range.

ActivationBiPolar

The **ActivationBiPolar** activation function is used with neural networks that require bipolar numbers. Bipolar numbers are either **true** or **false**. A **true** value is represented by a bipolar value of 1; a **false** value is represented by a bipolar value of -1. The bipolar activation function ensures

that any numbers passed to it are either -1 or 1. The **ActivationBiPolar** function does this with the following code:

```
if (d[i] > 0) {
  d[i] = 1;
} else {
  d[i] = -1;
}
```

As you can see the output from this activation is limited to either -1 or 1. This sort of activation function is used with neural networks that require bipolar output from one layer to the next. There is no derivative function for bipolar, so this activation function cannot be used with propagation training.

Activation Competitive

The **ActivationCompetitive** function is used to force only a select group of neurons to win. The winner is the group of neurons that has the highest output. The outputs of each of these neurons are held in the array passed to this function. The size of the winning group of neurons is definable. The function will first determine the winners. All non-winning neurons will be set to zero. The winners will all have the same value, which is an even division of the sum of the winning outputs.

This function begins by creating an array that will track whether each neuron has already been selected as one of the winners. We also count the number of winners so far.

```
final boolean[] winners = new boolean[d.length];
double sumWinners = 0;
```

First, we loop **maxWinners** a number of times to find that number of winners.

```
for (int i = 0; i < this.maxWinners; i++) {
  double maxFound = Double.NEGATIVE_INFINITY;
  int winner = -1;
```

Now, we must find one winner. We will loop over all of the neuron outputs and find the one with the highest output.

```
for (int j = 0; j < d.length; j++) {
```

If this neuron has not already won, and it has the maximum output then it might potentially be a winner, if no other neuron has a higher activation.

```
  if (!winners[j] && (d[j] > maxFound)) {
    winner = j;
    maxFound = d[j];
  }
}
```

Keep the sum of the winners that were found, and mark this neuron as a winner. Marking it a winner will prevent it from being chosen again. The sum of the winning outputs will ultimately be divided among the winners.

```
sumWinners += maxFound;
winners[winner] = true;
```

Now that we have the correct number of winners, we must adjust the values for winners and non-winners. The non-winners will all be set to zero. The winners will share the sum of the values held by all winners.

```
for (int i = 0; i < d.length; i++) {
  if (winners[i]) {
    d[i] = d[i] / sumWinners;
  } else {
    d[i] = 0.0;
}
```

This sort of an activation function can be used with competitive, learning neural networks, such as the Self Organizing Map. This activation function has no derivative, so it cannot be used with propagation training.

ActivationGaussian

The **ActivationGaussian** function is based on the Gaussian function. The Gaussian function produces the familiar bell-shaped curve. The equation for the Gaussian function is shown in Equation 3.1.

Equation 3.1: The Gaussian Function

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

There are three different constants that are fed into the Gaussian function. The constant **a** represents the curve's peak. The constant **b** represents the position of the curve. The constant **c** represents the width of the curve.

Figure 3.1: The Graph of the Gaussian Function

The Gaussian function is implemented in Java as follows.

```
return this.peak
      * BoundMath.exp(-Math.pow(x - this.center, 2)
      / (2.0 * this.width * this.width));
```

The Gaussian activation function is not a commonly used activation function. However, it can be used when finer control is needed over the activation range. The curve can be aligned to somewhat approximate certain functions.

The radial basis function layer provides an even finer degree of control, as it can be used with multiple Gaussian functions. There is a valid derivative of the Gaussian function; therefore, the Gaussian function can be used with propagation training. The radial basis function layer is covered in Chapter 14, "Common Neural Network Patterns".

ActivationLinear

The **ActivationLinear** function is really no activation function at all. It simply implements the linear function. The linear function can be seen in Equation 3.2.

Equation 3.2: The Linear Activation Function

$$f(x) = x$$

The graph of the linear function is a simple line, as seen in Figure 3.2.

Figure 3.2: Graph of the Linear Activation Function

The Java implementation for the linear activation function is very simple. It does nothing. The input is returned as it was passed.

```
public void activationFunction(final double[] d) {
}
```

The linear function is used primarily for specific types of neural networks that have no activation function, such as the self-organizing map. The linear activation function has a constant derivative of one, so it can be used with

propagation training. Linear layers are sometimes used by the output layer of a feedforward neural network trained with propagation.

ActivationLOG

The **ActivationLog** activation function uses an algorithm based on the log function. The following Java code shows how this is calculated.

```
if (d[i] >= 0) {
  d[i] = BoundMath.log(1 + d[i]);
} else {
  d[i] = -BoundMath.log(1 - d[i]);
}
```

This produces a curve similar to the hyperbolic tangent activation function, which will be discussed later in this chapter. You can see the graph for the logarithmic activation function in Figure 3.3.

Figure 3.3: Graph of the Logarithmic Activation Function

The logarithmic activation function can be useful to prevent saturation. A hidden node of a neural network is considered saturated when, on a given set of inputs, the output is approximately 1 or -1 in most cases. This can slow training significantly. This makes the logarithmic activation function a possible choice when training is not successful using the hyperbolic tangent activation function.

As illustrated in Figure 3.3, the logarithmic activation function spans both positive and negative numbers. This means it can be used with neural networks where negative number output is desired. Some activation functions, such as the sigmoid activation function will only produce positive output. The logarithmic activation function does have a derivative, so it can be used with propagation training.

ActivationSigmoid

The **ActivationSigmoid** activation function should only be used when positive number output is expected, because the **ActivationSigmoid** function will only produce positive output. The equation for the **ActivationSigmoid** function can be seen in Equation 3.3.

Equation 3.3: The ActivationSigmoid Function

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

The **ActivationSigmoid** function will move negative numbers into the positive range. This can be seen in Figure 3.4, which shows the graph of the sigmoid function.

Figure 3.4: Graph of the ActivationSigmoid Function

The **ActivationSigmoid** function is a very common choice for feedforward and simple recurrent neural networks. However, you must be sure that the training data does not expect negative output numbers. If negative numbers are required, consider using the hyperbolic tangent activation function.

ActivationSIN

The **ActivationSIN** activation function is based on the sine function. It is not a commonly used activation function. However, it is sometimes useful for certain data that periodically changes over time. The graph for the **ActivationSIN** function is shown in Figure 3.5.

Figure 3.5: Graph of the SIN Activation Function

The **ActivationSIN** function works with both negative and positive values. Additionally, the **ActivationSIN** function has a derivative and can be used with propagation training.

ActivationSoftMax

The **ActivationSoftMax** activation function is an activation that will scale all of the input values so that their sum will equal one. The **ActivationSoftMax** activation function is sometimes used as a hidden layer activation function.

The activation function begins by summing the natural exponent of all of the neuron outputs.

```
double sum = 0;
for (int i = 0; i < d.length; i++) {
  d[i] = BoundMath.exp(d[i]);
  sum += d[i];
}
```

The output from each of the neurons is then scaled according to this sum. This produces outputs that will sum to 1.

```
for (int i = 0; i < d.length; i++) {
  d[i] = d[i] / sum;
}
```

The **ActivationSoftMax** is generally used in the hidden layer of a neural network or a classification neural network.

ActivationTANH

The **ActivationTANH** activation function is an activation function that uses the hyperbolic tangent function. The hyperbolic tangent activation function is probably the most commonly used activation function, as it works with both negative and positive numbers. The hyperbolic tangent function is the default activation function for Encog. The equation for the hyperbolic tangent activation function can be seen in Equation 3.4.

Equation 3.4: The Hyperbolic Tangent Activation Function

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The fact that the hyperbolic tangent activation function accepts both positive and negative numbers can be seen in Figure 3.6, which shows the graph of the hyperbolic tangent function.

Figure 3.6: Graph of the Hyperbolic Tangent Activation Function

The hyperbolic tangent function that you see above calls the natural exponent function twice. This is an expensive function call. Even using Java's new **Math.tanh** is still fairly slow. We really do not need the exact hyperbolic tangent. An approximation will do. The following code does a fast approximation of the hyperbolic tangent function.

```
private double activationFunction(final double d) {
  return -1 + (2/ (1+BoundMath.exp(-2* d ) ) );
}
```

The hyperbolic tangent function is a very common choice for feedforward and simple recurrent neural networks. The hyperbolic tangent function has a derivative, so it can be used with propagation training.

Summary

Encog uses activation functions to scale the output from neural network layers. By default, Encog will use a hyperbolic tangent function, which is a good general purposes activation function. Any class that acts as an activation function must implement the **ActivationFunction** interface. This interface requires the implementation of several methods. First an **activationFunction** method must be created to actually perform the activation function. Secondly, a **derivativeFunction** method should be implemented to return the derivative of the activation function. If there is no way to take a derivative of the activation function, then an error should be thrown. Only activation functions that have a derivative can be used with propagation training.

The **ActivationBiPolar** activation function class is used when your network only accepts bipolar numbers. The **ActivationCompetitive** activation function class is used for competitive neural networks, such as the Self-Organizing Map. The **ActivationGaussian** activation function class is used when you want a Gaussian curve to represent the activation function. The **ActivationLinear** activation function class is used when you want to have no activation function at all. The **ActivationLOG** activation function class works similarly to the **ActivationTANH** activation function class except it will sometimes not saturate as a hidden layer. The **ActivationSigmoid** activation function class is similar to the **ActivationTANH** activation function class, except only positive numbers are returned. The **ActivationSIN** activation class can be used for periodic data. The **ActivationSoftMax** activation function class scales the output so that the sum is one.

Up to this point we have covered all of the major components of neural networks. Layers contain the neurons and threshold values. Synapses connect the layers together. Activation functions sit inside the layers and scale the output. Tags allow special layers to be identified. Properties allow configuration values to be associated with the neural network. The next chapter will introduce the Encog Workbench. The Encog Workbench is a GUI application that lets you build neural networks composed of all of these elements.

Questions for Review

1. When might you choose a sigmoid layer over the hyperbolic tangent layer?

2. What are the ramifications of choosing an activation function that does not have a way to calculate a derivative?

3. Which activation function should be used if you want no activation function at all for your layer?

4. Which activation function produces output that sums to one?

5. When might a logarithmic activation function be chosen over a hyperbolic tangent activation function?

Terms

BiPolar Activation Function

Competitive Activation Function

Derivative

Gaussian Activation Function

Linear Activation Function

LOG Activation Function

Sigmoid Activation Function

SIN Activation Function

SoftMax Activation Function

TANH Activation Function

Chapter 4: Using the Encog Workbench

- Creating a Neural Network
- Creating a Training Set
- Training a Neural Network
- Querying the Neural Network
- Generating Code

An important part of the Encog Framework is the Encog Workbench. The Encog Workbench is a GUI application that can be used to create and edit neural networks. Encog can persist neural networks to .EG files. These files are an XML representation of the neural networks, and other information in which Encog uses to store data. You will learn more about how to use Java to load and save from .EG files in Chapter 7, "Encog Persistence".

The Encog workbench can be downloaded from the following URL:

http://www.encog.org

There are several different ways that the Encog Workbench is packaged. Depending on your computer system, you should choose one of the following:

- **Universal** – Packaged with shell scripts and batch files to launch the workbench under UNIX, Macintosh or Windows.
- **Windows Application** – Packaged with a Windows launcher. Simply double click the application executable and the application will start.
- **Macintosh Application** – Packaged with a Macintosh launcher. Simply double click the application icon and the application will start.

In this chapter I will assume that you are using the Windows Application package of Encog Workbench. The others will all operate very similarly. Once you download the Encog workbench and unzip it to a directory, the directory will look similar to Figure 4.1.

The Encog Workbench was implemented as a Java application. However, it is compatible with the .Net and Silverlight versions of Encog as well. Java was chosen as the language to write the Workbench in due to its ability to run on many different hardware platforms.

Figure 4.1: The Encog Workbench Folder

To launch the Encog workbench double click the "Encog Workbench" icon. This will launch the Encog Workbench application. Once the workbench starts, you will see something similar to what is illustrated in Figure 4.2.

Figure 4.2: The Encog Workbench Application

The Encog Workbench can run a benchmark to determine how fast Encog will run on this machine. This may take several minutes, as it runs Encog through a number of different neural network operations. The benchmark is also a good way to make sure that Encog is functioning properly on a

computer. To run the benchmark, click the "Tools" menu and select "Benchmark Encog". The benchmark will run and display a progress bar. Once the benchmark is done, you will see the final benchmark number. This can be seen in Figure 4.3.

Figure 4.3: Benchmarking Encog

A lower number reflects a better score. The number is the amount of seconds that it took Encog to complete the benchmark tasks. Each part of the benchmark is run multiple times to try to produce consistent benchmark numbers. Encog's use of multicore processors will be reflected in this number. If the computer is already running other processes, this will slow down the benchmark. Because of this, you should not have other applications running while performing a benchmark using the Encog Workbench.

Creating a Neural Network

We will begin by creating a neural network. The Encog Workbench starts with an empty file. Once objects have been added to this empty file, it can be saved to an .EG file. This .EG file can then be loaded by the workbench again or loaded by Java or .Net Encog applications. The .Net and Java versions of Encog read exactly the same type of .EG files.

To create a neural network, select "Create Object on the "Objects menu". A small popup window will appear that asks for the type of object to create.

Choose "Neural Network" to create a new neural network. This will bring up a window that lets you browse the available types of neural networks to create. These are predefined templates for many of the common neural network types supported by Encog. This window can be seen in Figure 4.4.

Figure 4.4: Create a Neural Network

You will notice that the first option is to create an "Empty Neural Network". Any of the neural networks shown here could be created this way. You would simply create an empty network and add the appropriate layers, synapses, tags and properties to create the neural network type you wish to create. However, if you would like to create one of the common neural network types, it is much faster to simply use one of these predefined templates. Choose the "Feedforward Neural Network". You will need to fill in some information about the type of feedforward neural network you would like to create. This dialog box is seen in Figure 4.5.

Figure 4.5: Create a Feedforward Neural Network

We are going to create a simple, neural network that learns the XOR operator. Such a neural network should be created as follows:

- Input Neuron Count: 2
- Hidden Layer 1 Neuron Count: 2
- Output Neuron Count: 1

The two input neurons are necessary because the XOR operator takes two input parameters. The one output neuron is needed because the XOR operator takes one output parameter. This can be seen from the following truth table for the XOR operator.

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

As you can see from the code above, the XOR operator takes two parameters and produces one value. The XOR operator only returns true, or one, when the two input operators are different. This defines the input and output neuron counts.

The hidden layer count is two. The hidden neurons are necessary to assist the neural network in learning the XOR operator. Two is the minimum number of hidden neurons that can be provided for the XOR operator. You may be wondering how we knew to use two. Usually this is something of a

trial and error process. You want to choose the minimum number of hidden neurons that still sufficiently solves the problem.

Encog can help with this trial and error process. This process is called pruning. You will learn about pruning, and other automated techniques for determine good hidden layer counts in Chapter 13, "Pruning and Structuring Networks".

Now that the feedforward neural network has been created, you will see it in the workbench. Figure 4.6 shows the workbench with a neural network added.

Figure 4.6: Neural Network Added

If you double click the feedforward neural network shown in Figure 4.6, it will open. This allows you to see the layers and synapses. Figure 4.7 shows the feedforward neural network that was just created.

Figure 4.7: The Newly Created Neural Network

The above figure shows how neural networks are edited with Encog. You can add additional layers and synapses. You can also edit other aspects of the neural network, such as properties and the type of neural logic that it uses.

Now that the neural network has been created, a training set should be created. The training set will be used to train the neural network.

Creating a Training Set

A training set is a collection of data to be used to train the neural network. There are two types of training sets commonly used with Encog.

- Supervised Training
- Unsupervised Training

Supervised training data has both an input and expected output specified for the neural network. For example, a truth table above could be represented as a training set. There would be four rows, one for each of the combinations fed to the XOR operator. You would have two input columns and one output column. These correspond to the input and output neurons. The training sets are not concerned with hidden layers. Hidden layers are simply present to assist in learning.

Unsupervised training data only has input values. There are no expected outputs. The neural network will train, in an unsupervised way, and determine for itself what the outputs should be. Unsupervised training is often used for classification problems where you want the neural network to group input data.

First, we must create a training set. Select "Create Object" from the "Objects" menu. Select a training set. Once the training set has been created it will be added along with the network that was previously created.

Figure 4.8: The Newly Created Training Set

Double clicking the training set will open it. The training set will open in a spreadsheet style window, as seen in Figure 4.9.

Figure 4.9: Editing the Training Set

Here you can see the training set. By default, Encog creates a training set for XOR. This is just the default. Usually you would now create the desired number of input and output columns. However, because we are training the XOR operator, the data is fine as it is.

Training a Neural Network

Training a neural network is a process where the neural network's weights and thresholds are modified so that the neural network will produce output according to the training data. There are many different ways to train a neural network. The choice of training method will be partially determined by the neural network type you are creating. Not all neural network types work with all training methods.

To train the neural network open it as you did for Figure 4.7. Click the "Train" button at the top of the window. This will display a dialog box that allows you to choose a training method, as seen in Figure 4.10.

Figure 4.10: Choosing a Training Method

Choose the resilient training method, under propagation. This is usually the best training method available for a supervised feedforward neural network. There are several parameters you can set for the resilient training method. For resilient training it is very unlikely that you should ever change any of these options, other than perhaps the desired maximum error, which defaults to 1%. You can see this dialog box in Figure 4.11.

Figure 4.11: Resilient Propagation Training

Selecting OK will open a window that will allow you to monitor the training progress, as seen in Figure 4.12.

Figure 4.12: About to Begin Training

To begin training, click the "Start" button on the training dialog box. The network will begin training. For complex networks, this process can go on for days. This is a very simple network that will finish in several hundred iterations. You will not likely even see the graph begin as the training will complete in a matter of seconds. Once the training is complete, you will see the following screen.

Figure 4.13: Training Complete

The training is complete because the current error fell below the maximum error allowed that was entered in Figure 4.11, which is 1%. Now that the network has been trained it can produce meaningful output when queried. The training finished very quickly. As a result, there were not enough iterations to draw a chart to show the training progress.

Querying the Neural Network

Querying the neural network allows you to specify values for the inputs to the neural network and observe the outputs. To query the neural network, click "Query" at the top of the network editor seen in Figure 4.7. This will open the query window as seen in Figure 4.14.

Figure 4.14: Query the Neural Network

As you can see from the above window, you are allowed to enter two values for the input neurons. When you click "Calculate", the output values will be shown. In the example above two zeros were entered, which resulted in 0.008. This is consistent with the XOR operator, as 0.008 is close to zero. To get a value even closer to zero, train the neural network to a lower error rate.

You can also view the weights and threshold values that were generated by the training. From the network editor, shown in Figure 4.7, right click the synapse and choose "Edit Weight Matrix" from the popup menu. Likewise, you can view the thresholds by right-clicking and choosing "Edit Layer" from the pop-up menu. Figure 4.15 shows the dialog used to edit the layer properties.

Figure 4.15: View Layer Properties

You can also browse available activation functions. If you choose to change the activation function you will see something similar to that shown in Figure 4.16.

Figure 4.16: Edit the Activation Function

In Figure 4.16 you can see that the current activation function is the hyperbolic tangent. The graph for the hyperbolic tangent function is also shown for reference.

Generating Code

The Encog workbench provides two ways that you can make use of your neural network in Java code. First, you can save the neural network and training data to an .EG file. Java applications can then load data from this .EG file. Using .EG files will be covered in much greater detail in Chapter 7, "Encog Persistence".

Another way to generate code is to use the Encog Workbench. The Encog workbench can generate code in the following languages.

- Java
- C#
- VB.Net

Code generation simply generates the code needed to create the neural network only. No code is generated to train or use the neural network. For the generated program to be of any use, you will need to add your own training code. Listing 4.1 shows the generated Java code from the XOR, feedforward neural network.

Listing 4.1: Generated Java Code

```
import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.layers.Layer;

/**
 * Neural Network file generated by Encog.  This file
 * shows just a simple
 * neural network generated for the structure designed
 * in the workbench.
 * Additional code will be needed for training and
 * processing.
 *
 * http://www.encog.org
 *
 */

public class EncogGeneratedClass {

  public static void main(final String args[]) {

    BasicNetwork network = new BasicNetwork();

    Layer inputLayer = new BasicLayer(
      new ActivationSigmoid(),true,2);
    inputLayer.addNext(inputLayer);

    Layer hiddenLayer1 = new BasicLayer(
      new ActivationSigmoid(),true,2);
    inputLayer.addNext(hiddenLayer1);

    Layer outputLayer = new BasicLayer(
      new ActivationSigmoid(),true,1);
    hiddenLayer1.addNext(outputLayer);
    network.tagLayer("INPUT",inputLayer);
    network.tagLayer("OUTPUT",outputLayer);
    network.getStructure().finalizeStructure();
    network.reset();
  }
}
```

The same network could also have been generated in C# or VB.Net.

Summary

In this chapter you saw how to use the Encog Workbench. The Encog Workbench provides a way to edit the .EG files produced by the Encog Framework. There are also templates available to help you quickly create common neural network patterns. There is also a GUI network editor that allows networks to be designed using drag and drop functionality.

The workbench allows training data to be created as well. Training data can be manually entered or imported from a CSV file. Training data includes the input to the neural network, as well as the expected output. Training data that only includes input data will be used in unsupervised training. Training data that includes both input and expected output will be used in supervised training.

The neural network can be trained using many different training algorithms. For a feedforward neural network, one of the best choices is the resilient propagation algorithm. The Encog Workbench allows you to enter parameters for the training, and then watch the progress of the training.

The Encog Workbench will generate the code necessary to produce a neural network that was designed with it. The workbench can generate code in Java, C# or VB.Net. This code shows how to construct the neural network with the necessary layers, synapses, properties and layer tags.

The code generated in this chapter was capable of creating the neural network that was designed in the workbench. However, you needed to add your own training code to make the program functional. The next chapter will introduce some of the ways to train a neural network.

Questions for Review

1. What is the best general-purpose training algorithm, provided by Encog, for a feedforward neural network?

2. What is the difference in the training data used by supervised and unsupervised training?

3. Can both neural networks and training data be stored in an .EG file?

4. Why should training a neural network occur before querying it?

5. How else can you load training into the workbench, other than manually entering it.

Terms

CSV File

Encog Benchmark

Supervised Training

Unsupervised Training

XML File

Chapter 5: Propagation Training

- How Propagation Training Works
- Backpropagation Training
- Manhattan Update Rule
- Resilient Propagation Training

Training is the means by which the weights and threshold values of a neural network are adjusted to give desirable outputs. This book will cover both supervised and unsupervised training. Propagation training is a form of supervised training, where the expected output is given to the training algorithm.

Encog also supports unsupervised training. With unsupervised training you do not provide the neural network with the expected output. Rather, the neural network is left to learn and make insights into the data with limited direction. Chapter 8 will discuss unsupervised training.

Propagation training can be a very effective form of training for feedforward, simple recurrent and other types of neural networks. There are several different forms of propagation training. This chapter will focus on the forms of propagation currently supported by Encog. These three forms are listed as follows:

- Backpropagation Training
- Manhattan Update Rule
- Resilient Propagation Training

All three of these methods work very similarly. However, there are some important differences. In the next section we will explore propagation training in general.

Understanding Propagation Training

Propagation training algorithms use supervised training. This means that the training algorithm is given a training set of inputs and the ideal output for each input. The propagation training algorithm will go through a series of iterations. Each iteration will most likely improve the error rate of the neural network by some degree. The error rate is the percent difference

between the actual output from the neural network and the ideal output provided by the training data.

Each iteration will completely loop through the training data. For each item of training data, some change to the weight matrix and thresholds will be calculated. These changes will be applied in batches. Encog uses batch training. Therefore, Encog updates the weight matrix and threshold values at the end of an iteration.

We will now examine what happens during each training iteration. Each training iteration begins by looping over all of the training elements in the training set. For each of these training elements a two-pass process is executed: a forward pass and a backward pass.

The forward pass simply presents data to the neural network as it normally would if no training had occurred. The input data is presented, and the algorithm calculates the error, which is the difference between the actual output and the ideal output. The output from each of the layers is also kept in this pass. This allows the training algorithms to see the output from each of the neural network layers.

The backward pass starts at the output layer and works its way back to the input layer. The backward pass begins by examining the difference between each of the ideal outputs and the actual output from each of the neurons. The gradient of this error is then calculated. To calculate this gradient, the network the actual output of the neural network is applied to the derivative of the activation function used for this level. This value is then multiplied by the error.

Because the algorithm uses the derivative function of the activation function, propagation training can only be used with activation functions that actually have a derivative function. This derivative is used to calculate the error gradient for each connection in the neural network. How exactly this value is used depends on the training algorithm used.

Understanding Backpropagation

Backpropagation is one of the oldest training methods for feedforward neural networks. Backpropagation uses two parameters in conjunction with the gradient descent calculated in the previous section. The first parameter is the learning rate. The learning rate is essentially a percent that

determines how directly the gradient descent should be applied to the weight matrix and threshold values. The gradient is multiplied by the learning rate and then added to the weight matrix or threshold value. This will slowly optimize the weights to values that will produce a lower error.

One of the problems with the backpropagation algorithm is that the gradient descent algorithm will seek out local minima. These local minima are points of low error, but they may not be a global minimum. The second parameter provided to the backpropagation algorithm seeks to help the backpropagation out of local minima. The second parameter is called momentum. Momentum specifies, to what degree, the weight changes from the previous iteration should be applied to the current iteration.

The momentum parameter is essentially a percent, just like the learning rate. To use momentum, the backpropagation algorithm must keep track of what changes were applied to the weight matrix from the previous iteration. These changes will be reapplied to the current iteration, except scaled by the momentum parameters. Usually the momentum parameter will be less than one, so the weight changes from the previous training iteration are less significant than the changes calculated for the current iteration. For example, setting the momentum to 0.5 would cause fifty percent of the previous training iteration's changes to be applied to the weights for the current weight matrix.

Understanding the Manhattan Update Rule

One of the problems with the backpropagation training algorithm is the degree to which the weights are changed. The gradient descent can often apply too large of a change to the weight matrix. The Manhattan update rule and resilient propagation training algorithms only use the sign of the gradient. The magnitude is discarded. This means it is only important if the gradient is positive, negative or near zero.

For the Manhattan update rule, this magnitude is used to determine how to update the weight matrix or threshold value. If the magnitude is near zero, then no change is made to the weight or threshold value. If the magnitude is positive, then the weight or threshold value is increased by a specific amount. If the magnitude is negative, then the weight or threshold value is decreased by a specific amount. The amount by which the weight or

threshold value is changed is defined as a constant. You must provide this constant to the Manhattan update rule algorithm.

Understanding Resilient Propagation Training

The resilient propagation training (RPROP) algorithm is usually the most efficient training algorithm provided by Encog for supervised feedforward neural networks. One particular advantage to the RPROP algorithm is that it requires no setting of parameters before using it. There are no learning rates, momentum values or update constants that need to be determined. This is good because it can be difficult to determine the exact learning rate that might be optimal.

The RPROP algorithms works similar to the Manhattan update rule, in that only the magnitude of the descent is used. However, rather than using a fixed constant to update the weights and threshold values, a much more granular approach is used. These deltas will not remain fixed, like in the Manhattan update rule or backpropagation algorithm. Rather these delta values will change as training progresses.

The RPROP algorithm does not keep one global update value, or delta. Rather, individual deltas are kept for every threshold and weight matrix value. These deltas are first initialized to a very small number. Every iteration through the RPROP algorithm will update the weight and threshold values according to these delta values. However, as previously mentioned, these delta values do not remain fixed. The gradient is used to determine how they should change, using the magnitude to determine how the deltas should be modified further. This allows every individual threshold and weight matrix value to be individually trained. This is an advantage that is not provided by either the backpropagation algorithm or the Manhattan update rule.

Propagation Training with Encog

Now that you understand the primary differences between the three different types of propagation training used by Encog, we will see how to actually implement each of them. The following sections will show Java examples that make use of all three. The XOR operator, which was

introduced in the last chapter, will be used as an example. The XOR operator is trivial to implement, so it is a good example for a new training algorithm.

Using Backpropagation

In the last chapter we saw how to use the Encog Workbench to implement a solution with the XOR operator using a neural network. In this chapter we will now see how to do this with a Java program. Listing 5.1 shows a simple Java program that will train a neural network to recognize the XOR operator.

Listing 5.1: Using Backpropagation

```
package org.encog.examples.neural.xorbackprop;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.logic.FeedforwardLogic;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation
  .back.Backpropagation;
import org.encog.util.logging.Logging;

public class XorBackprop {

  public static double XOR_INPUT[][] = {
    { 0.0, 0.0 },
    { 1.0, 0.0 },
    { 0.0, 1.0 },
    { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
    { 0.0 },
    { 1.0 },
    { 1.0 },
    { 0.0 } };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();
```

```
  BasicNetwork network = new BasicNetwork();
  network.addLayer(
new BasicLayer(new ActivationSigmoid(),true,2));
  network.addLayer(
new BasicLayer(new ActivationSigmoid(),true,3));
  network.addLayer(
new BasicLayer(new ActivationSigmoid(),true,1));

  network.setLogic(new FeedforwardLogic());
  network.getStructure().finalizeStructure();
  network.reset();

  NeuralDataSet trainingSet =
    new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

  // train the neural network
  final Train train =
    new Backpropagation(network, trainingSet,
      0.7, 0.8);

  int epoch = 1;

  do {
    train.iteration();
    System.out.println(
      "Epoch #"
      + epoch
      + " Error:"
      + train.getError());
    epoch++;
  } while(train.getError() > 0.01);

  // test the neural network
  System.out.println(
    "Neural Network Results:");

  for(NeuralDataPair pair: trainingSet ) {
    final NeuralData output =
       network.compute(pair.getInput());
       System.out.println(
       pair.getInput().getData(0)
          + ","
          + pair.getInput().getData(1)
          + ", actual="
          + output.getData(0)
          + ",ideal="
```

```
            + pair.getIdeal().getData(0));
    }
  }
}
```

We will now examine the parts of the program necessary to implement the XOR backpropagation example.

Truth Table Array

A truth table defines the possible inputs and ideal outputs for a mathematical operator. The truth table for XOR is shown below.

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

The backpropagation XOR example must store the XOR truth table as a 2D array. This will allow a training set to be constructed. We begin by creating **XOR_INPUT**, which will hold the input values for each of the rows in the XOR truth table.

```
public static double XOR_INPUT[][] = {
  { 0.0, 0.0 },
  { 1.0, 0.0 },
  { 0.0, 1.0 },
  { 1.0, 1.0 } };
```

Next we create the array **XOR_IDEAL**, which will hold the expected output for each of the inputs previously defined.

```
public static double XOR_IDEAL[][] = {
  { 0.0 },
  { 1.0 },
  { 1.0 },
  { 0.0 } };
```

You may wonder why it is necessary to use a 2D array for **XOR_IDEAL**. In this case it looks unnecessary, because the XOR neural network has a single output value. However, neural networks can have many output neurons. Because of this, a 2D array is used to allow each row to potentially have multiple outputs.

Constructing the Neural Network

First, the neural network must now be constructed. First we create a **BasicNetwork** class. The **BasicNetwork** class is very extensible. It is currently the only implementation of the more generic **Network** interface needed by Encog.

```
BasicNetwork network = new BasicNetwork();
```

This neural network will have three layers. The input layer will have two input neurons, the output layer will have a single output neuron. There will also be a three neuron hidden layer to assist with processing. All three of these layers can use the **BasicLayer** class. This implements a feedforward neural network, or a multilayer perceptron. Each of these layers makes use of the **ActivationSigmoid** activation function. Sigmoid is a good activation function for XOR because the Sigmoid function only processes positive numbers. Finally, the **true** value specifies that this network should have thresholds.

```
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,2));
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,3));
network.addLayer(new BasicLayer(new ActivationSigmoid(),true,1));
```

The **FeedforwardLogic** class is used to provide the logic for this neural network. The default logic type of **SimpleRecurrentLogic** would have also worked, but **FeedforwardLogic** will provide better performance because there are no recurrent connections in this network.

```
network.setLogic(new FeedforwardLogic());
```

Lastly, the neural network structure is finalized. This builds temporary structures to allow the network to be quickly accessed. It is very important that **finalizeStructure** is always called after the network has been built.

```
network.getStructure().finalizeStructure();
network.reset();
```

Finally, the **reset** method is called to initialize the weights and thresholds to random values. The training algorithm will organize these random values into meaningful weights and thresholds that produce the desired result.

Constructing the Training Set

Now that the network has been created, the training data must be constructed. We already saw the input and ideal arrays created earlier. Now, we must take these arrays and represent them as **NeuralDataSet**. The following code does this.

```
NeuralDataSet trainingSet = new BasicNeuralDataSet(XOR_INPUT,
XOR_IDEAL);
```

A **BasicNeuralDataSet** is used, it is one of several training set types that implement the **NeuralDataSet** interface. Other implementations of **NeuralDataSet** can pull data from a variety of abstract sources, such as SQL, HTTP or image files.

Training the Neural Network

We now have a **BasicNetwork** object and a **NeuralDataSet** object. This is all that is needed to train a neural network. To implement backpropagation training we instantiate a **Backpropagation** object, as follows.

```
final Train train = new Backpropagation(network, trainingSet,
0.7, 0.8);
```

As previously discussed, backpropagation training makes use of a learning rate and a momentum. The value 0.7 is used for the learning rate, the value 0.8 is used for the momentum. Picking proper values for the learning rate and momentum is something of a trial and error process. Too high of a learning rate and the network will no longer decrease its error rate. Too low of a learning rate will take too long to train. If the error rate refuses to lower, even with a lower learning rate, the momentum should be increased to help the neural network get out of a local minimum.

Propagation training is very much an iterative process. The **iteration** method is called over and over; each time the network is slightly adjusted for a better error rate. The following loop will loop and train the neural network until the error rate has fallen below one percent.

```
do {
  train.iteration();
```

```
  System.out.println("Epoch #"
  + epoch + " Error:" + train.getError());

  epoch++;
} while (train.getError() > 0.01);
```

Each trip through the loop is called an epoch, or an iteration. The error rate is the amount that the actual output from the neural network differs from the ideal output provided to the training set.

Evaluating the Neural Network

Now that the neural network has been trained, it should be executed to see how well it functions. We begin by displaying a heading as follows:.

```
System.out.println("Neural Network Results:");
```

We will now loop through each of the training set elements. A **NeuralDataSet** is made up of a collection of **NeuralDataPair** classes. Each **NeuralDataPair** class contains an input and an ideal property. Each of these two properties is a **NeuralData** object that essentially contains an array. This is how Encog stores the training data. We begin by looping over all of the **NeuralDataPair** objects contained in the **NeuralDataSet** object.

```
for(NeuralDataPair pair: trainingSet ) {
```

For each of the **NeuralDataPair** objects, we compute the neural network's output using the input property of the **NeuralDataPair** object.

```
  final NeuralData output = network.compute(pair.getInput());
```

We now display the ideal output, as well as the actual output for the neural network.

```
  System.out.println(pair.getInput().getData(0) + "," +
pair.getInput().getData(1)
  + ", actual=" + output.getData(0) + ",ideal=" +
pair.getIdeal().getData(0));
}
```

The output from this neural network is shown here.

```
Epoch #1 Error:0.504998283847474
Epoch #2 Error:0.504948046227928
Epoch #3 Error:0.5028968616826613
Epoch #4 Error:0.5034596686580215
Epoch #5 Error:0.5042340438643891
Epoch #6 Error:0.5034282078077391
Epoch #7 Error:0.501995999394481
Epoch #8 Error:0.5014532303103851
Epoch #9 Error:0.5016773751196401
Epoch #10 Error:0.5016348354128658
...
Epoch #3340 Error:0.01000800225100623
Epoch #3341 Error:0.010006374293649473
Epoch #3342 Error:0.01000474710532496
Epoch #3343 Error:0.010003120685432222
Epoch #3344 Error:0.010001495033371149
Epoch #3345 Error:0.009999870148542572
Neural Network Results:
0.0,0.0, actual=0.010977229866756838,ideal=0.0
1.0,0.0, actual=0.9905671966735671,ideal=1.0
0.0,1.0, actual=0.989931152973507,ideal=1.0
1.0,1.0, actual=0.009434016119752921,ideal=0.0
```

First, you will see the training epochs counting upwards and decreasing the error. The error starts out at 0.50, which is just above 50%. At epoch 3,345, the error has dropped below one percent and training can stop.

The program then evaluates the neural network by cycling through the training data and presenting each training element to the neural network. You will notice from the above data that the results do not exactly match the ideal results. For instance the value 0.0109 does not exactly match 0.0. However, it is close. Remember that the network was only trained to a one percent error. As a result, the data is not going to match precisely.

In this example, we are evaluating the neural network with the very data that it was trained with. This is fine for a simple example, where we only have four training elements. However, you will usually want to hold back some of your data to with which to validate the neural network. Validating the network with the same data that it was trained with does not prove much. However, validating good results with data other than what the

neural network was trained with proves that the neural network has gained some sort of an insight into the data that it is processing.

Something else that is interesting to note is the number of iterations it took to get an acceptable error. Backpropagation took 3,345 iterations to get to an acceptable error. Different runs of this example produce different results, as we are starting from randomly generated weights and thresholds. However, the number 3,345 is a fairly good indication of the efficiency of the backpropagation algorithm. This number will be compared to the other propagation training algorithms.

Using the Manhattan Update Rule

Next, we will look at how to implement the Manhattan update rule. There are very few changes that are needed to the backpropagation example to cause it to use the Manhattan update rule. Listing 5.2 shows the complete Manhattan update rule example.

Listing 5.2: Using the Manhattan Update Rule

```
package org.encog.examples.neural.xormanhattan;

import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation
  .manhattan.ManhattanPropagation;
import org.encog.util.logging.Logging;

public class XORManhattan {

  public static double XOR_INPUT[][] = {
    { 0.0, 0.0 },
    { 1.0, 0.0 },
    { 0.0, 1.0 },
    { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
    { 0.0 },
    { 1.0 },
```

```
    { 1.0 },
    { 0.0 }
  };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();

    final BasicNetwork network = new BasicNetwork();
    network.addLayer(new BasicLayer(2));
    network.addLayer(new BasicLayer(3));
    network.addLayer(new BasicLayer(1));
    network.getStructure().finalizeStructure();
    network.reset();

    final NeuralDataSet trainingSet =
      new BasicNeuralDataSet(
      XORManhattan.XOR_INPUT, XORManhattan.XOR_IDEAL);

    // train the neural network
    final Train train =
       new ManhattanPropagation(network, trainingSet,
       0.0001);

    int epoch = 1;

    do {
      train.iteration();
      System.out.println("Epoch #"
        + epoch
        + " Error:"
        + train.getError());
        epoch++;
    } while (train.getError() > 0.01);

    // test the neural network
    System.out.println("Neural Network Results:");
    for (final NeuralDataPair pair : trainingSet) {
      final NeuralData output =
        network.compute(pair.getInput());

      System.out.println(pair.getInput().getData(0)
        + ","
        + pair.getInput().getData(1)
        + ", actual="
        + output.getData(0) + ",ideal="
```

```
        + pair.getIdeal().getData(0));
      }
    }
}
```

There is really only one line that has changed from the backpropagation example. Because the **ManhattanPropagation** object uses the same **Train** interface, there are very few changes needed. We simply create a **ManhattanPropagation** object in place of the **Backpropagation** class that was used in the previous section.

```
final Train train = new ManhattanPropagation(network,
trainingSet, 0.0001);
```

As previously discussed, the Manhattan update rule works by using a single constant value to adjust the weights and thresholds. This is usually a very small number so as not to introduce rapid of change into the network. For this example, the number 0.0001 was chosen. Picking this number usually comes down to trial and error, as was the case with backpropagation. A value that is too high causes the network to change randomly and never converge to a number.

The Manhattan update rule will tend to behave somewhat randomly at first. The error rate will seem to improve and then worsen. But it will gradually trend lower. After 710,954 iterations the error rate is acceptable.

```
Epoch #710941 Error:0.011714647667850289
Epoch #710942 Error:0.011573263349587842
Epoch #710943 Error:0.011431878106128258
Epoch #710944 Error:0.011290491948778713
Epoch #710945 Error:0.011149104888883382
Epoch #710946 Error:0.011007716937768005
Epoch #710947 Error:0.010866328106765183
Epoch #710948 Error:0.010724938407208937
Epoch #710949 Error:0.010583547850435736
Epoch #710950 Error:0.010442156447783919
Epoch #710951 Error:0.010300764210593727
Epoch #710952 Error:0.01015937115020837
Epoch #710953 Error:0.010017977277972472
Epoch #710954 Error:0.009876582605234318
Neural Network Results:
0.0,0.0, actual=-0.013777528025884167,ideal=0.0
```

```
1.0,0.0, actual=0.9999999999999925,ideal=1.0
0.0,1.0, actual=0.9999961061923577,ideal=1.0
1.0,1.0, actual=-0.013757731687977337,ideal=0.0
```

As you can see the Manhattan update rule took considerably more iterations to find a solution than the backpropagation. There are certain cases where the Manhattan rule is preferable to backpropagation training. However, for a simple case like the XOR problem, backpropagation is a better solution than the Manhattan rule. Finding a better delta value may improve the efficiency of the Manhattan update rule.

Using Resilient Propagation

One of the most difficult aspects of the backpropagation and the Manhattan update rule learning is picking the correct training parameters. If a bad choice is made for the learning rate, training momentum or delta values will not be as successful as it might have been. Resilient propagation does have training parameters, but it is extremely rare that they need to be changed from their default values. This makes resilient propagation a very easy way to use a training algorithm. Listing 5.3 shows an XOR example using the resilient propagation algorithm.

Listing 5.3: Using Resilient Propagation

```
package org.encog.examples.neural.xorresilient;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.
  resilient.ResilientPropagation;
import org.encog.util.logging.Logging;

public class XORResilient {

  public static double XOR_INPUT[][] =
    { { 0.0, 0.0 },
      { 1.0, 0.0 },
      { 0.0, 1.0 },
```

```
      { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
      { 0.0 },
      { 1.0 },
      { 1.0 },
      { 0.0 } };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();

    BasicNetwork network = new BasicNetwork();
    network.addLayer(
      new BasicLayer(new ActivationSigmoid(), false,2));
    network.addLayer(
      new BasicLayer(new ActivationSigmoid(), false,4));
    network.addLayer(new BasicLayer(
      new ActivationSigmoid(), false,1));

    network.getStructure().finalizeStructure();
    network.reset();

    NeuralDataSet trainingSet =
      new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

// train the neural network
    final Train train = new ResilientPropagation(
      network,
      trainingSet);


    int epoch = 1;

    do {
      train.iteration();
      System.out.println("Epoch #"
        + epoch
        + " Error:"
        + train.getError());
      epoch++;
      } while(train.getError() > 0.01);

// test the neural network
  System.out.println("Neural Network Results:");
  for(NeuralDataPair pair: trainingSet ) {
```

```
    final NeuralData output =
      network.compute(pair.getInput());
    System.out.println(
      pair.getInput().getData(0)
      + ","
      + pair.getInput().getData(1)
      + ", actual="
      + output.getData(0)
      + ",ideal="
      + pair.getIdeal().getData(0));
    }
  }
}
```

The following line of code creates a **ResilientPropagation** object that will be used to train the neural network.

```
final Train train = new ResilientPropagation(network,
trainingSet);
```

As you can see there are no training parameters provided to the **ResilientPropagation** object. Running this example program will produce the following results.

```
Epoch #1 Error:0.5108505683309112
Epoch #2 Error:0.5207537811846186
Epoch #3 Error:0.5087933421445957
Epoch #4 Error:0.5013907858935785
Epoch #5 Error:0.5013907858935785
Epoch #6 Error:0.5000489677062201
Epoch #7 Error:0.49941437656150733
Epoch #8 Error:0.49798185395576444
Epoch #9 Error:0.4980795840636415
Epoch #10 Error:0.4973134271412919
...
Epoch #270 Error:0.010865894525995278
Epoch #271 Error:0.010018272841993655
Epoch #272 Error:0.010068462218315439
Epoch #273 Error:0.009971267210982099
Neural Network Results:
0.0,0.0, actual=0.00426845952539745,ideal=0.0
1.0,0.0, actual=0.9849930511468161,ideal=1.0
0.0,1.0, actual=0.9874048605752819,ideal=1.0
1.0,1.0, actual=0.0029321659866812233,ideal=0.0
```

Not only is the resilient propagation algorithm easier to use, it is also considerably more efficient than backpropagation or the Manhattan update rule.

Propagation and Multithreading

As of the writing of this book, single core computers are becoming much less common than multi core computers. A dual core computer effectively has two complete processors in a single chip. Quadcore computers have four processors on a single chip. The latest generation of Quadcores, the Intel i7, comes with hyperthreading as well. Hyperthreading allows one core processor to appear as two by simultaneously executing multiple instructions. A computer that uses hyperthreading technology will actually report twice the number of cores that is actually installed.

Processors seem to have maxed out their speeds at around 3 gigahertz. Growth in computing power will not be in the processing speed of individual processors. Rather, future growth will be in the number of cores a computer has. However, taking advantage of these additional cores can be a challenge for the computer programmer. To take advantage of these cores you must write multithreaded software.

Entire books are written on multithreaded programming, so it will not be covered in depth here. However, the general idea is to take a large problem and break it down into manageable pieces that be executed independently by multiple threads. The final solution must then be pieced back together from each of the threads. This process is called aggregation.

Encog makes use of multithreading in many key areas. One such area is training. By default the propagation training techniques will use multithreading if it appears that multithreading will help performance. Specifically, there should be more than one core and sufficient training data for multithreading to be worthwhile. If both of these elements are present, any of the propagation techniques will make use of multithreading.

It is possible to tell Encog to use a specific number of threads, or disable threading completely. The **setNumThreads** method provided by all of the propagation training algorithms does this. To run in single threaded mode, specify one thread. To specify a specific number of threads specify the number of threads desired. Finally, to allow Encog to determine the optimal

number of threads, specify zero threads. Zero is the default value for the number of threads.

When Encog is requested to determine the optimal number of threads to use, several things are considered. Encog considers the number of cores that are available. Encog also considers the size of the training data. Multithreaded training works best with larger training sets.

How Multithreaded Training Works

Multithreaded training works particularly well with larger training sets and machines multiple cores. If Encog does not detect that both are present, it will fall back to single threaded. When there is more than one processing core, and enough training set items to keep both cores busy, multithreaded training will function significantly faster than single threaded.

We've already looked at three propagation training techniques. All propagation training techniques work similarly. Whether it is backpropagation, resilient propagation or the Manhattan update rule, the technique is similar. There are two three distinct steps:

1. Perform a Regular Feed Forward Pass.

2. Process the levels backwards, and determine the errors at each level.

3. Apply the changes to the weights and thresholds.

First, a regular feed forward pass is performed. The output from each level is kept so the error for each level can be evaluated independently. Second, the errors are calculated at each level, and the derivatives of each of the activation functions are used to calculate gradient descents. These gradients show the direction that the weight must be modified to improve the error of the network. These gradients will be used in the third step.

The third step is what varies among the different training algorithms. Backpropagation simply takes the gradient descents and scales them by a learning rate. The scaled gradient descents are then directly applied to the weights and thresholds. The Manhattan Update Rule only uses the sign of the gradient to decide in which direction to affect the weight. The weight is then changed in either the positive or negative direction by a fixed constant.

RPROP keeps an individual delta value for every weight and thresholds and only uses the sign of the gradient descent to increase or decrease the delta amounts. The delta amounts are then applied to the weights and thresholds.

The multithreaded algorithm uses threads to perform Steps 1 and 2. The training data is broken into packets that are distributed among the threads. At the beginning of each iteration, threads are started to handle each of these packets. Once all threads have completed, a single thread aggregates all of the results from the threads and applies them to the neural network. There is a very brief amount of time where only one thread is executing, at the end of the iteration. This can be seen from Figure 5.1.

Figure 5.1: Encog Training on a Hyperthreaded Quadcore

As you can see from the above image, the i7 is currently running at 100%. You can clearly see the end of each iteration, where each of the processors falls briefly. Fortunately, this is a very brief time, and does not have a large impact on overall training efficiency. I did try implementations where I did not force the threads to wait at the end of the iteration for a resynchronization. However, these did not provide efficient training because the propagation training algorithms need all changes applied before the next iteration begins.

Using Multithreaded Training

To see multithreaded training really shine, a larger training set is needed. In the next chapter we will see how to gather information for Encog, and larger training sets will be used. However, for now, we will look a simple benchmarking example that generates a random training set and compares multithreaded and single-threaded training times.

A simple benchmark is shown that makes use of an input layer of 40 neurons, a hidden layer of 60 neurons, and an output layer of 20 neurons. A training set of 50,000 elements is used. This example is shown in Listing 5.4.

Listing 5.4: Using Multithreaded Training

```
package org.encog.examples.neural.benchmark;

import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.propagation
  .resilient.ResilientPropagation;
import org.encog.util.benchmark.RandomTrainingFactory;
import org.encog.util.logging.Logging;

public class MultiBench {

  public static final int INPUT_COUNT = 40;
  public static final int HIDDEN_COUNT = 60;
  public static final int OUTPUT_COUNT = 20;

  public static BasicNetwork generateNetwork() {
    final BasicNetwork network = new BasicNetwork();
    network.addLayer(
      new BasicLayer(MultiBench.INPUT_COUNT));
    network.addLayer(
      new BasicLayer(MultiBench.HIDDEN_COUNT));
    network.addLayer(
      new BasicLayer(MultiBench.OUTPUT_COUNT));
    network.getStructure().finalizeStructure();
    network.reset();
    return network;
  }

  public static NeuralDataSet generateTraining() {
    final NeuralDataSet training =
```

```
      RandomTrainingFactory.generate(50000,
      INPUT_COUNT, OUTPUT_COUNT, -1, 1);
    return training;
  }

  public static double evaluateRPROP(
    BasicNetwork network, NeuralDataSet data) {

    ResilientPropagation train =
      new ResilientPropagation(network, data);
    train.setNumThreads(1);
    long start = System.currentTimeMillis();
    System.out.println(
"Training 20 Iterations with Single-threaded");

    for (int i = 1; i <= 20; i++) {
      train.iteration();
      System.out.println("Iteration #" + i
        + " Error:" + train.getError());
    }

    train.finishTraining();
    long stop = System.currentTimeMillis();
    double diff = ((double) (stop - start)) / 1000.0;
    System.out.println(
      "RPROP Result:" + diff + " seconds.");
    System.out.println("Final RPROP error: "
      + network.calculateError(data));
      return diff;
  }

  public static double evaluateMPROP(
    BasicNetwork network, NeuralDataSet data) {

    ResilientPropagation train =
      new ResilientPropagation(network, data);
    train.setNumThreads(0);
    long start = System.currentTimeMillis();
    System.out.println(
      "Training 20 Iterations with Multithreading");
    for (int i = 1; i <= 20; i++) {
      train.iteration();
      System.out.println(
   "Iteration #" + i + " Error:" + train.getError());
    }
    train.finishTraining();
```

```
    long stop = System.currentTimeMillis();
    double diff = ((double) (stop - start)) / 1000.0;
    System.out.println("Multi-Threaded Result:"
       + diff + " seconds.");

    System.out.println(
      "Final Multi-thread error: " +
       network.calculateError(data));
    return diff;
  }

  public static void main(String args[]) {
    Logging.stopConsoleLogging();
    BasicNetwork network = generateNetwork();
    NeuralDataSet data = generateTraining();

    double rprop = evaluateRPROP(network, data);
    double mprop = evaluateMPROP(network, data);
    double factor = rprop / mprop;
    System.out.println("Factor improvement:"
      + factor);
  }
}
```

I executed this program on a Quadcore i7 with Hyperthreading. The following was the result.

```
Training 20 Iterations with Single-threaded
Iteration #1 Error:1.0594453784075148
Iteration #2 Error:1.0594453784075148
Iteration #3 Error:1.0059791059086385
Iteration #4 Error:0.955845375587124
Iteration #5 Error:0.934169803870454
Iteration #6 Error:0.9140418793336804
Iteration #7 Error:0.8950880473422747
Iteration #8 Error:0.8759150228219456
Iteration #9 Error:0.8596693523930371
Iteration #10 Error:0.843578483629412
Iteration #11 Error:0.8239688415389107
Iteration #12 Error:0.8076160458145523
Iteration #13 Error:0.7928442431442133
Iteration #14 Error:0.7772585699972144
Iteration #15 Error:0.7634533283610793
Iteration #16 Error:0.7500401666509937
```

```
Iteration #17 Error:0.7376158116045242
Iteration #18 Error:0.7268954113068246
Iteration #19 Error:0.7155784667628093
Iteration #20 Error:0.705537166118038
RPROP Result:35.134 seconds.
Final RPROP error: 0.6952141684716632
Training 20 Iterations with Multithreading
Iteration #1 Error:0.6952126315707992
Iteration #2 Error:0.6952126315707992
Iteration #3 Error:0.90915249248788
Iteration #4 Error:0.8797061675258835
Iteration #5 Error:0.8561169673033431
Iteration #6 Error:0.7909509694056177
Iteration #7 Error:0.7709539415065737
Iteration #8 Error:0.7541971172618358
Iteration #9 Error:0.7287094412886507
Iteration #10 Error:0.715814914438935
Iteration #11 Error:0.7037730808705016
Iteration #12 Error:0.6925902585055886
Iteration #13 Error:0.6784038181007823
Iteration #14 Error:0.6673310323078667
Iteration #15 Error:0.6585209150749294
Iteration #16 Error:0.6503710867148986
Iteration #17 Error:0.6429473784897797
Iteration #18 Error:0.6370962075614478
Iteration #19 Error:0.6314478792705961
Iteration #20 Error:0.6265724296587237
Multi-Threaded Result:8.793 seconds.
Final Multi-thread error: 0.6219704300851074
Factor improvement:4.0106783805299674
```

As you can see from the above results, the single threaded RPROP algorithm finished in 128 seconds, the multithreaded RPROP algorithm finished in only 31 seconds. Multithreading improved performance by a factor of four. Your results running the above example will depend on how many cores your computer has. If your computer is single core, with no hyperthreading, then the factor will be close to one. This is because the second multi-threading training will fall back to a single thread.

Summary

In this chapter you saw how to use three different propagation algorithms with Encog. Propagation training is a very common class of supervised training algorithms. In this chapter you saw how to use three different propagation training algorithms. Resilient propagation training is usually the best choice; however; the Manhattan update rule and backpropagation may be useful for certain situations.

Backpropagation was one of the original training algorithms for feedforward neural networks. Though Encog supports it mostly for historic purposes, it can sometimes be used to further refine a neural network after resilient propagation has been used. Backpropagation uses a learning rate and momentum. The learning rate defines how quickly the neural network will learn; the momentum helps the network get out of local minima.

The Manhattan update rule uses a delta value to change update the weight and threshold values. It can be difficult to choose this delta value correctly. Too high of a value will cause the network to learn nothing at all.

Resilient propagation (RPROP) is one of the best training algorithms offered by Encog. It does not require you to provide training parameters, like the other two propagation training algorithms. This makes it much easier to use. Additionally, resilient propagation is considerably more efficient than Manhattan update rule or backpropagation.

Multithreaded training is a training technique that adapts propagation training to perform faster with multicore computers. Given a computer with multiple cores and a large enough training set, multithreaded training is considerably faster than single-threaded training. Encog can automatically set an optimal number of threads. If these conditions are not present, Encog will fall back to single threaded training.

Propagation training is not the only type of supervised training that can be used with Encog. In the next chapter we will see some other types of training algorithms that can be used for supervised training. You will see how training techniques such as simulated annealing and genetic algorithms can be used.

Questions for Review

1. What is the primary difference in the way backpropagation and the Manhattan update rule function?

2. What training parameters must be provided to the backpropagation algorithm?

3. What training parameters must be provided to the Manhattan update rule?

4. What is the difference between learning rate and momentum?

5. What is the "error rate" for a neural network using supervised training?

Terms

Backpropagation

Backward Pass

Batch Training

Epoch

Error Rate

Forward Pass

Gradient

Ideal Output

Iteration

Learning rate

Manhattan Update Rule

Momentum

Multicore

Multithreaded

Propagation Training

Resilient Propagation

Single Threaded

Update Delta

Chapter 6: Obtaining Data for Encog

- Finding Data for Neural Networks
- Why Normalize?
- Specifying Normalization Sources
- Specifying Normalization Targets
- Managing Long Training Times

Neural networks can provide profound insights into the data supplied to them. However, you can't just feed any sort of data directly into a neural network. This "raw" data must usually be normalized into a form that the neural network can process. This chapter will show how to normalize "raw" data for use by Encog.

Before we can normalize data, we must first have data. Once you decide what you would like your neural network to do, you must find data so that you can teach the neural network how to perform a task. Fortunately, the Internet provides a wealth of information that can be used with neural networks.

Where to Get Data for Neural Networks

The Internet can be a great source of data for the neural network. There are many sources of data available on the Internet. Data found on the Internet can be in many different formats. One of the most convenient formats for data is the comma-separated value (CSV) format. Other times it may be necessary to create a spider or bot to obtain this data.

One very useful source for neural network data is called Data.gov. This is a site maintained by the United States Government. This site acts as a repository for a great deal of statistical data. It can be accessed from the following URL.

http://www.data.gov/

Another useful site is the Knowledge Discovery site, which is run by the University of California at Irvine.

http://kdd.ics.uci.edu/

The Knowledge Discovery site is a repository of various datasets that have been donated to the University of California. One of these datasets will be used for this chapter's example.

What is Normalization?

Data obtained from sites, such as those listed above, often cannot be directly fed into neural networks. Neural networks can be very "intelligent", but you cannot simply feed any sort of data into a neural network and expect a meaningful result. Often the data must first be normalized. We will begin by looking at what normalization is.

Neural networks are designed to accept floating-point numbers as their input. Usually these input numbers should be in either the range of -1 to +1 or 0 to +1 for maximum efficiency. Your choice of which range is often dictated by your choice of activation function, as certain activation functions have a positive range and others have both a negative and positive range. The sigmoid activation function, for example, has a range of only positive numbers, whereas the hyperbolic tangent activation function has a range of positive and negative numbers.

Normalizing Numeric Values

Numeric data is a very commonly used as both the input and output data. By numeric data I mean integer or floating point numbers. The values of these numbers have meaning as numbers. For example, it is significant that input "a" is larger than input "b". Examples of seemingly numeric values that do not have meaning are US zip codes. The fact that zip code 63123 is larger than 63121 is meaningless. The zip codes are not numeric values, they are nominal values. Nominal values are normalized differently than numeric values. The process for normalizing nominal values is covered in the next section.

In this chapter we will see how to normalize real world data for Encog. We will examine data collected by the United States Forestry Service. This data provides statistical information for a large number of small areas of forest. We will attempt to create a neural network to analyze the statistics about an area of forest and to predict the type of tree cover that area has.

There are several numeric values provided for each area of the forest that was sampled. One of these numeric values is elevation. Elevation is defiantly a numeric value. Consider whether "point a" is at 1,000 meters, and "point b" is at 2,000 meters. The fact that "point b" is higher than "point a" is quite significant. The difference between these two values is also quite significant. Altitude is an example of a numeric value.

Encog normalizes numeric values by either encoding or mapping the input values. The simplest form of numeric normalization used by Encog is encoding. Encoding allows you to specify numeric ranges that should be mapped to a specific value. For example, you could specify that every number between 1 and 1,000 should be mapped to 0.1. Additionally, every number between 1,001 and 2,000 should be mapped to 0.2. You can provide as many of these mappings as needed. The **OutputFieldEncode** class handles this sort of normalization.

Mapping is a slightly more complex way of normalizing numeric values. Mapping allows you to map one numeric range to another. Equation 6.1 shows this.

Equation 6.1: Normalizing Numeric Values

$$f(x) = \left(\left(\frac{x - min}{max - min}\right) \cdot (high - low)\right) + low$$

Where:

- x = The value to normalize
- min = The minimum value x will ever reach
- max = The maximum value that x will ever reach
- low = The low value of the range to normalize into (typically -1 or 0)
- high = The high value of the range to normalize into (typically 0 or 1)

As you can see from the above variables we must know the minimum and maximum values that the data will reach. If mapped normalization is used, Encog must make two passes over the input data. The first pass will collect the maximum and minimum values. The second pass will actually normalize the input values.

Normalizing Nominal Values

Nominal values are used to name things. One very common example of a simple nominal value is gender. Something is either male or female. Another is any sort of boolean question. Nominal values also include values that are either "yes/true" or "no/false". However, not all nominal value have only two values.

Nominal values can also be used to describe an attribute of something, such as color. Neural networks deal best with nominal values where the set is fixed. One nominal variable that will be used later in this chapter is "soil type"; we will have 40 different soil types that the neural network will use to determine the type of tree that would likely grow there. The "tree type" is also a nominal variable. However, both sizes are fixed. We only deal with 40 different soil types and seven different tree types.

Nominal values are used both for neural network input and output. When used with neural network input, the nominal value is describes an attribute of whatever you are trying to recognize. An example of this is the soil type. When used with neural network output, nominal values allow the neural network to communicate what something is. An example of this is the type of tree that would grow on the soil type specified by the neural network input.

Encog supports two different ways to encode nominal values. The simplest means of representing nominal values is called "one-of-n" encoding. One-of-n encoding can often be hard to train, especially if there are more than a few nominal types that you are trying to encode. Equilateral encoding is usually a better choice than the simpler "one-of-n" encoding. Both encoding types will be explored in the next two sections.

Understanding one-of-n Normalization

One-of-n is a very simple form of normalization. For an example, consider the forest cover example that we will examine in this chapter. The input to the neural network is statistics about a sample of forest region. The output signifies which of seven different tree types may be covering this land. The seven tree types are listed as follows:

- Spruce/Fir
- Lodgepole Pine
- Ponderosa Pine
- Cottonwood/Willow
- Aspen
- Douglas-fir
- Krummholz

If we were using one-of-n normalization, the neural network would have seven output neurons. Each of these seven neurons would represent one tree type. The tree type predicted by the neural network would correspond to the output neuron with the highest activation.

Generating training data for one-of-n is relatively easy. Simply assign a +1 to the neuron that corresponds to the tree that should have been chosen, and a -1 to the remaining neurons. For example, the Spruce/Fir tree type "ideal output" would be encoded as follows.

```
1,-1,-1,-1,-1,-1,-1
```

Likewise, the Ponderosa Pine would be encoded as follows.

```
-1,1,-1,-1,-1,-1,-1
```

The **OutputOneOf** class performs this sort of normalization. The one-of-n encoding is usually a good choice for input neurons. The example shown later in this chapter uses one-of-n to normalize the soil types used to predict the tree cover.

Understanding Equilateral Normalization

The output neurons are constantly checked against the ideal output values provided in the training set. The error between the actual output and the ideal output is represented by a percent. This can cause a problem for the

one-of-n normalization method. Consider whether the neural network predicted a Spruce/Fir tree, when it should have predicted a Ponderosa Pine. We would have output and ideal as follows:

```
Ideal Output: 1,-1,-1,-1,-1,-1,-1
Actual Output: -1,1,-1,-1,-1,-1,-1
```

The problem is that only two output neurons are incorrect. We would like to spread the "guilt" for this error over more of the neurons. To do this, we must come up with a unique set of values for each. Each set of values should have an equal Euclidean distance from the others. The equal distance makes sure that incorrectly choosing tree 3 for tree 4 has the same error weight as choosing tree 5 for tree 1.

The following code segment shows how to use the **Equilateral** class to generate these values.

```
Equilateral eq = new Equilateral(7,-1,1);
for(int i=0;i<7;i++) {
  StringBuilder line = new StringBuilder();
  line.append(i);
line.append(':');
double[] d = eq.encode(i);
  for(int j=0;j<d.length;j++)
  {
    if( j>0 )
      line.append(',');
    line.append(d[j]);
  }
  System.out.println(line.toString());
}
```

This would produce the following output.

```
0:0.7637,0.4409,0.3118,0.2415,0.1972,0.1666
1:-0.7637,0.4409,0.3118,0.2415,0.1972,0.1666
2:0.0,-0.8819,0.3118,0.2415,0.1972,0.1666
3:0.0,0.0,-0.9354,0.2415,0.1972,0.1666
4:0.0,0.0,0.0,-0.9660,0.1972,0.1666
5:0.0,0.0,0.0,0.0,-0.9860,0.1666
6:0.0,0.0,0.0,0.0,0.0,-1.0
```

These are the values that would be used for tree types 0 through 6. As you can see the difference between each of these usually involves more than one

neuron. This will spread the training more effectively. Equalaterial normalization requires that there be at least three sets. If there are only two sets, simply use one-of-n encoding, as the error will have to be equally spread over the two output neurons, as there are only two.

The Euclidean normalization technique produces one fewer output neurons than one-of-n. Notice that each of the above sets contains only six numbers. This is a side effect of finding values that are equal in distance.

What is meant by each of the sets being equal in distance from each other? It means that their Euclidean distance is equal. The Euclidean distance can be calculated using Equation 6.2.

Equation 6.2: Euclidean Distance

$$distance = \sqrt{\frac{(i_1 - a_1)^2 + (i_2 - a_2)^2 + \cdots + (i_n - a_n)^2}{n}}$$

In the above equation the variable "i" represents the ideal output value, the variable "a" represents the actual output value. There are "n" sets of ideal and actual. Every set of values in the above listing will produce a Euclidean distance of 0.623. Euclidean normalization is implemented using the **Equilateral** class in Encog.

Using the DataNormalization Class

Encog supports normalization using the **DataNormalization** class. The normalization class works by accepting data through input fields and processing them into output fields. There are really two ways to use the **DataNormalization** class. They are summarized as follows:

- Batch processing
- Single record processing

In single record processing, you provide a set of numbers to be normalized. These numbers are normalized and returned to you. Batch processing accepts a data source and data target. All records are read from the data source, then normalized, and written to the data target. Often you will batch process data from one CSV file to another. Batch processing is particularly

useful when training a neural network. Single record processing is very useful for when you are actually using the neural network.

Using Normalization in Batch Mode

The general format for using the **DataNormalization** class in batch mode is as follows.

```
DataNormalization norm = new DataNormalization();

norm.setReport(this);
norm.setTarget( ...data target... );
norm.addInputField( ... Input Field 1 ... );
norm.addInputField( ... Input Field 2 ... );
norm.addInputField( ... Input Field 3 ... );

norm.addOutputField(... Output Field 1 ... );
norm.addOutputField(... Output Field 2 ... );
norm.addOutputField(... Output Field 3 ... );

norm.addSegregator(... Segregator 1...);

norm.process();
```

First, a new **DataNormalization** object is created. Then a reporting object is specified. The reporting object will receive status updates as normalization progresses. A normalization job can take some time to process when dealing with large data sets.

Next, the input and output fields are added. The input fields specify the data sources to use. The data may come from one single source, or it may come from several to be aggregated together. The output fields specify how the data should be normalized. Segregators can also be added to trim some of the data. Segregators can be very useful for separating the data into groups. One group might be used to train the network, a second might be used to evaluate the network after it has been trained. The forest cover example, shown later in this chapter, will expand upon batch normalization processing.

Using Normalization in Single Record Mode

Using a **DataNormalization** object in single record mode is a simplified form of batch mode. There is no need to specify a reporting object, segregators or data target. The results for a single record will be calculated very quickly and returned in a **NeuralData** object.

```
DataNormalization norm = new DataNormalization();

norm.addInputField( ... Input Field 1 ... );
norm.addInputField( ... Input Field 2 ... );
norm.addInputField( ... Input Field 3 ... );

norm.addOutputField(... Output Field 1 ... );
norm.addOutputField(... Output Field 2 ... );
norm.addOutputField(... Output Field 3 ... );

NeuralData input = norm.buildForNetworkInput(data);
NeuralData output = this.network.compute(input);
```

As you can see, the input and output fields are created, and added, just as before. However, rather than calling **process**, we call **buildForNetworkInput**. This method will create input suitable for the **compute** method of the **BasicNetwork** class.

Specifying the Input Fields

Input fields map to the individual elements from the raw data. Output fields specify the normalized fields that are produced by the normalization class. However, these "output fields" will then become the input to the neural network. There is not necessarily a one-to-one correspondence between the normalization input fields and output fields. Some of the raw data may be ignored, or used to filter the rest of the data.

There are several different types of input fields that the Encog normalization class can work with. All input fields must implement the interface **InputField**. Input fields simply specify where to get a value from; they do not specify how to normalize it. The output fields will specify how fields are to be normalized. Input can be taken from a number of different sources. The different input field types are covered in the next sections.

Using BasicInputField

The **BasicInputField** class is the simplest of the input fields. It is the base class for most of the other input field types. It simply passes its current value to the output fields. It is not capable of reading from an input source. The **BasicInputField** is useful in its own right. It is often used when a normalization object is to be constructed for single record mode. **BasicInputField** objects can be used to provide a place to store the prenormalized data. The example program in Chapter 8, "Other Supervised Training Methods" demonstrates this concept.

The following code shows a **BasicInputField** being set up for single record mode.

```
InputField fuelIN;

norm.addInputField(fuelIN = new BasicInputField());

fuelIN.setMax(200);
fuelIN.setMin(0);
```

Here you can see an input field that will hold the amount of fuel remaining in a spacecraft. These lines are from the example you will see in Chapter 8. Here you can see that the minimum and maximum values are set. In batch mode, these values will be calculated as the data is processed. However, in single-record mode, the **min** and **max** values must be supplied to Encog.

Using InputFieldArray1D

Data to be normalized can be read from a one-dimensional array. Each row in the array maps to one record that will be fed into the neural network. Because the array is one-dimensional, only a single field per record is allowed. Of course you can aggregate multiple one-dimensional arrays by using multiple **InputFieldArray1D** objects. The following code shows how to use an **InputfieldArray1D** object.

```
public static final double[] ARRAY_1D = { 1.0,2.0,3.0,4.0,5.0 };

InputField a;
double[] arrayOutput = new double[5];
```

```
NormalizationStorageArray1D target = new
  NormalizationStorageArray1D(arrayOutput);

DataNormalization norm = new DataNormalization();
norm.setReport(new NullStatusReportable());
norm.setTarget(target);
norm.addInputField(a = new InputFieldArray1D(false,ARRAY_1D));
norm.addOutputField(new OutputFieldRangeMapped(a,0.1,0.9));
norm.process();
```

The above code normalizes the values contained in **ARRAY_1D** and stores them in **arrayOutput**. You will notice that there are two parameters passed to the **InputFieldArray1D** object, as seen here.

```
new InputFieldArray1D(false,ARRAY_1D)
```

The first parameter specifies whether this field is actually used by the neural network. If the value were **false**, the field would likely only be used for comparison purposes. The second parameter specifies the actual array.

It is somewhat limiting that a one-dimensional array only allows a single field per array. Encog also allows you to use a two-dimensional array, which is more flexible.

Using InputFieldArray2D

Encog can normalize data using a two-dimensional array. Each of array's rows becomes one record. Each column can be used as a single input field. It is not necessary to make use of every column in the array. The following code segment shows how to use **InputFieldArray2D**.

```
public static final double[][] ARRAY_2D = {
  {1.0,2.0,3.0,4.0,5.0},
  {6.0,7.0,8.0,9.0} };

InputField a,b;
double[][] arrayOutput = new double[2][2];

NormalizationStorageArray2D target = new

NormalizationStorageArray2D(arrayOutput);

DataNormalization norm = new DataNormalization();
```

```
norm.setReport(new NullStatusReportable());
norm.setTarget(target);

norm.addInputField(a = new InputFieldArray2D(true,ARRAY_2D,0));


norm.addInputField(b = new InputFieldArray2D(true,ARRAY_2D,1));


norm.addOutputField(new OutputFieldRangeMapped(a,0.1,0.9));
norm.addOutputField(new OutputFieldRangeMapped(b,0.1,0.9));

norm.process();
```

You will notice that there are three parameters passed to the **InputFieldArray2D** object, as seen here.

```
InputFieldArray2D(true,ARRAY_2D,0));
```

The first parameter specifies if this field is actually used by the neural network. If the value were **false**, the field would likely only be used for comparison purposes. The second parameter specifies the actual array. The zero parameter specifies the column that this field should map to. The value zero specifies the first column.

Using InputFieldCSV

One of the most commonly used input field types is the **InputFieldCSV** class. This class allows fields to be read from a CSV file. Often the output fields will be written to a CSV file as well. The following code shows how to define three fields to be read in from a CSV file.

```
double[][] outputArray = new double[2][5];

InputField a;
InputField b;
InputField c;

DataNormalization norm = new DataNormalization();
norm.setReport(new NullStatusReportable());
norm.setTarget(new NormalizationStorageCSV(FILENAME));
norm.addInputField(a = new InputFieldCSV(false,FILENAME,0));
norm.addInputField(b = new InputFieldCSV(false,FILENAME,1));
```

```
norm.addInputField(c = new InputFieldCSV(false,FILENAME,2));

norm.addOutputField(new OutputFieldRangeMapped(a,0.1,0.9));
norm.addOutputField(new OutputFieldRangeMapped(b,0.1,0.9));
norm.addOutputField(new OutputFieldRangeMapped(c,0.1,0.9));
norm.setTarget(new NormalizationStorageArray2D(outputArray));

norm.process();
```

You will notice that there are three parameters passed to the **InputFieldCSV** object, as seen here.

```
new InputFieldCSV(false,FILENAME,0)
```

The first parameter specifies if this field is actually used by the neural network. If the value were **false**, the field would likely only be used for comparison purposes. The second parameter specifies the filename of the CSV file. The zero parameter specifies the column that this field should map to. The value zero specifies the first column.

Using InputFieldNeuralDataSet

It is also possible to take input fields from a **NeuralDataSet** object. This is done using an **InputFieldNeuralDataSet** object. The **NeuralDataSet** objects will more often be the target of normalization, rather than the source. However, it may sometimes be useful to normalize from a **NeuralDataSet**. The following code reads from a **NeuralDataSet** and normalizes the results to a two dimensional array.

```
InputField a,b;
double[][] arrayOutput = new double[2][2];

BasicNeuralDataSet dataset = new
BasicNeuralDataSet(ARRAY_2D,null);

NormalizationStorageArray2D target = new
NormalizationStorageArray2D(arrayOutput);

DataNormalization norm = new DataNormalization();
norm.setReport(new NullStatusReportable());
norm.setTarget(target);

norm.addInputField(a = new
```

```
  InputFieldNeuralDataSet(false,dataset,0));
norm.addInputField(b = new
  InputFieldNeuralDataSet(false,dataset,1));

norm.addOutputField(
  new OutputFieldRangeMapped(a,0.1,0.9));

norm.addOutputField(new OutputFieldRangeMapped(b,0.1,0.9));

norm.process();
```

You will notice that there are three parameters passed to the **InputFieldNeuralDataSet** object, as seen here.

```
New InputFieldNeuralDataSet(false,dataset,0)
```

The first parameter specifies if this field is actually used by the neural network. If the value were **false**, the field would likely only be used for comparison purposes. The second parameter specifies the dataset to use. The zero parameter specifies the column that this field should map to. The value zero specifies the first column.

Specifying the Output Fields

Encog uses output fields to process the input fields. The output fields are what will be fed into the neural network. The output fields of normalization become the input fields to the neural network. If you specify eight output fields on your normalization object, you will need to have a neural network with eight input neurons to receive this data.

There are many different kinds of output field types. The type of output neuron specifies the type of normalization that you are using. It is very common to use different normalization types for different input neurons. Each output neuron will usually map to a single input field. This output field will process that input field's values. Sometimes an output field will not map to a specific input field. Such an output field is considered synthetic. Synthetic fields will be discussed later in this section.

Encog currently supports two types of output fields. Grouped fields are grouped together with other output fields. The values of the members of the group influence each other. Non-grouped fields act independently. In this

section you will learn about both grouped and nongrouped fields. We will begin with field groups.

Understanding Field Groups

Several of the output fields supported by Encog can belong to field groups. A field group is a collection of objects associated with a group object. Any group object must be of a class that implements the **FieldGroup** interface. Grouped fields do not act independently. The value of a single field in a group is affected by the other grouped fields.

Using OutputFieldMultiplicative

The **OutputFieldMultiplicative** object implements a grouped field that makes use of multiplicative normalization. Because multiplicative normalization is grouped, the values of the individual grouped fields will influence each other. The following code shows three different output fields being set up with multiplicative normalization.

```
MultiplicativeGroup group = new MultiplicativeGroup();

norm.addOutputField(new OutputFieldMultiplicative(group,a));
norm.addOutputField(new OutputFieldMultiplicative(group,b));
norm.addOutputField(new OutputFieldMultiplicative(group,c));
```

The multiplicative normalization algorithm ensures that all grouped fields are between the range of -1 and +1. Further, it ensures that their vector length is defined as one. The vector length is the square root of a sum of squares, as shown in Equation 6.3.

Equation 6.3: Calculating Vector Length for Multiplicative Normalization

$$l = \sqrt{\sum_{i=0}^{n-1} x_i^2}$$

The equation above essentially squares the value of every grouped field. The resulting length is the square root of the sum of these individual squares. We then divide each of the field values by this length, as shown in Equation 6.4.

Equation 6.4: Multiplicative Normalization of Each Value

$$\hat{x_i} = \frac{x_i}{l}, l = 0, \ldots, n-1$$

The multiplicative normalization type can be very useful for vector quantization. One of the problems with multiplicative normalization is that the sign of the input fields is completely disregarded. This is because each of the inputs is squared. Because of this, Z-axis normalization is often used in place of multiplicative normalization.

Using OutputFieldZAxis

Z-axis normalization is also often used with self organizing maps. Encog implements Z-axis normalization using the **OutputFieldZAxis** class. Z-axis normalization accomplishes the same goal as multiplicative normalization, in that it causes the vector length of the grouped fields to be one. The following code shows how to set up three input fields to be normalized using Z-axis normalization

```
ZAxisGroup group = new ZAxisGroup();

norm.addOutputField(new OutputFieldZAxis(group,a));
norm.addOutputField(new OutputFieldZAxis(group,b));
norm.addOutputField(new OutputFieldZAxis(group,c));
norm.addOutputField(new OutputFieldZAxisSynthetic(group));
```

One thing that you should notice from the above code is that the number of input and output fields is not the same. Z-axis normalization will always result in one additional output field than the number of input fields that were provided to it. This additional output field, which uses the **OutputFieldZAxisSynthetic** class, is called a synthetic field.

To perform Z-axis normalization, a normalization factor is calculated. This factor is calculated using Equation 6.5.

Equation 6.5: Z-Axis Normalization Factor

$$f = \frac{1}{\sqrt{n}}$$

The normalization factor is calculated independently of the actual data. This allows the sign of the data to be preserved. This factor is then applied to each of the grouped fields. This step is performed in Equation 6.6.

Equation 6.6: Normalizing with the Z-Axis Normalization Factor

$$\hat{x}_i = f x_i \,, i = 0, \ldots, n-1$$

The synthetic field must now be calculated. This is where z-axis normalization derives its name. The additional field is thought of as an additional axis, just as the z-axis is an imaginary axis used in computer graphics to give the appearance of three dimensions on a two dimensional display. The synthetic field is calculated with Equation 6.7.

Equation 6.7: The Z-Axis Synthetic Field

$$s = f\sqrt{n - l^2}$$

Either Z-axis or multiplicative normalization should be used when you need a consistent vector length. Z-axis normalization is usually a better choice than multiplicative. One of the few times that multiplicative normalization may perform better than Z-axis is when all of the input fields are near zero. In this case the synthetic field may dominate them.

Using OutputFieldDirect

The **OutputFieldDirect** is a very simple field class that simply passes the input field directly to the output. Normalization is not performed. The following code shows how to set up a direct output field.

```
norm.addOutputField(new OutputFieldDirect(inputField));
```

Direct output fields can be very useful when an input field is already normalized or is already within an acceptable range.

Using OutputFieldRangedMapped

The **OutputFieldRangedMapped** field object allows an input field to be mapped into a specific range. The range that is usually chosen is a range that is either close to -1 to +1 or 0 to +1. The following line of code shows how to use **OutputFieldRangedMapped.**

```
norm.addOutputField(
  new OutputFieldRangeMapped(inputField,0.0,1.0));
```

In the above code, the field **inputField** is mapped to a range between 0.0 and 1.0. This is one of the most commonly used neural network normalization techniques.

Using OutputFieldEncode

The **OutputFieldEncode** field object allows different ranges of an input field to be mapped to output field values. The following code shows a typical set up for this field type.

```
OutputFieldEncode encode = new OutputFieldEncode(inputField);
encode.addRange(0, 999, 0.1);
encode.addRange(1000, 1999, 0.2);
encode.addRange(2000, 2999, 0.3);
encode.setCatchAll(0.5);
```

Here you see an encode field that will encode three different ranges. This code also includes a "catch all". If the input field is between 0 and 999 the output value will be 0.1. Likewise if the input field is between the other two ranges the output will be either 0.2 or 0.3. If the input field does not match any of the ranges provided, the output will be the "catch all" value of 0.5. If a "catch all" value was not provided, the "catch all" defaults to zero. The **OutputFieldEncode** field can provide more precise control than **OutputFieldRangeMapped** field; however, the ranges need to be defined manually.

Using OutputOneOf

The last two output field types that will be examined are used to encode nominal values. Nominal values indicate set membership. Later in this chapter we will look at an example program that attempts to predict the type of tree that will live on a sample area of land. This tree type is a nominal value, as there are seven distinct tree types that were sampled from the land surveyed.

As discussed previously, there are two different ways that this group can be represented in a neural network. These two approaches are called one-of-n and equilateral normalization. For more information on the differences between one-of-n and equilateral normalization, refer to the material earlier in this chapter.

To implement one-of-n encoding in Encog use the **OutputOneOf** class. The following lines of code demonstrate how to set up one-of-n encoding.

```
OutputOneOf outType = new OutputOneOf(1.0,0.0);
outType.addItem(coverType, 1);
outType.addItem(coverType, 2);
outType.addItem(coverType, 3);
outType.addItem(coverType, 4);
outType.addItem(coverType, 5);
outType.addItem(coverType, 6);
outType.addItem(coverType, 7);
norm.addOutputField(outType, true);
```

Not all output field objects create a single output field. The above code would actually create seven output fields. If the item were a member of one of the sets, the corresponding output field would have a value of 1.0, otherwise it would have a value of 0.0. These two values were specified by the constructor which created an **OutputOneOf** object called **outType**.

Using OutputEquilateral

To make use of equilateral norm, you should use the **OutputEquilateral** class. The following lines of code show how to set up an **OutputEquilateral** set of output fields.

```
OutputEquilateral outType = new OutputEquilateral(1.0,0.0);
outType.addItem(coverType, 1);
outType.addItem(coverType, 2);
outType.addItem(coverType, 3);
outType.addItem(coverType, 4);
outType.addItem(coverType, 5);
outType.addItem(coverType, 6);
outType.addItem(coverType, 7);
norm.addOutputField(outType, true);
```

The **OutputEquilateral** object created above will actually create six output fields. As previously discussed, in this chapter, equilateral

normalization actually one less than the number of items provided. As a result, you must have at least three item classes for equilateral normalization to be effective.

Using Segregators

Segregators are used to exclude certain records from normalization. You can segregate based on input field values, or you can simply segregate a certain percentage of the record. This can allow you exclude certain records all together, or simple segregate records into different sets. This can be a very effective way to build a training and evaluation set.

All Encog segregators implement the **Segregator** interface. The following sections will examine the different Encog segregation types.

Using IndexRangeSegregator

The **IndexRangeSegregator** is useful when you know exactly how many records are in your dataset and you would like to specify a certain range. For example, if you had 10,000 records, and you knew that you wanted records 1 through 7,500, you might choose to use an **IndexRangeSegregator**. The following lines of code illustrate this concept.

```
IndexSampleSegregator segregator =
  new IndexRangeSegregator(0,7499);
norm.addSegregator(segregator);
```

There are several disadvantages to this simple of an approach. First, you must know exactly how many values there are to normalize. Second, the records you are collecting will all occur next to each other. If you are simply grabbing the first 7,500 records, you are only accessing records from the first part of the dataset. It would be better to have a more uniform distribution.

Using IndexSampleSegregator

The **IndexSampleSegregator** does not require you to know the size of the dataset, and it provides a more uniform distribution. The following lines of code show how to set up an **IndexSampleSegregator**.

```
IndexSampleSegregator segregator =
```

```
  new IndexSampleSegregator(start,stop,size);
norm.addSegregator(segregator);
```

The variables **start**, **stop** and **size** specify how to select elements. First, you must select a sample size. This sample size will be repeated over and over as elements are processed. Only elements between the **start** and **stop** indexes will be included. For example, consider the following list of ten records.

```
Record 0
Record 1
Record 2
Record 3
Record 4
Record 5
Record 6
Record 7
Record 8
Record 9
```

We will specify a sample size of five, a start index of zero and an ending index of three. The following records would be included.

```
Record 0: Sample 0, Sample index 0, Included
Record 1: Sample 0, Sample index 1, Included
Record 2: Sample 0, Sample index 2, Included
Record 3: Sample 0, Sample index 3, Included
Record 4: Sample 0, Sample index 4, Not Included
Record 5: Sample 1, Sample index 0, Included
Record 6: Sample 1, Sample index 1, Included
Record 7: Sample 1, Sample index 2, Included
Record 8: Sample 1, Sample index 3, Included
Record 9: Sample 1, Sample index 4, Not Included
```

Because the sample repeats through the dataset, a much more uniform distribution is achieved.

Using IntegerBalanceSegregator

Sometimes the training set will contain too many samples of one particular item. Consider the forest cover example that will be presented in this chapter. For all of the land areas sampled, certain tree are far more common than others. This could cause the more prevalent tree types to saturate the training data. To prevent this from happening, the data should

be balanced. The **IntegerBalanceSegregator** class can perform such a balance. The following lines of code show how to set up an **IntegerBalanceSegregator**.

```
IntegerBalanceSegregator segregator = new
IntegerBalanceSegregator(balanceField,count);
norm.addSegregator(segregator);
```

You must specify a field to balance on, called **balanceField**. Additionally, a **count** must be provided to tell Encog the maximum number of records for each unique value in the **balanceField**. To determine how many unique sets there are, Encog will truncate each unique value in the **balanceField** to an integer. Every unique integer will be allowed up to **count** records.

Once the normalization has been processed, you can display how many samples were present for each unique integer value on the balancing field. The following code shows how to do this.

```
norm.process();
System.out.println("Samples per tree type:");
System.out.println(segregator.dumpCounts());
```

This will display a simple listing of the count for each of the trees.

Using RangeSegregator

The range segregator object allows you to exclude records because the value of one of the input fields falls in a specific range. For example, the forest cover data contains a field that designates which wilderness area the data was collected from. If you wished to only process data from one wilderness area, you would use a range segregator. The following lines of code show how to use a **RangeSegregator**.

```
RangeSegregator seg = new RangeSegregator(inputField,false);
seg.addRange(1, 10, true);
norm.addSegregator(seg);
```

The above code would only allow records in the range of 1 to 10. The **true** value on the **addRange** method call indicates that values in this range are included. The **false** value on the constructor indicates that records that do not fall under any of the defined ranges should be excluded.

Normalization Targets

Normalization targets specify what Encog should actually do with the normalized data that it generates. Every normalization target must implement the **NormalizationStorage** interface. Normalization targets are provided for arrays, CSV fields and **NeuralDataSet** objects. The following sections describe the Encog normalization targets.

Using the NormalizationStorageArray1D Class

A one-dimensional array can be used as the target for normalized data. The **NormalizationStorgageArray1D** class is used to do this. The one-dimensional array is limited, in that it can only hold one single field. The following code shows how to normalize to a one-dimensional array.

```
double[] arrayOutput = new double[5];

NormalizationStorageArray1D target = new
NormalizationStorageArray1D(arrayOutput);

DataNormalization norm = new DataNormalization();
norm.setTarget(target);
```

As you can see from the above code, the one-dimensional array is created and passed to the constructor of the **NormalizationStorgageArray1D** object. If more than one output field is generated by the normalization class, an error will occur. To support multiple fields per record the **NormalizationStorgageArray2D** class should be used.

Using NormalizationStorageArray2D

A two-dimensional array can be used as the target for normalized data. The **NormalizationStorgageArray2D** class is used to do this. The two-dimensional array is less limited than the one-dimensional array, in that it can only hold multiple fields. The following code shows how to set up to normalize to a two-dimensional array.

```
double[][] arrayOutput = new double[2][2];

NormalizationStorageArray2D target = new
NormalizationStorageArray2D(arrayOutput);
```

```
DataNormalization norm = new DataNormalization();
norm.setTarget(target);
```

As you can see from the above code the two-dimensional array is created and passed to the constructor of the **NormalizationStorgageArray2D** object.

Using NormalizationStorageCSV

A very common technique is to use a CSV file to hold the normalized data. The following code defines a normalization target that will save to a CSV file.

```
File file = new File("output.csv");
DataNormalization norm = new DataNormalization();
norm.setTarget(new NormalizationStorageCSV(file));
```

Once the normalization process is complete the CSV file will hold the results of the normalization. This CSV file can then be used for neural network training.

Using NormalizationStorageNeuralDataSet

You can also normalize directly into a **NeuralDataSet**. The following code shows how to create a new dataset and normalize directly into it.

```
DataNormalization norm = new DataNormalization();
norm.setTarget(new NormalizationStorageNeuralDataSet (2,1));
norm.process();
NeuralDataSet training = norm.getTarget().getDataSet();
```

Once the normalization process is complete, the dataset will contain the results of the normalization.

You can also pass an already created **NeuralDataSet** into the constructor of the **BufferedNeuralDataSet**. This can be a powerful technique for saving the normalized data to a binary file that can be used for training later. Binary files train much faster than CSV files. The following lines show how this is done.

```
BufferedNeuralDataSet buffer =
  new BufferedNeuralDataSet(filename);

DataNormalization norm = new DataNormalization();

norm.setTarget(new NormalizationStorageNeuralDataSet(buffer));

buffer.beginLoad(inputLayerSize,outputLayerSize);

norm.process();

buffer.endLoad();
```

The above code would create a binary file that contains the results of the normalization. This binary file could be used later to train a neural network. The forest cover example, shown later in the next section, uses this technique.

Running the Forest Cover Example

To demonstrate how to use normalization this chapter presents an example that attempts to predict what type of trees might be growing on an area of wilderness. It will use publicly available data. This example is meant to be very "real world". It demonstrates the steps that you might go through with a neural network project of your own. The following four steps are needed to set up and process a neural network.

- Obtaining the data
- Generate training and evaluation files
- Train the neural network
- Evaluate the neural network

We will begin in the next section with obtaining the raw data.

Obtaining the Raw Data

The data that we will use was obtained from the United States Forest Service (USFS). It can be downloaded from the University of California at Irvine, at the following URL.

http://kdd.ics.uci.edu/databases/covertype/covertype.html

The data to be used is described as follows (from the Web site).

```
The actual forest cover type for a given observation (30 x 30
meter cell) was determined from US Forest Service (USFS) Region 2
Resource Information System (RIS) data. Independent variables
were derived from data originally obtained from US Geological
Survey (USGS) and USFS data. Data is in raw form (not scaled) and
contains binary (0 or 1) columns of data for qualitative
independent variables (wilderness areas and soil types).
Summary Statistics
Number of instances (observations)  581012
Number of Attributes    54
Attribute breakdown     12 measures, but 54 columns of data (10
quantitative variables, 4 binary wilderness areas and 40 binary
soil type variables)
Missing Attribute Values      None
```

The file that you will download is named **covtype.data**. This file will be used in the next step to generate training and evaluation data.

Generating Data Files

You should place the **covtype.data** file from the last section in a directory. For this example, I will assume that it is in the location **c:\data**. You must modify the **Constant.java** file to reflect it. You can see the line to modify here.

```
/**
 * The base directory that all of the data for this example is
 * stored in.
 */
public static final File BASE_DIRECTORY = new File("c:\\data");
```

All of the other data files are based on this directory. To generate the files, run the forest example with the parameter **generate**. You must also specify an **e** or an **o** parameter to determine if equilateral or one-of-n normalization should be used for the tree types. Generally, you will get better results with equilateral. For more information on the difference between equilateral, refer to the material earlier in this chapter. For example, to run with equilateral training you would use the following arguments.

```
java ForestCover generate e
```

Of course you will need to add the appropriate path and class path information. Once the program executes, you will see the following output:

```
Step 1: Generate training and evaluation files
Generate training file
10000/0 Processing data (single pass)
20000/0 Processing data (single pass)
30000/0 Processing data (single pass)
40000/0 Processing data (single pass)
50000/0 Processing data (single pass)
60000/0 Processing data (single pass)
70000/0 Processing data (single pass)
80000/0 Processing data (single pass)
90000/0 Processing data (single pass)
100000/0 Processing data (single pass)
110000/0 Processing data (single pass)
120000/0 Processing data (single pass)
130000/0 Processing data (single pass)
140000/0 Processing data (single pass)
...
390000/0 Processing data (single pass)
400000/0 Processing data (single pass)
410000/0 Processing data (single pass)
420000/0 Processing data (single pass)
430000/0 Processing data (single pass)
Generate evaluation file
10000/0 Processing data (single pass)
20000/0 Processing data (single pass)
30000/0 Processing data (single pass)
40000/0 Processing data (single pass)
50000/0 Processing data (single pass)
60000/0 Processing data (single pass)
70000/0 Processing data (single pass)
80000/0 Processing data (single pass)
90000/0 Processing data (single pass)
100000/0 Processing data (single pass)
110000/0 Processing data (single pass)
120000/0 Processing data (single pass)
130000/0 Processing data (single pass)
140000/0 Processing data (single pass)
Step 2: Balance training to have the same number of each tree
10000/0 Processing data (single pass)
20000/0 Processing data (single pass)
Samples per tree type:
1 -> 3000 count
```

```
2 -> 3000 count
3 -> 3000 count
4 -> 2066 count
5 -> 3000 count
6 -> 3000 count
7 -> 3000 count

Step 3: Normalize training data
0/0 Analyzing file
10000/0 First pass, analyzing file
20000/0 First pass, analyzing file
10000/20066 Second pass, normalizing data
20000/20066 Second pass, normalizing data
```

First, when you generate the data files, **covtype.data** is split into training data and evaluation data. The training data, which is 75% of the file, is named **training.csv**. The evaluation data, which is 25% of the file, is named **evaluate.csv**.

Next, the training data is balanced so that there are at most 3,000 of each tree type. The data has considerably more of one tree types than others. This decreases training time, and also prevents one tree type from saturating the weight matrix with its patterns. The balanced tree data is written to the file **balance.csv**. There is no need to balance the evaluation data. The evaluation data is meant to be what the neural network faces after it is trained. We want to do nothing to "stage" the evaluation data.

Once the data has been balanced it must be normalized. The data is still in raw form in the **balance.csv** file. At this point the data has been pared down, but it is still in the same form as was in the original **covtype.data** file. The normalized data is written to the **normalized.csv** file. This is the file that will be used to train the neural network. The **DataNormalization** object is also saved to the **forest.eg** file. The **forest.eg** file is an Encog XML persistence file. Encog persistence will be covered in Chapter 7. The exact process that was used to normalize each field will be covered later in this chapter when the source code to the forest example is reviewed.

Now that the files have been generated, the neural network is ready to train. Training will be covered in the next section.

Training the Network

There are two methods provided for training. The first is simple console-mode training. For console training you must specify how long you would like the neural network to train in the **Constant.java** file. There is a constant named **TRAINING_MINUTES** that specifies how long to train the network. The default is 10 minutes, however you can change it to any number you like. Longer training times will produce better results. You can see the setting here.

```
/**
 * How many minutes to train for (console mode only)
 */
public static final int TRAINING_MINUTES = 10;
```

To begin console-mode training the following command should be used.

```
java ForestCover train
```

Of course you will need to add the appropriate path and class path information. Once the program executes, you will see the following output.

```
Converting training file to binary
Beginning training...
Iteration #1 Error:45.093191% elapsed time = 00:00:23 time left =
00:10:00
Iteration #2 Error:45.660918% elapsed time = 00:00:46 time left =
00:10:00
Iteration #3 Error:44.983507% elapsed time = 00:01:09 time left =
00:09:00
Iteration #4 Error:49.432105% elapsed time = 00:01:32 time left =
00:09:00
Iteration #5 Error:39.701852% elapsed time = 00:01:55 time left =
00:09:00
Iteration #6 Error:30.401943% elapsed time = 00:02:18 time left =
00:08:00
Iteration #25 Error:13.369462% elapsed time = 00:09:48 time left
= 00:01:00
Iteration #26 Error:13.275960% elapsed time = 00:10:14 time left
= 00:00:00
Training complete, saving network...
```

The ten-minute default is not enough to thoroughly train the neural network. However, it is enough for a quick example of what the program is

capable of. In this example the neural network was trained to around 13% error.

It is also possible to train using the GUI. GUI training displays statistics about training and does not require the training time to be specified. To begin in GUI training mode run the example with the **traingui** argument.

```
java ForestCover traingui
```

Of course you will need to add the appropriate path and class path information. Once the program executes, you will see the training dialog. Figure 6.1 shows the GUI training being used.

Figure 6.1: GUI Training

When you are ready to stop training, simply click "Stop" and training will cease. Once training has stopped, the neural network will be saved to the **forest.eg** file. As you can see from the above dialog, I ran the training for over two days. I allowed it to continue even further. However, training progressed very slowly after this point. Training was stopped once I had reached 63,328 iterations. This took five days and eleven hours. The additional three days of training had only lowered the error rate from 7.4% to 7.19%.

Now that the neural network has been trained, it is time to evaluate its performance.

Evaluating the Network

While evaluating the performance of the neural network the **evaluate.csv** file is used. This file contains the 25% of the raw data that was saved for evaluation. To evaluate the neural network you should run the example with the evaluate argument.

```
java ForestCover evaluate
```

Of course you will need to add the appropriate path and class path information. Once the program executes, you will see the following output.

```
Total cases:145253
Correct cases:92725
Correct percent:64%
Tree Type #0 - Correct/total: 35560/52986(67%)
Tree Type #1 - Correct/total: 39151/70779(55%)
Tree Type #2 - Correct/total: 6724/8947(75%)
Tree Type #3 - Correct/total: 650/681(95%)
Tree Type #4 - Correct/total: 2227/2384(93%)
Tree Type #5 - Correct/total: 3451/4348(79%)
Tree Type #6 - Correct/total: 4962/5128(97%)
```

The above output is from a neural network that was trained to a 7.19% error rate. Overall, the success rate was 62%. However, you will notice that tree type #1 is the primary reason for this somewhat low score. Most of the other tree types scored at least 70% or higher. Some scored 90% or higher.

Further training may be able to improve it. More advanced handling of the data may improve it as well. This example does not make use of the "wilderness area" column. This column tells from which wilderness area the data was collected. You may want to limit the example to only one wilderness area, or in some way incorporate this field into the input data for the neural network. The four areas are relatively close, so it is unlikely that it will have a significant effect; however, it is an area for further study.

Another method to further refine the results might be to examine what tree type the network is consistently guessing incorrectly for tree type 0. It could be that these two species of trees are very similar and some additional criteria might be required to tell them apart.

In the past few sections you saw how to execute the forest cover example. In the next section we will examine how the forest example was constructed.

Understanding the Forest Cover Example

The last few sections described how to execute the forest cover example. We will now look the source code behind the forest cover neural network

example. There are several files that make up this example. These files are listed here.

- **Constant.java** – Configuration information for the program.
- **Evaluate.java** – Evaluate the trained neural network.
- **ForestCover.java** – Main entry point for the program.
- **GenerateData.java** – Generate the data files.
- **TrainNetwork.java** – Train the neural network.

The **Constant** class contains configuration items that you can change. For example, you can set the number of hidden neurons to use. By default the program uses 100 hidden neurons. The main entry point for the program is the **ForestCover** class. This class is shown in Listing 6.1.

Listing 6.1: The Forest Cover Program Entry Point

```
package org.encog.examples.neural.forest.feedforward;

import org.encog.normalize.DataNormalization;
import org.encog.persist.EncogPersistedCollection;
import org.encog.util.logging.Logging;

public class ForestCover {

  public static void generate(boolean useOneOf) {

  GenerateData generate = new GenerateData();
  generate.step1();
  generate.step2();
  DataNormalization norm =
    generate.step3(useOneOf);

  EncogPersistedCollection encog =
  new EncogPersistedCollection(
    Constant.TRAINED_NETWORK_FILE);
      encog.add(Constant.NORMALIZATION_NAME,
        norm);
  }

  public static void train(boolean useGUI) {
    TrainNetwork program = new TrainNetwork();
    program.train(useGUI);
  }

  public static void evaluate() {
```

```
    Evaluate evaluate = new Evaluate();
    evaluate.evaluate();
  }

  public static void main(String args[]) {
    if (args.length < 1) {
      System.out.println(
        "Usage: ForestCover [generate
[e/o]/train/traingui/evaluate] ");
    } else {
      Logging.stopConsoleLogging();
      if (args[0].equalsIgnoreCase(
        "generate")) {
      if (args.length < 2) {
        System.out.println(
"When using generate, you must specify an 'e' or an 'o' as the
second parameter.");
      } else {
        boolean useOneOf;

        if (args[1].toLowerCase().equals("e"))
          useOneOf = false;
        else
          useOneOf = true;

        generate(useOneOf);
      }
  } else if (args[0].equalsIgnoreCase("train"))
    train(false);
  else if (args[0].equalsIgnoreCase("traingui"))
    train(true);
  else if (args[0].equalsIgnoreCase("evaluate"))
    evaluate();
    }
  }
}
```

As you can see this class is mainly concerned with passing control to one of the other classes listed above. We will examine each of these classes in the following sections.

Generating Training and Evaluation Data

The **generate** method is used to generate the training and evaluation data. This method begins by accepting a parameter to determine if one-of-n normalization should be used.

```
public static void generate(boolean useOneOf) {
```

Next, an instance to the **GenerateData** class is created. This class will be examined later in this section.

```
  GenerateData generate = new GenerateData();
```

Steps one and two of the generation process are executed. Step one segregates the data into training and evaluation files. Step two balances the numbers of cover types we have so that one cover type does not saturate the training.

```
  generate.step1();
  generate.step2();
```

Step 3 of file generation is executed. The **DataNormalization** object that was used by step three is obtained.

```
  DataNormalization norm = generate.step3(useOneOf);
```

The normalization object is then saved to an Encog persistence file. Encog persistence will be covered in greater detail in Chapter 7.

```
  EncogPersistedCollection encog =
    new EncogPersistedCollection(
      Constant.TRAINED_NETWORK_FILE);
  encog.add(Constant.NORMALIZATION_NAME, norm);
```

The **generate** method makes use of methods from the **GenerateData** class. The **GenerateData** class is shown in Listing 6.2.

Listing 6.2: The Forest Cover Data File Generation

```
package org.encog.examples.neural.forest.feedforward;

import java.io.File;

import org.encog.StatusReportable;
import org.encog.normalize.DataNormalization;
```

```
import org.encog.normalize.input.InputField;
import org.encog.normalize.input.InputFieldCSV;
import org.encog.normalize.output.OutputField;
import org.encog.normalize.output.OutputFieldDirect;
import org.encog.normalize.output
  .OutputFieldRangeMapped;
import org.encog.normalize.output.nominal
  .OutputEquilateral;
import org.encog.normalize.output.nominal.OutputOneOf;
import org.encog.normalize.segregate
  .IntegerBalanceSegregator;
import org.encog.normalize.segregate.index
  .IndexSampleSegregator;
import org.encog.normalize.target
  .NormalizationStorageCSV;

public class GenerateData implements StatusReportable {

  public void buildOutputOneOf(
    DataNormalization norm,
    InputField coverType) {
    OutputOneOf outType = new OutputOneOf(0.9, 0.1);
    outType.addItem(coverType, 1);
    outType.addItem(coverType, 2);
    outType.addItem(coverType, 3);
    outType.addItem(coverType, 4);
    outType.addItem(coverType, 5);
    outType.addItem(coverType, 6);
    outType.addItem(coverType, 7);
    norm.addOutputField(outType, true);
  }

  public void buildOutputEquilateral(DataNormalization norm,
    InputField coverType) {
    OutputEquilateral outType =
      new OutputEquilateral(0.9, 0.1);
      outType.addItem(coverType, 1);
      outType.addItem(coverType, 2);
      outType.addItem(coverType, 3);
      outType.addItem(coverType, 4);
      outType.addItem(coverType, 5);
      outType.addItem(coverType, 6);
      outType.addItem(coverType, 7);
      norm.addOutputField(outType, true);
  }
```

```
  public void copy(
    File source,
    File target,
    int start,
    int stop,
    int size) {

    InputField inputField[] = new InputField[55];

    DataNormalization norm = new DataNormalization();
    norm.setReport(this);
    norm.setTarget(new NormalizationStorageCSV(target));
    for (int i = 0; i < 55; i++) {
      inputField[i] = new InputFieldCSV(true, source, i);
      norm.addInputField(inputField[i]);
      OutputField outputField =
        new OutputFieldDirect(inputField[i]);
      norm.addOutputField(outputField);
    }

// load only the part we actually want, i.e. training or eval

    IndexSampleSegregator segregator2 =
      new IndexSampleSegregator(start, stop, size);
      norm.addSegregator(segregator2);
      norm.process();
    }

    public void narrow(
      File source,
      File target,
      int field,
      int count) {

      InputField inputField[] = new InputField[55];

      DataNormalization norm = new DataNormalization();
      norm.setReport(this);
      norm.setTarget(new NormalizationStorageCSV(target));
      for (int i = 0; i < 55; i++) {
        inputField[i] = new InputFieldCSV(true, source, i);
        norm.addInputField(inputField[i]);
        OutputField outputField =
          new OutputFieldDirect(inputField[i]);
        norm.addOutputField(outputField);
      }
```

```
      IntegerBalanceSegregator segregator =
        new IntegerBalanceSegregator(
      inputField[field], count);
      norm.addSegregator(segregator);

      norm.process();
      System.out.println("Samples per tree type:");
      System.out.println(segregator.dumpCounts());
    }

    public void step1() {
      System.out.println(
        "Step 1: Generate training and evaluation files");
      System.out.println("Generate training file");
      copy(Constant.COVER_TYPE_FILE,
        Constant.TRAINING_FILE, 0, 2, 4); // take 3/4
      System.out.println("Generate evaluation file");
      copy(Constant.COVER_TYPE_FILE,
        Constant.EVALUATE_FILE, 3, 3, 4); // take 1/4
    }

    public void step2() {

      System.out.println(
"Step 2: Balance training to have the same number of each tree");
      narrow(Constant.TRAINING_FILE, Constant.BALANCE_FILE,
        54, 3000);
    }

    public DataNormalization step3(boolean useOneOf) {
      System.out.println("Step 3: Normalize training data");
      InputField inputElevation;
      InputField inputAspect;
      InputField inputSlope;
      InputField hWater;
      InputField vWater;
      InputField roadway;
      InputField shade9;
      InputField shade12;
      InputField shade3;
      InputField firepoint;
      InputField[] wilderness = new InputField[4];
      InputField[] soilType = new InputField[40];
      InputField coverType;
```

```
    DataNormalization norm = new DataNormalization();
    norm.setReport(this);
    norm.setTarget(
      new NormalizationStorageCSV(Constant.NORMALIZED_FILE));
    norm.addInputField(inputElevation = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 0));
    norm.addInputField(inputAspect = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 1));
    norm.addInputField(inputSlope = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 2));
    norm.addInputField(hWater = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 3));
    norm.addInputField(vWater = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 4));
    norm.addInputField(roadway = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 5));
    norm.addInputField(shade9 = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 6));
    norm.addInputField(shade12 = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 7));
    norm.addInputField(shade3 = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 8));
    norm.addInputField(firepoint = new InputFieldCSV(true,
      Constant.BALANCE_FILE, 9));

    for (int i = 0; i < 4; i++) {
      norm.addInputField(wilderness[i] =
        new InputFieldCSV(true,
      Constant.BALANCE_FILE, 10 + i));
    }

    for (int i = 0; i < 40; i++) {
      norm.addInputField(soilType[i] = new InputFieldCSV(true,
        Constant.BALANCE_FILE, 14 + i));
    }

    norm.addInputField(coverType = new InputFieldCSV(
      false, Constant.BALANCE_FILE, 54));

    norm.addOutputField(new OutputFieldRangeMapped(
      inputElevation, 0.1, 0.9));
    norm.addOutputField(new OutputFieldRangeMapped(
      inputAspect, 0.1, 0.9));
    norm.addOutputField(new OutputFieldRangeMapped(
      inputSlope, 0.1, 0.9));
    norm.addOutputField(new OutputFieldRangeMapped(
```

```
        hWater, 0.1, 0.9));
      norm.addOutputField(new OutputFieldRangeMapped(
        vWater, 0.1, 0.9));
      norm.addOutputField(new OutputFieldRangeMapped(
        roadway, 0.1, 0.9));
      norm.addOutputField(new OutputFieldRangeMapped(
        shade9, 0.1, 0.9));
      norm.addOutputField(new OutputFieldRangeMapped(
        shade12, 0.1, 0.9));
      norm.addOutputField(new OutputFieldRangeMapped(
        shade3, 0.1, 0.9));
      norm.addOutputField(new OutputFieldRangeMapped(
        firepoint, 0.1, 0.9));

      for (int i = 0; i < 40; i++) {
        norm.addOutputField(new OutputFieldDirect(soilType[i]));
      }

      if (useOneOf)
        buildOutputOneOf(norm, coverType);
      else
        buildOutputEquilateral(norm, coverType);

    norm.process();
    return norm;
  }

  public void report(int total, int current, String message) {
    System.out.println(current + "/" + total + " " + message);
  }
}
```

The **copy** method is used twice by the first step. It essentially copies one CSV file to another, while segregating away some of the data. This is how the training and evaluation CSV files are created. The **copy** method begins by accepting a source and target file. The **start**, **stop** and **size** parameters are used with an **IndexSampleSegregator**. For more information on the meaning of these three parameters, refer to the description of **IndexSampleSegregator** earlier in this chapter.

```
public void copy(
  File source,
  File target,
```

```
  int start,
  int stop,
  int size)
{
```

First we create an array of input fields to hold the 55 fields that make up the cover type CSV file downloaded earlier in this chapter.

```
  InputField inputField[] = new InputField[55];
```

A **DataNormalization** object is created that reports its progress to the current object, and has a normalization target of a CSV file. This sends the output to the CSV file specified by the **target** parameter.

```
  DataNormalization norm = new DataNormalization();
  norm.setReport(this);
  norm.setTarget(new NormalizationStorageCSV(target));
```

Now we must create all 55 input and output fields. The input fields come from fields in the CSV file, using **InputFieldCSV**. The output fields are all direct copies of the input fields, using **OutputFieldDirect**.

```
  for(int i=0;i<55;i++)
  {
    inputField[i] = new InputFieldCSV(true,source,i);
    norm.addInputField(inputField[i]);
    OutputField outputField =
      new OutputFieldDirect(inputField[i]);
    norm.addOutputField(outputField);
  }
```

Next a segregator is created. It will work on a sample size of **size**. Only indexes, within this sample, that are between **start** and **stop** will be written to the **target** file.

```
  IndexSampleSegregator segregator =
    new IndexSampleSegregator(start,stop,size);
  norm.addSegregator(segregator);

  norm.process();
}
```

Short of segregation, no actual normalization is done by the **copy** methods. The copy method is used by the **step1** method.

```
public void step1()
{
```

Step one generates the training and evaluation files.

```
System.out.println(
  "Step 1: Generate training and evaluation files");
  System.out.println("Generate training file");
```

First, we create the training file. We specify a sample size of four. This breaks the file up into sections of four rows. The first four rows in the file make up the first sample. The second four make up the next sample, and so on. Specifying a start and stop index of zero and two means that of the four-sized sample, we will use index zero, one and two. The third will be left for the evaluation data. As a result, we use 75% of the data for training.

```
 copy(Constant.COVER_TYPE_FILE,Constant.TRAINING_FILE,
  0,2,4);
```

Next we create the evaluation file. We again use a sample size of four, however the starting and stopping index are both three. This means that we will only use the fourth sample (index 3) from each sample.

```
  System.out.println("Generate evaluation file");
  copy(Constant.COVER_TYPE_FILE,Constant.EVALUATE_FILE,
  3,3,4);
```

The result is that we use 25% of the data for evaluation. The narrow method is used by step two to narrow down the files and allow a maximum of 3,000 of each tree type.

```
public void narrow(
  File source,
  File target,
  int field,
  int count) {
```

The **narrow** method accepts the source and target files to use. We also specify the field to narrow on, as well as the maximum count of each of this field to allow. This method begins very similarly to the **copy** method. We create 55 fields to be directly copied from the input field to the output field.

```
  InputField inputField[] = new InputField[55];
  DataNormalization norm = new DataNormalization();
  norm.setReport(this);
  norm.setTarget(new NormalizationStorageCSV(target));
  for(int i=0;i<55;i++) {
    inputField[i] = new InputFieldCSV(true,source,i);
    norm.addInputField(inputField[i]);
    OutputField outputField =
      new OutputFieldDirect(inputField[i]);
    norm.addOutputField(outputField);
  }
```

The **narrow** method differs from the **copy** method, in that an **IntegerBalanceSegregator** is used. This segregator will allow at most **count** items from the specified balancing field.

```
  IntegerBalanceSegregator segregator =
    new IntegerBalanceSegregator(inputField[field],count);

  norm.addSegregator(segregator);
```

The normalization is now performed.

```
  norm.process();
  System.out.println("Samples per tree type:");
  System.out.println(segregator.dumpCounts());
```

The last activity performed by the **narrow** method is to display the counts for each unique value found on the balancing field. Step three will actually normalize the data. This covered in the next section.

Normalizing the Data

The **step3** method normalizes the data. It accepts a parameter to tell whether we are using one-of-n normalization.

```
public DataNormalization step3(boolean useOneOf)
{
  System.out.println("Step 3: Normalize training data");
```

First we must create a number of local variables to use to set up the normalization object.

```
  InputField inputElevation;
```

```
InputField inputAspect;
InputField inputSlope;
InputField hWater;
InputField vWater;
InputField roadway;
InputField shade9;
InputField shade12;
InputField shade3;
InputField firepoint;
InputField[] wilderness = new InputField[4];
InputField[] soilType = new InputField[40];
InputField coverType;
```

The normalization object is created. It will report to the current object, and it will output to a CSV file.

```
DataNormalization norm = new DataNormalization();
norm.setReport(this);
norm.setTarget(
  new NormalizationStorageCSV(Constant.NORMALIZED_FILE));
```

Next we must add all of the input fields from the data file. There are 55 of them. We start with the elevation field, and add in the other simple fields.

```
norm.addInputField(
  inputElevation =
    new InputFieldCSV(true,Constant.BALANCE_FILE,0));
norm.addInputField(
  inputAspect =
    new InputFieldCSV(true,Constant.BALANCE_FILE,1));
norm.addInputField(
  inputSlope =
    new InputFieldCSV(true,Constant.BALANCE_FILE,2));
norm.addInputField(
  hWater =
    new InputFieldCSV(true,Constant.BALANCE_FILE,3));
norm.addInputField(
  vWater =
    new InputFieldCSV(true,Constant.BALANCE_FILE,4));
norm.addInputField(
  roadway =
    new InputFieldCSV(true,Constant.BALANCE_FILE,5));
norm.addInputField(
  shade9 =
    new InputFieldCSV(true,Constant.BALANCE_FILE,6));
norm.addInputField(
```

```
    shade12 =
      new InputFieldCSV(true,Constant.BALANCE_FILE,7));
  norm.addInputField(
    shade3 =
      new InputFieldCSV(true,Constant.BALANCE_FILE,8));
  norm.addInputField(
  firepoint = new InputFieldCSV(true,Constant.BALANCE_FILE,9));
```

Once the initial fields have been added we must add in the wilderness area and soil types. Both of these are arrays of fields. There are four wilderness areas and 44 soil types.

```
  for(int i=0;i<4;i++) {
    norm.addInputField(
      wilderness[i]=
      new InputFieldCSV(true,Constant.BALANCE_FILE,10+i));
  }

  for(int i=0;i<40;i++) {
    norm.addInputField(
      soilType[i]=
        new InputFieldCSV(true,Constant.BALANCE_FILE,14+i));
  }
```

This field is the cover type; it is index 54 in the CSV file. The cover type is what we are attempting to predict.

```
  norm.addInputField(
    coverType=
      new InputFieldCSV(false,Constant.BALANCE_FILE,54));
```

For the initial fields, we will range map them to values between 0.1 and 0.9. The values 0.1 and 0.9 where chosen over 0.0 and 1.0 to prevent the data from being too close to the extreme ends of what the neural network allows. This is 0.0 and 1.0 because this neural network will use a sigmoidal activation function.

```
  norm.addOutputField(
    new OutputFieldRangeMapped(inputElevation,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(inputAspect,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(inputSlope,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(hWater,0.1,0.9));
```

```
  norm.addOutputField(
    new OutputFieldRangeMapped(vWater,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(roadway,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(shade9,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(shade12,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(shade3,0.1,0.9));
  norm.addOutputField(
    new OutputFieldRangeMapped(firepoint,0.1,0.9));
```

The soil types already have vales of 0 or 1, so they can be directly placed into the network. It might be interesting to try values of 0.1 and 0.9 to see how they affect the efficiency of the neural network. However, because these are absolute Boolean values, we placed them at the extremes of 0.0 and 1.0.

```
  for(int i=0;i<40;i++) {
    norm.addOutputField(new OutputFieldDirect(soilType[i]));
  }
```

The cover type is normalized using either equilateral or one-of-n. The methods **buildOuputOneOf** and **buildOutputEquilateral** can be seen in Listing 6.2.

```
  if( useOneOf )
    buildOutputOneOf(norm,coverType);
  else
    buildOutputEquilateral(norm,coverType);
```

Finally, the normalization is performed and the normalization object is returned.

```
  norm.process();
  return norm;
}
```

The training data has now been normalized. The neural network can be trained.

Training the Network

Now that the data is ready, the network must be trained. The code for training is shown in Listing 6.3.

Listing 6.3: The Forest Cover Network Training

```
package org.encog.examples.neural.forest.feedforward;

import org.encog.neural.activation.ActivationSigmoid;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.buffer.BufferedNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.logic.FeedforwardLogic;
import org.encog.normalize.DataNormalization;
import org.encog.persist.EncogPersistedCollection;
import org.encog.util.simple.EncogUtility;

public class TrainNetwork {

  public static BasicNetwork generateNetwork(
  NeuralDataSet trainingSet) {
    BasicNetwork network = new BasicNetwork();
    network.addLayer(new BasicLayer(
      new ActivationSigmoid(), true,
      trainingSet.getInputSize()));
    network.addLayer(new BasicLayer(
      new ActivationSigmoid(), true,
      Constant.HIDDEN_COUNT));
    network.addLayer(new BasicLayer(
      new ActivationSigmoid(), true,
      trainingSet.getIdealSize()));
    network.setLogic(new FeedforwardLogic());
    network.getStructure().finalizeStructure();
    network.reset();
    return network;
  }

  public void train(boolean useGUI) {
    System.out.println("Converting training file to binary");
    EncogPersistedCollection encog =
      new EncogPersistedCollection(
        Constant.TRAINED_NETWORK_FILE);
    DataNormalization norm = (DataNormalization)
      encog.find(Constant.NORMALIZATION_NAME);
```

```
    EncogUtility.convertCSV2Binary(
      Constant.NORMALIZED_FILE,
      Constant.BINARY_FILE, norm.getNetworkInputLayerSize(),
      norm.getNetworkOutputLayerSize(), false);

    BufferedNeuralDataSet trainingSet =
      new BufferedNeuralDataSet(
      Constant.BINARY_FILE);

    BasicNetwork network = (BasicNetwork) encog
      .find(Constant.TRAINED_NETWORK_NAME);
    if (network == null)
      network = EncogUtility.simpleFeedForward(norm
      .getNetworkInputLayerSize(), Constant.HIDDEN_COUNT, 0, norm
      .getNetworkOutputLayerSize(), false);

    if (useGUI) {
      EncogUtility.trainDialog(network, trainingSet);
    } else {
      EncogUtility.trainConsole(network, trainingSet,
      Constant.TRAINING_MINUTES);
    }

    System.out.println("Training complete, saving network...");
    encog.add(Constant.TRAINED_NETWORK_NAME, network);
  }
}
```

The **train** method will actually train the neural network. A parameter is passed in that indicates if GUI training should be used or not.

```
public void train(boolean useGUI) {
```

The network and normalization objects are read from the **forest.eg** file, which is an Encog persistence file.

```
System.out.println("Converting training file to binary");
  EncogPersistedCollection encog =
    new EncogPersistedCollection(Constant.TRAINED_NETWORK_FILE);
  DataNormalization norm =
    (DataNormalization) encog.find(Constant.NORMALIZATION_NAME);
```

The neural network will train from the **normalized.csv** file. Generally, it is a bad idea to train directly from a CSV file. A CSV file contains ASCII

encoded numbers, and these must be passed for each line. It is much better to parse all of the numbers at once, and store them to a binary file. The neural network is then trained from this binary file. To convert the CSV file to a binary file, use the **convertCSV2Binary** method.

```
EncogUtility.convertCSV2Binary(
  Constant.NORMALIZED_FILE,
  Constant.BINARY_FILE,
  norm.getNetworkInputLayerSize(),
  norm.getNetworkOutputLayerSize(),
  false);
```

A **BufferedNeuralDataSet** is then created that will read from the newly created binary file. This allows a file that might be too large and cannot fit in memory to be trained. As the rows are needed, they are read from the file. Also, because it is binary data, no time is wasted reparsing the same rows over and over.

```
BufferedNeuralDataSet trainingSet = new BufferedNeuralDataSet(
  Constant.BINARY_FILE);
```

A new neural network is created, using a utility function. This utility function is a quick way to create a simple feed forward network. The input size, hidden layer one size, hidden layer two size, output layer and activation function type are all passed in. These parameters are passed in this order. The final parameter, which specifies the activation type, uses false for sigmoid and true for hyperbolic tangent.

```
BasicNetwork network = EncogUtility.simpleFeedForward(
  norm.getNetworkInputLayerSize(),
  Constant.HIDDEN_COUNT,
  0,
  norm.getNetworkOutputLayerSize(),
  false);
```

Now that the network has been created, it must be trained. A utility method is also used to train the network. In Chapter 5 we trained by looping and repeatedly calling the iteration method of the trainer. You can also use the **trainConsole** or **trainDialog** methods of **EncogUtility** to perform this training.

```
if( useGUI) {
  EncogUtility.trainDialog(network, trainingSet);
```

```
  } else {
    EncogUtility.trainConsole(network,
      trainingSet,
    Constant.TRAINING_MINUTES);
  }
```

You can train using console mode only by calling the **trainConsole** method. You can also use the GUI to train. Using the GUI displays the dialog seen in Figure 6.1.

```
  System.out.println(
    "Training complete, saving network...");
  encog.add(Constant.TRAINED_NETWORK_NAME, network);
```

Once the network has been trained, it is saved to an Encog persistence file.

Evaluating the Network

Now that the network has been trained, it can be evaluated. The code used to evaluate the neural network is shown in Listing 6.4.

Listing 6.4: The Forest Cover Network Evaluation

```
package org.encog.examples.neural.forest.feedforward;

import java.io.File;
import org.encog.neural.data.NeuralData;
import org.encog.neural.networks.BasicNetwork;
import org.encog.normalize.DataNormalization;
import org.encog.normalize.output.nominal.OutputEquilateral;
import org.encog.persist.EncogPersistedCollection;
import org.encog.util.Format;
import org.encog.util.csv.ReadCSV;

public class Evaluate {

  private int[] treeCount = new int[10];
  private int[] treeCorrect = new int[10];

  public void keepScore(int actual, int ideal) {
    treeCount[ideal]++;
    if (actual == ideal)
      treeCorrect[ideal]++;
    }

  public BasicNetwork loadNetwork() {
```

```
    File file = Constant.TRAINED_NETWORK_FILE;

    if (!file.exists()) {
      System.out.println(
        "Can't read file: " + file.getAbsolutePath());
      return null;
    }

    EncogPersistedCollection encog =
      new EncogPersistedCollection(file);
    BasicNetwork network = (BasicNetwork) encog
      .find(Constant.TRAINED_NETWORK_NAME);

    if (network == null) {
      System.out.println("Can't find network resource: "
        + Constant.TRAINED_NETWORK_NAME);
      return null;
    }

    return network;
  }

  public DataNormalization loadNormalization() {
    File file = Constant.TRAINED_NETWORK_FILE;

    EncogPersistedCollection encog =
      new EncogPersistedCollection(file);

    DataNormalization norm = (DataNormalization) encog
      .find(Constant.NORMALIZATION_NAME);
    if (norm == null) {
      System.out.println("Can't find normalization resource: "
      + Constant.NORMALIZATION_NAME);
      return null;
    }

  return norm;
  }

  public int determineTreeType(
    OutputEquilateral eqField,
    NeuralData output) {

    int result = 0;

    if (eqField != null) {
```

```
      result = eqField.getEquilateral().decode(output.getData());
    } else {
      double maxOutput = Double.NEGATIVE_INFINITY;
      result = -1;

      for (int i = 0; i < output.size(); i++) {
        if (output.getData(i) > maxOutput) {
          maxOutput = output.getData(i);
          result = i;
        }
      }
    }
  return result;
  }

  public void evaluate() {
    BasicNetwork network = loadNetwork();
    DataNormalization norm = loadNormalization();

    ReadCSV csv = new ReadCSV(
      Constant.EVALUATE_FILE.toString(), false, ',');
      double[] input = new double[norm.getInputFields().size()];
      OutputEquilateral eqField = (OutputEquilateral)
        norm.findOutputField(
        OutputEquilateral.class, 0);

      int correct = 0;
      int total = 0;

      while (csv.next()) {
        total++;
        for (int i = 0; i < input.length; i++) {
          input[i] = csv.getDouble(i);
      }

      NeuralData inputData = norm.buildForNetworkInput(input);
      NeuralData output = network.compute(inputData);
      int coverTypeActual = determineTreeType(eqField, output);
      int coverTypeIdeal = (int) csv.getDouble(54) - 1;

      keepScore(coverTypeActual, coverTypeIdeal);

      if (coverTypeActual == coverTypeIdeal) {
        correct++;
      }
    }
```

```
    System.out.println("Total cases:" + total);
    System.out.println("Correct cases:" + correct);
    double percent = (double) correct / (double) total;
    System.out.println("Correct percent:"
      + Format.formatPercentWhole(percent));
    for (int i = 0; i < 7; i++) {
      double p = ((double) this.treeCorrect[i]
      / (double) this.treeCount[i]);
      System.out.println("Tree Type #" + i + " - Correct/total: "
      + this.treeCorrect[i] + "/" + treeCount[i] + "("
      + Format.formatPercentWhole(p) + ")");
    }
  }
}
```

The **evaluate** method is called to evaluate the neural network. It begins by loading the network and the normalization objects. Both of these objects are read from the Encog persistence file.

```
  public void evaluate() {
    BasicNetwork network = loadNetwork();
    DataNormalization norm = loadNormalization();
```

We will now open the evaluation file for use with a **ReadCSV** object. The contents of the evaluation file will be read and the network evaluated. For evaluation we will only pass over the contents of the file once, so it is okay to read a CSV file without converting it to binary, as we did earlier.

```
ReadCSV csv =
  new ReadCSV(Constant.EVALUATE_FILE.toString(),false,',');
```

Next, we read in all of the input fields into a **double** array that will be presented to the neural network.

```
double[] input = new double[norm.getInputFields().size()];
```

We also obtain the **OutputEquilateral** object from the normalization object. We will use the **OutputEquilateral** to interpret the results of the neural network and see which actual tree type the neural network is predicting.

```
OutputEquilateral eqField =
  (OutputEquilateral)norm.findOutputField(
```

```
  OutputEquilateral.class, 0);
```

We now loop over every row in the CSV file. We will count the number of **correct** records, as well as the **total** number of records.

```
  int correct = 0;
  int total = 0;
  while(csv.next()) {
    total++;
```

The input for the neural network is read right from the CSV file and loaded into an array to be normalized.

```
    for(int i=0;i<input.length;i++) {
      input[i] = csv.getDouble(i);
    }
```

Next, the normalization object is used to normalize the input data.

```
    NeuralData inputData = norm.buildForNetworkInput(input);
```

The data is presented to the neural network. The output will tell us what tree type the network predicted for the input data.

```
    NeuralData output = network.compute(inputData);
```

The neural network was trained with the tree type converted to an equilateral normalized array. As a result, the output from the neural network is normalized and must be converted to an actual tree number. The **determineTreeType** takes the equilateral normalized output from the neural network and converts it to a tree type number. We also get the ideal tree type(that should have been predicted) from the CSV file.

```
    int coverTypeActual = determineTreeType(eqField,output);
    int coverTypeIdeal = (int)csv.getDouble(54)-1;
```

The **keepScore** method is a very simple method that keeps track of correct guesses by the neural network on a tree type basis. This allows us to see that the neural network is better at predicting some tree types than others.

```
    keepScore(coverTypeActual,coverTypeIdeal);
```

We also keep track of the overall correct count.

```
    if( coverTypeActual==coverTypeIdeal ) {
      correct++;
    }
  }
```

Finally, we display the statistics of how well the evaluation went.

```
  System.out.println("Total cases:" + total);
  System.out.println("Correct cases:" + correct);
  double percent = (double)correct/(double)total;
  System.out.println("Correct percent:"
    + Format.formatPercentWhole(percent));
  for(int i=0;i<7;i++)
  {
    double p = ((double)this.treeCorrect[i]
      / (double)this.treeCount[i]);
    System.out.println("Tree Type #"
      + i
      + " - Correct/total: "
      + this.treeCorrect[i]
      + "/"
      + treeCount[i] + "("
      + Format.formatPercentWhole(p) + ")" );
  }
}
```

We will now examine how the **determineTreeType** method converts an equilateral output from the neural network into an actual tree number. The **determineTreeType** method is passed both the equilateral object, from the normalization object, as well as the neural network output.

```
public int determineTreeType(
  OutputEquilateral eqField,
  NeuralData output)
      {
```

We are going to loop over all of the equilateral encodings for each of the seven tree types, held in the equilateral object. Whichever one has the lowest Euclidean distance to the neural network output is considered to be the tree type that the neural network predicted.

```
  int result = 0;
```

First, we see if equilateral normalization was used. If it was, simply use the **decode** method. This method will determine which tree type had the lowest equilateral distance.

```
  if( eqField!=null ) {
    result = eqField.getEquilateral().decode(output.getData());
  } else {
```

For one-of-n encoding, we loop over all of the output neurons and see which has the highest activation. The neuron with the highest activation corresponds to the tree type that was predicted.

```
    double maxOutput = Double.NEGATIVE_INFINITY;
    result = -1;

    for(int i=0;i<output.size();i++) {
      if( output.getData(i)>maxOutput ) {
        maxOutput = output.getData(i);
        result = i;
      }
   }
}
```

Finally, return the **result**.

```
return result;
```

The forest example is a very good starting point for creating an Encog-based application that classifies input data into specific groups. In this case, the trees were these groups. Any such application will have to go through the process of generating data, training and evaluation.

Summary

In this chapter you saw how to normalize data for a neural network. Neural networks can very rarely handle data in a raw form. To normalize the data you restrict and map the data into specific ranges. You also convert nominal data into arrays of values using either equilateral or one-of-n encoding.

Encog provides the **NormalizeData** class to make normalization easier. This class supports a variety of normalization types. This class makes use of

InputField, **OutputField**, **Segregator**, and **NormalizationStorage** classes. Using subclasses of these four basic classes, many different normalization techniques can be achieved.

InputField derived objects are added to the **NormalizeData** class to define where raw input data should be come from. CSV files are a very common choice, as they are a convenient means of storing many rows of numeric data. There are also input fields defined for arrays and **NeuralDataSet** objects.

OutputField-derived objects are added to the **NormalizeData** class to define how the input fields should be normalized. Encog supports many different normalization types. There are normalization types for multiplicative, z-axis, one-of-n, equilateral and ranged mapped normalization techniques.

Segregator-derived objects are added to the **NormalizeData** class to define which rows should not be processed. There are many different segregators available. You can choose to take a sample for training or evaluation. You can also exclude rows based on the values of their fields. Rows can be removed to maintain balance, and prevent one row type from saturating the network.

A single **NormalizationStorage** derived object tells the **NormalizeData** object what to do with the normalized data. Data can be written to a variety of targets, such as arrays, CSV files and **NeuralDataSet** objects.

This chapter demonstrated an example that attempts to predict the type of tree that may cover wilderness area. This example used real-world data provided by the United States Forestry Service. This example demonstrated how to normalize this raw data, and predict forest cover. This example demonstrated many common techniques in neural network programming, such as normalization, training and evaluation.

This chapter also introduced Encog persistence files. These files can contain different types of Encog objects. These files are very useful because it can take days to properly train a neural network. The next chapter will expand on Encog persistence files and show how they are used.

Questions for Review

1. Why is it necessary to normalize data for neural networks?
2. Describe the purpose of each of the following Encog normalization object types: input fields, output fields, segregators and output targets.
3. When is it necessary for Encog to do a "two-pass" normalization process?
4. What is the difference between z-axis normalization and multiplicative normalization? When would you use each?
5. What is the difference between one-of-n normalization and equilateral normalization?
6. Given a type of gender, which has the values male/female? Would you use one-of-n or equilateral normalization? Why?
7. What are balance segregators used for?
8. Training data read directly from CSV files can be slow, as the data must be reparsed for each iteration. How can an Encog application overcome this?
9. Describe what happens in each of these phases of a typical neural network program: generation, training, and evaluation.
10. What advantage does a two-dimensional array have over a one-dimensional array for output storage?

Terms

CSV

EG File

Equilateral Normalization

Euclidian Distance

Evaluation

Field Group

Input Field

Multiplicative Normalization

Nominal Value

Normalization

Normalization Target

Numeric Value

one-of-n Normalization

Output Field

Segregator

Training

Vector Length

Z-Axis Normalization

Chapter 7: Encog Persistence

- Encog XML Persistence
- Encog Serialization
- Encog XML Format

It can take considerable time to train a neural network. You do not want to lose all of this work once the network has been trained. Encog provides several means for this data to be saved. There are two primary ways that you can store Encog data objects. You can use Encog XML persistence, or you can use Java's own persistence.

Java provides its own means to serialize objects, known as Java serialization. Java Serialization allows many different object types to be written to a stream, such as a disk file. Java Serialization for Encog works the same way that you would save any Java object using Java Serialization. Every important Encog object that should support serialization implements the **Serializable** interface.

Java Serialization is a quick way to store an Encog object. However, it has some important limitations. The files that you create with Java Serialization can only be used by Encog for Java. They will be incompatible with Encog for .Net or Encog for Silverlight. Further, Java Serialization is directly tied to the underlying objects. As a result, future versions of Encog may not be compatible with your serialized files.

To create universal files that will work with all Encog platforms you should consider the Encog XML format. Extensible Markup Language (XML) is a standard way to represent data in a human-readable text file. There are many tools for processing XML. Encog XML files, which end in the extension .EG, are stored in a very human readable XML format. It is intended that .EG files could be used by programs other than Encog, as they are relatively simple XML files. Further, Encog .EG files are interchangeable between Encog for Java and Encog for .Net.

This chapter will introduce both methods of Encog persistence. We will begin with Encog XML persistence. The chapter will end by exploring how a neural network is saved in an Encog persistence file.

Using Encog XML Persistence

Encog XML persistence files are the native file format for Encog. They are stored with the extension .EG, and are often called Encog EG files. The Encog EG format is the format in which the Encog Workbench processes files. This format can be exchanged over different operating systems and Encog platforms. The Encog EG format should be the format of choice for an Encog application.

We will begin this section by looking at an XOR example that makes use of Encog's EG files. In a later section we will examine this same example and see how it would make use of Java Serialization. We will begin with the Encog EG persistence example.

Using Encog EG Persistence

It is very easy to use Encog EG XML persistence. The **EncogPersistedCollection** class is used to load and save objects from an Encog EG file. Listing 7.1 shows an example that trains an XOR network and then persists the trained neural network to an Encog EG file.

Listing 7.1: Encog XML Persistence

```
package org.encog.examples.neural.persist;

import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.resilient.
  ResilientPropagation;
import org.encog.persist.EncogPersistedCollection;
import org.encog.util.logging.Logging;

public class EncogPersistence {

  public static final String FILENAME = "encogexample.eg";

  public static double XOR_INPUT[][] = {
    { 0.0, 0.0 },
    { 1.0, 0.0 },
    { 0.0, 1.0 },
```

```
    { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
    { 0.0 },
    { 1.0 },
    { 1.0 },
    { 0.0 } };

  public void trainAndSave() {
    System.out.println(
      "Training XOR network to under 1% error rate.");
    BasicNetwork network = new BasicNetwork();
    network.addLayer(new BasicLayer(2));
    network.addLayer(new BasicLayer(2));
    network.addLayer(new BasicLayer(1));
    network.getStructure().finalizeStructure();
    network.reset();

    NeuralDataSet trainingSet =
      new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

    // train the neural network
    final Train train =
      new ResilientPropagation(network, trainingSet);

    do {
      train.iteration();
    } while (train.getError() > 0.009);

    double e = network.calculateError(trainingSet);
    System.out.println("Network traiined to error: " + e);

    System.out.println("Saving network");
    final EncogPersistedCollection encog =
      new EncogPersistedCollection(FILENAME);
    encog.create();
    encog.add("network", network);
  }

  public void loadAndEvaluate() {
    System.out.println("Loading network");

    final EncogPersistedCollection encog =
      new EncogPersistedCollection(FILENAME);

    BasicNetwork network = (BasicNetwork) encog.find("network");
```

```
    NeuralDataSet trainingSet = new BasicNeuralDataSet(
      XOR_INPUT, XOR_IDEAL);
    double e = network.calculateError(trainingSet);
    System.out.println(
      "Loaded network's error is(should be same as above): "+ e);
  }

  public static void main(String[] args) {
    Logging.stopConsoleLogging();

  try {
    EncogPersistence program = new EncogPersistence();
    program.trainAndSave();
    program.loadAndEvaluate();
  } catch (Throwable t) {
    t.printStackTrace();
  }
 }
}
```

This example is made up of two primary methods. The first method, named **trainAndSave** trains a neural network and then saves it to an Encog EG file. The second method, named **loadAndEvaluate,** loads the Encog EG file and evaluates it. This proves that the Encog EG file was saved correctly. The **main** method simply calls these two in sequence. We will begin by examining the **trainAndSave** method.

```
public void trainAndSave() {
  System.out.println(
    "Training XOR network to under 1% error rate.");
```

This method begins by creating a basic neural network to be trained with the XOR operator. It is a simple three layer feedforward neural network.

```
  BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(2));
  network.addLayer(new BasicLayer(6));
  network.addLayer(new BasicLayer(1));
  network.getStructure().finalizeStructure();
  network.reset();
```

A training set is created that contains the expected outputs and inputs for the XOR operator.

```
NeuralDataSet trainingSet =
  new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);
```

We will train this neural network using resilient propagation (RPROP).

```
// train the neural network
final Train train =
  new ResilientPropagation(network, trainingSet);
```

We will perform RPROP iterations until the error rate is very small.

```
  do {
    train.iteration();
  } while(train.getError() > 0.009);
```

Once the network has been trained, display the final error rate. We can now save the neural network.

```
  double e = network.calculateError(trainingSet);
  System.out.println("Network traiined to error: "
  + e);
  System.out.println("Saving network");
```

An Encog **PersistedCollection** object is created with the specified filename.

```
  final EncogPersistedCollection encog =
    new EncogPersistedCollection(FILENAME);
```

We wish to create a new Encog EG file, so the **create** method is called. This will also overwrite any Encog EG file with the same name. If you do not call the **create** method, objects already stored in the file will remain. Encog EG files are built to allow resources to be added and removed without disturbing objects already in the file.

```
  encog.create();
```

Adding an object to the Encog EG file is done with the **add** method. The quoted name "network" specifies the resource name for the object we are adding. If there is already an object named "network", then it will be overwritten.

```
    encog.add("network", network);
```

At this point the network has been added to the Encog EG files. There is no need to call any sort of a final "save" command. Because Encog EG files can grow very large, the **EncogPersistedCollection** does not read the entire file into memory at once. Only the object actually being saved or loaded is in memory.

Now that the Encog EG file has been created, we should load the neural network back from the file and see if it still performs well. This is performed by the **loadAndEvaluate** method.

```
public void loadAndEvaluate()
{
  System.out.println("Loading network");
```

First, a new **EncogPersistedCollection** is created for the specified file name. This follows exactly the same pattern as saving an object to an **EncogPersistedCollection**.

```
  final EncogPersistedCollection encog =
    new EncogPersistedCollection(FILENAME);
```

Now that the collection has been constructed we must find the network that is named "network". This neural network was saved earlier.

```
  BasicNetwork network = (BasicNetwork)encog.find("network");
```

We would like to evaluate the neural network to prove that it is still trained. To do this we create a training set for the XOR operator.

```
  NeuralDataSet trainingSet =
    new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);
```

We now calculate the error for the given training data.

```
  double e = network.calculateError(trainingSet);
  System.out.println(
    "Loaded network's error is(should be same as above): " + e);
}
```

This error is displayed. It should be the same as before the network was saved.

Encog EG-Compatible Objects

Any class that implements the **EncogPersistedObject** interface can be saved to an Encog EG file. There are a number of useful objects that persist and are already built into Encog. These objects will be discussed in the next section. The general form for saving any Encog object is as follows.

```
encogCollection.add("resource_name", encogObject);
```

The general form for loading an Encog object is as follows:

```
BasicNetwork network =
  (BasicNetwork)encog.find("resource_name");
```

This code assumed that you were using a **BasicNetwork**. Simply replace **BasicNetwork** with the class type you wish to load.

Using BasicNetwork Objects

BasicNetwork objects can be saved to an Encog EG file. These are the neural networks used by Encog. Examples of loading and saving **BasicNetwork** objects were given earlier in this chapter. Every Encog persistable object will follow this same format.

Using NeuralDataSet Objects

The **BasicNeuralDataSet** object holds a training set in memory. This is the only type of dataset that can be persisted to Encog. If you attempt to persist any other **NeuralDataSet** derived class, the class will be loaded into memory first and then converted to a **BasicNeuralData** set object. For example, if you persisted a SQL dataset, the connection information would not be persisted. Rather the data would be exported from the SQL database and saved.

Using PropertyData Objects

It might be useful to save configuration information to an Encog EG file. The **PropertyData** object works something like a hash map, in that it can store a series of name-value pairs. You can create these name value pairs in a **PropertyData** object and store it in your EG files. Encog does not use

PropertyData objects directly; they are simply ways for you to store configuration information that your program might need.

Using TextData Objects

Text files can also be stored inside of Encog. The **TextData** allows a string to be stored in the Encog EG file. This string can be long. It is often used to attach "readme.txt" type information to an Encog .EG file.

Other Useful EG File Operations

There are other useful classes and methods that can be used with the Encog persisted collection class. These will be covered in the following sections.

Memory-Based Encog Collections

The **EncogMemoryCollection** works similarly to a Java Collection object. It allows the entire contents of an Encog EG file to be loaded into memory. Using the regular **EncogPersistedCollection** object, which was just discussed, will allow your application to support very large Encog EG files. However, for short files it may be easier to simply load and save the entire file at once.

The **EncogMemoryCollection** provides **load** and **save** methods. The **load** method will read the entire contents of an Encog EG file into memory. The **save** method will save the contents of the collection to an Encog EG file on disk.

Listing the Contents of an Encog EG File

Calling the **buildDirectory** function on the **EncogPersistedCollection** will obtain a list of the resources contained in the Encog EG file. Using these **DirectoryEntry** objects you can see what the names and types of all of the Encog resources in the specified file are.

Using Java Serialization

It is also possible to use standard Java Serialization with Encog neural networks and training sets. Listing 7.2 shows the same example from the last section, except that it uses Java Serialization.

Listing 7.2: Encog Java Serialization

```
package org.encog.examples.neural.persist;

import java.io.IOException;

import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.
  resilient.ResilientPropagation;
import org.encog.util.SerializeObject;
import org.encog.util.logging.Logging;

public class Serial {

  public static final String FILENAME = "encogexample.ser";

  public static double XOR_INPUT[][] = {
    { 0.0, 0.0 },
    { 1.0, 0.0 },
    { 0.0, 1.0 },
    { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
    { 0.0 },
    { 1.0 },
    { 1.0 },
    { 0.0 } };

  public void trainAndSave() throws IOException {
  System.out.println(
    "Training XOR network to under 1% error rate.");
  BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(2));
  network.addLayer(new BasicLayer(2));
  network.addLayer(new BasicLayer(1));
  network.getStructure().finalizeStructure();
```

```
  network.reset();

  NeuralDataSet trainingSet =
    new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);

  // train the neural network
  final Train train =
    new ResilientPropagation(network, trainingSet);

  do {
    train.iteration();
  } while (train.getError() > 0.009);

  double e = network.calculateError(trainingSet);
  System.out.println("Network traiined to error: " + e);

  System.out.println("Saving network");
  SerializeObject.save(FILENAME, network);
  }

  public void loadAndEvaluate()
    throws IOException, ClassNotFoundException {
    System.out.println("Loading network");
    BasicNetwork network = (BasicNetwork)
    SerializeObject.load(FILENAME);
    NeuralDataSet trainingSet = new BasicNeuralDataSet(
      XOR_INPUT, XOR_IDEAL);
    double e = network.calculateError(trainingSet);
    System.out.println(
      "Loaded network's error is(should be same as above): "+ e);
  }

  public static void main(String[] args) {
    Logging.stopConsoleLogging();

    try {
      Serial program = new Serial();
      program.trainAndSave();
      program.loadAndEvaluate();
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

Encog XML persistence is much more flexible than Java Serialization. However, there are cases where you may want to simply save a neural network to a platform-dependant binary file. This example shows you how to use Java Serialization with Encog. The example begins by calling the **trainAndSave** method.

```
public void trainAndSave() throws IOException {
  System.out.println(
    "Training XOR network to under 1% error rate.");
```

This method begins by creating a basic neural network to be trained with the XOR operator. It is a simple, three layer feedforward neural network.

```
  BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(2));
  network.addLayer(new BasicLayer(6));
  network.addLayer(new BasicLayer(1));
  network.getStructure().finalizeStructure();
  network.reset();

  NeuralDataSet trainingSet =
    new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);
```

We will train this neural network using resilient propagation (RPROP).

```
  // train the neural network
  final Train train =
    new ResilientPropagation(network, trainingSet);
```

The following code loops through training iterations until the error rate is below one percent (<0.01).

```
  do {
    train.iteration();
  } while(train.getError() > 0.01);
```

The final error for the neural network is displayed.

```
  double e = network.calculateError(trainingSet);
  System.out.println("Network traiined to error: " + e);

  System.out.println("Saving network");
```

You can use regular Java Serialization code to save the network, or you can use the **SerializeObject** class. This utility class provides a **save** method that will write any single serializable object to a binary file. Here the **save** method is used to save the neural network.

```
  SerializeObject.save(FILENAME, network);
}
```

Now that the binary serialization file has been created, we should load the neural network back from the file and see if it still performs well. This is performed by the **loadAndEvaluate** method.

```
public void loadAndEvaluate()
  throws IOException, ClassNotFoundException {

  System.out.println("Loading network");
```

The **SerializeObject** class also provides a **load** method that will read an object back from a binary serialization file.

```
  BasicNetwork network =
    (BasicNetwork) SerializeObject.load(FILENAME);
  NeuralDataSet trainingSet =
    new BasicNeuralDataSet(XOR_INPUT, XOR_IDEAL);
```

Now that the network is loaded, we report the error level.

```
  double e = network.calculateError(trainingSet);
  System.out.println(
    "Loaded network's error is(should be same as above): " + e);
}
```

This error level should match the one with which the network was originally trained.

Format of the Encog XML Persistence File

The Encog EG file has a very specific XML file format. Any program can modify the Encog EG file or read data from it. It is only necessary to understand its format. This section will show the general format of the Encog EG file, and show how neural networks and training data are stored to an Encog EG file. Listing 7.3 shows an empty Encog EG file.

Listing 7.3: An Empty Encog EG File

```
<Document>
  <Header>
    <platform>Java</platform>
    <fileVersion>1</fileVersion>
    <encogVersion>2.3.0</encogVersion>
    <modified>Wed Nov 25 12:38:16 CST 2009</modified>
  </Header>
  <Objects></Objects>
</Document>
```

The above file shows the minimum elements that are necessary for an Encog EG file. Encog EG files consist of two parts. First is the header. The header stores information about the file. The platform specifies what platform created this file. Encog EG files are meant to be interchangeable. Therefore the platform is stored only for tracking purposes. The file version specifies the file format. If the Encog file format is ever updated to the point that it is no longer backward compatible, this number will be incremented. As of Encog version 2.3, the **fileVersion** element is set to one. The Encog version that created this file is also kept, for informational purposes. Finally, the header element also stores the last modified date. This date is stored in a platform dependant way and is not used by Encog.

After the Header is the objects collection. Encog objects will be stored inside of the objects collection. In this section we will see how two different Encog objects are stored and how neural networks and training data are stored. We will begin with how a neural network is stored.

How a Neural Network is Stored

The Encog **BasicNetwork** class is designed to be very flexible. It can contain many different forms properties, layers, layer tags and other objects. The format that the **BasicNetwork** is saved to XML as it does not exactly match the in-memory object. This allows some flexibility for upgrades to Encog and does not break the file format.

This section is not meant to be an exhaustive definition of the Encog EG file format. Rather, it will highlight some of the more commonly used Encog objects, which should give you an idea of how Encog EG files are structured.

We will examine a neural network that has been trained for the XOR operator, as well as the training data for the XOR operator. The **BasicNetwork** XML tag signals the beginning of a saved **BasicNetwork** object. You can see the beginning of this tag here.

```
<BasicNetwork
  id="1"
  native="org.encog.neural.networks.BasicNetwork" description=""
  name="network-1">
```

All Encog objects begin in a very similar way. You can see the **id** listed above. This is a local id to this object. All Encog objects start with an ID of one. If there were a second **BasicNetwork** stored, it too would start with one, as well. The **id** tag is only useful to reference inside of an Encog objects. Any sub-objects, inside of the **BasicNetwork** that might need to be referenced would be assigned successively higher **id** values. The **BasicNetwork** does not need to internally reference sub-objects, so the id values are not really used. However, all Encog objects will start with an **id** of one, and **BasicNetwork** is no exception.

The **native** tag specifies the actual name of the class, on the platform that encoded this object. The native attribute is platform dependant; however, it is rarely used. It simply provides a "fall back" when the object name is unknown.

The **name** attribute is very important. The **name** attribute is how Encog finds the resource type that you are trying to access. No two resources should have the same name.

Next, there is a **layers** list. This list stores the layers of the neural network. Each layer is enclosed in a **layer** tag; they have an ID, which will be used to link the layers together.

```
<layers>
  <layer id="1">
```

This layer contains a **BasicLayer**. This is the most common Encog layer type; however, you may also see **ContextLayer** and **RadialBasisFunctionLayer** layers, as well as others, because Encog is

further enhanced. Chapter 2, "The Parts of an Encog Neural Network", discussed the various layer types that Encog can use.

```
    <BasicLayer
      id="1"
      native="org.encog.neural.networks.layers.BasicLayer">
```

Next, some information about the layer is provided. For this layer there are two neurons. The x and y coordinates are both 50. These coordinates are not used by Encog for any type of neural network processing. However, the Encog Workbench does make use of them to determine where to display the layer.

```
        <neurons>2</neurons>
        <x>50</x>
        <y>50</y>
```

Thresholds, along with weights, allow the neural network to learn. The threshold values are stored in the following list:

```
        <threshold>
          -0.5804767997917228,
          0.9035488954678172
        </threshold>
```

Some layers make use of activation functions. This layer uses a sigmoid activation function.

```
        <activation>
          <ActivationSigmoid></ActivationSigmoid>
        </activation>
      </BasicLayer>
```

This completes this layer.

```
    </layer>
```

However, there are still two more layers to the neural network. The layer that was just seen was the input layer. There is also a hidden layer and an output layer. These layers are very similar to the input layer in their XML format, so they will not be shown.

```
  </layers>
```

The layers have now been described. Now the XML document must describe the synapses.

The first synapse is a connection from the neural layer number two to the neuron layer number one.

```
  <synapses>
    <synapse to="2" from="1">
```

This synapse is a **WeightedSynapse**. There are also **WeightlessSynapse** objects and **DirectSynapse** objects. They all have a very similar format. Chapter 2 discussed the various layer types and synapses that can make up an Encog neural network.

```
      <WeightedSynapse
        id="1"
        native=
        "org.encog.neural.networks.synapse.WeightedSynapse">
```

The weight matrix is stored inside of the weights tag. The weight matrix holds the weights between each of the neurons on either side of this connection. Each row holds the connections from one of the two neurons in the source layer to one of the three neurons in the target layer.

```
        <weights>
          <Matrix cols="3" rows="2">
            <row>-27.56342234018966,
                 -47.032368713399975,
                 -46.32991712253923
            </row>
            <row>64.8824309191636,
                  3.00755949829821,
                -41.12878494573593</row>
            </Matrix>
         </weights>
      </WeightedSynapse>
    </synapse>
  </synapses>
```

Some neural networks contain properties. A simple feedforward neural network does not. Properties are simply name-value pairs. This allows important configuration information to be held for certain types of neural networks. Properties were covered in Chapter 2.

```
  <properties>
  </properties>
```

Tags name different layers. Some neural networks require the layers to have very specific tag names. Most neural networks will have an **INPUT** and an **OUTPUT** layer. The following tags demonstrate this concept:

```
  <tags>
    <tag name="INPUT" layer="1"/>
    <tag name="OUTPUT" layer="3"/>
  </tags>
```

How the layers, weights and thresholds are computed is determined by the neural logic class that is used with the neural network. This neural network uses simple recurrent logic, which is the default for Encog. Other neural logic types were covered in Chapter 2.

```
<logic>SimpleRecurrentLogic</logic>
</BasicNetwork>
```

This section demonstrated how a feedforward neural network, trained for the XOR operator, was stored. This is by no means an exhaustive description of how every neural network type could be saved in Encog. Most of the additional layers and synapse types persist very similar to what you saw in this section. Perhaps one of the best ways to see how other layer and synapse types are stored is simply to model other neural networks using the Encog Workbench, and examine the source code.

How Training Data is Stored

Training data can also be stored in an Encog EG file. The format for training data is very simple. The training data begins with a standard Encog XML start tag, as seen here.

```
<TrainingData id="1"
  native=
  "org.encog.neural.data.basic.BasicNeuralDataSet"
  description="Training data"
  name="data-1">
```

The training data is made up of training items. Each training item has an input and ideal tag. If the training data is supervised, both **input** and

ideal will contain valid values. Unsupervised training data does not contain any value for the ideal tag.

```
  <Item>
    <Input>0,0</Input>
    <Ideal>0</Ideal>
  </Item>
```

The rest of the XOR training items are handled similarly.

```
  <Item>
    <Input>1,0</Input>
    <Ideal>1</Ideal>
  </Item>
  <Item>
    <Input>0,1</Input>
    <Ideal>1</Ideal>
  </Item>
  <Item>
    <Input>1,1</Input>
    <Ideal>0</Ideal>
  </Item>
</TrainingData>
```

You have now seen how to store a neural network and training data. This demonstrates the basis of storing data in an Encog EG file. Most other data types in Encog follow a similar pattern. The best way to see how an Encog object is stored is to model it in the Encog Workbench, and then examine the XML file.

Summary

In this chapter you saw how to persist Encog objects. Encog provides two methods by which you can persist objects. Objects may be persisted by using either the Encog EG XML format, or by using Java Serialization.

The Encog EG XML format is the preferred means for saving Encog neural networks. The EG file can contain several Encog objects. These objects are accessed using their resource name. The EG file can be interchanged between any platform that Encog supports.

Encog also allows you to use Java Serialization to store your objects to disk or a stream. Java Serialization is more restrictive than Encog EG files.

Because the binary files are automatically stored directly from the objects, even the smallest change to one of the Encog objects can result in incompatible files. Additionally, you could not allow other platforms to use your file.

So far the neural networks that have been created have been trained using propagation training. Propagation training is not the only form of neural network training. In the next chapter we will see how simulated annealing and genetic algorithms can be used to train a neural network.

Questions for Review

1. What is the preferred way to save Encog objects?
2. What information is stored in the header of an EG file?
3. What class do you use if you would like to read the entire contents of an EG file into memory?
4. Which method do you use to read an object from an **EncogPersistedCollection**?
5. What happens if you attempt to add an object to an Encog EG file and the file already contains an object with the same name?
6. What is the base class for any object that can be stored using the **EncogPersistedCollection**?
7. What Encog object would allow you to store property or configuration information to an EG file?
8. What Encog object would allow you to save a text file as part of an EG file?
9. Is an Encog object immediately saved to disk when the **add** method of **EncogPersistedCollection** is called? If not, when is it actually saved?
10. If you would like Encog Java, as well as Encog .Net to read your network from the same file, how should you persist this network?

Terms

Memory Collection

Persistence

Serializable

Chapter 8: More Supervised Training

- Introducing the Lunar Lander Example
- Supervised Training without Training Sets
- Using Genetic Algorithms
- Using Simulated Annealing
- Genetic Algorithms and Simulated Annealing with Training Sets

So far, the only means by which we trained a neural network has been by using the supervised propagation training methods. In this chapter, we will look at some nonpropagation training techniques. The neural network in this chapter will be trained without a training set. It is still supervised, in that feedback from the neural network's output is constantly used to help train the neural network. We simply will not supply training data ahead of time.

Two common techniques for this sort of training are simulated annealing and genetic algorithms. Encog provides built-in support for both. The example in this chapter can be trained with either algorithm, and both algorithms will be discussed later in this chapter.

The example in this chapter presents the classic "Lunar Lander" game. This game has been implemented many times and is almost as old as computers themselves. You can read more about the Lunar Lander game on Wikipedia.

http://en.wikipedia.org/wiki/Lunar_Lander_%28computer_game%29

The idea behind most variants of the Lunar Lander game is very similar. The example program presented here works as follows. The lunar lander spacecraft will begin to fall. As it falls, it accelerates. There is a maximum velocity that the lander can reach, which is called the 'terminal velocity'. Thrusters can be applied to the lander to slow its descent. However, there is a limited amount of fuel. Once the fuel is exhausted, the lander will simply fall, and nothing can be done.

In this chapter, we are going to teach a neural network to pilot the lander. This is a very simple text-only simulation. The neural network will have only one option available to it. It can either decide to fire the thrusters or not to fire the thrusters. No training data will be created ahead of time. No assumptions will be made about how the neural network should pilot the craft. If we were using training sets, we would be inputting, ahead of time,

what we feel that the neural network should do in certain situations. For this example we want the neural network to learn everything on its own.

However, this is still supervised training. We will not leave the neural network totally to its own devices. We will provide a way to score the neural network. To score the neural network we must give it some goals, and then calculate a numeric value that determines how well the neural network achieved its goals.

These goals are arbitrary, and simply reflect what was picked to score the network. The goals are summarized here.

- Land as softly as possible
- Cover as much distance as possible
- Conserve fuel

The first goal is not to crash, but to try to hit the lunar surface as softly as possible. Therefore, any velocity, at the time of impact, is a very big negative score. The second goal for the neural network is to try to cover as much distance as possible, while falling. To do this, it needs to stay aloft as long as possible. Additional points are awarded for staying aloft longer. Finally, bonus points are given for still having fuel once the craft lands. The score calculation can be seen in Equation 8.1.

Equation 8.1: Scoring the Neural Pilot

$$score = (fuel \cdot 10) + (velocity \cdot 1000) + fuel$$

In the next section we will run the Lunar Lander example and observe as it learns to land a spacecraft.

Running the Lunar Lander Example

To run the Lunar Lander game you should execute the **LunarLander** class. This class requires no arguments. Once the program begins, the neural network immediately begins training. It will cycle through 50 epochs, or training iterations, before it is done. As you can see, when it first begins, the score is a negative number. These early attempts by the untrained neural network are hitting the moon at high velocity and are not covering much distance.

```
Epoch #1 Score:-299.0
Epoch #2 Score:-299.0
Epoch #3 Score:-299.0
Epoch #4 Score:-299.0
Epoch #5 Score:-299.0
Epoch #6 Score:-299.0
Epoch #7 Score:-299.0
```

After the seventh epoch, the score begins to increase.

```
Epoch #8 Score:-96.0
Epoch #9 Score:5760.0
Epoch #10 Score:5760.0
Epoch #11 Score:5760.0
Epoch #12 Score:5760.0
Epoch #13 Score:5760.0
Epoch #14 Score:5760.0
Epoch #15 Score:5760.0
Epoch #16 Score:5760.0
Epoch #17 Score:6196.0
Epoch #18 Score:6196.0
Epoch #19 Score:6196.0
```

The score will hover at 6,196 for awhile, but will improve at a later epoch.

```
Epoch #45 Score:6275.0
Epoch #46 Score:6275.0
Epoch #47 Score:7347.0
Epoch #48 Score:7347.0
Epoch #49 Score:7460.0
Epoch #50 Score:7460.0
```

By the 50th epoch, a score of 7,460 has been achieved. The training techniques used in this chapter make extensive use of random numbers. As a result, when you run this example, you may get entirely different scores.

More epochs may have produced a better trained neural network; however, the program limits it to 50. This number usually produces a fairly skilled neural pilot. Once the network is trained, we run the simulation with the winning pilot. We display the telemetry at each second.

The neural pilot kept the craft aloft for 911 seconds. So, we will not show every telemetry report. However, we will highlight some of the interesting actions that this neural pilot learned. The neural network learned it was best to just let the craft freefall for awhile.

```
How the winning network landed:
Elapsed: 1 s, Fuel: 200 l, Velocity: -1.6200 m/s, 9998 m
Elapsed: 2 s, Fuel: 200 l, Velocity: -3.2400 m/s, 9995 m
Elapsed: 3 s, Fuel: 200 l, Velocity: -4.8600 m/s, 9990 m
Elapsed: 4 s, Fuel: 200 l, Velocity: -6.4800 m/s, 9983 m
Elapsed: 5 s, Fuel: 200 l, Velocity: -8.1000 m/s, 9975 m
Elapsed: 6 s, Fuel: 200 l, Velocity: -9.7200 m/s, 9965 m
Elapsed: 7 s, Fuel: 200 l, Velocity: -11.3400 m/s, 9954 m
Elapsed: 8 s, Fuel: 200 l, Velocity: -12.9600 m/s, 9941 m
Elapsed: 9 s, Fuel: 200 l, Velocity: -14.5800 m/s, 9927 m
Elapsed: 10 s, Fuel: 200 l, Velocity: -16.2000 m/s, 9910 m
Elapsed: 11 s, Fuel: 200 l, Velocity: -17.8200 m/s, 9893 m
Elapsed: 12 s, Fuel: 200 l, Velocity: -19.4400 m/s, 9873 m
Elapsed: 13 s, Fuel: 200 l, Velocity: -21.0600 m/s, 9852 m
Elapsed: 14 s, Fuel: 200 l, Velocity: -22.6800 m/s, 9829 m
Elapsed: 15 s, Fuel: 200 l, Velocity: -24.3000 m/s, 9805 m
Elapsed: 16 s, Fuel: 200 l, Velocity: -25.9200 m/s, 9779 m
Elapsed: 17 s, Fuel: 200 l, Velocity: -27.5400 m/s, 9752 m
Elapsed: 18 s, Fuel: 200 l, Velocity: -29.1600 m/s, 9722 m
Elapsed: 19 s, Fuel: 200 l, Velocity: -30.7800 m/s, 9692 m
Elapsed: 20 s, Fuel: 200 l, Velocity: -32.4000 m/s, 9659 m
Elapsed: 21 s, Fuel: 200 l, Velocity: -34.0200 m/s, 9625 m
Elapsed: 22 s, Fuel: 200 l, Velocity: -35.6400 m/s, 9590 m
Elapsed: 23 s, Fuel: 200 l, Velocity: -37.2600 m/s, 9552 m
Elapsed: 24 s, Fuel: 200 l, Velocity: -38.8800 m/s, 9514 m
Elapsed: 25 s, Fuel: 200 l, Velocity: -40.0000 m/s, 9473 m
Elapsed: 26 s, Fuel: 200 l, Velocity: -40.0000 m/s, 9431 m
Elapsed: 27 s, Fuel: 200 l, Velocity: -40.0000 m/s, 9390 m
```

You can see that 27 seconds in, and 9,390 meters above the ground, the terminal velocity of -40 m/s has been reached. There is no real science behind -40 m/s being the terminal velocity; it was just chosen as an arbitrary number. Having a terminal velocity is interesting because the neural networks learn that once this is reached, the craft will not speed up. They use the terminal velocity to save fuel, and only "break their fall" when they get close to the surface. The freefall at terminal velocity continues for some time.

Finally, at 6,102 meters above the ground, the thrusters are fired for the first time.

```
Elapsed: 105 s, Fuel: 200 l, Velocity: -40.0000 m/s, 6143 m
Elapsed: 106 s, Fuel: 200 l, Velocity: -40.0000 m/s, 6102 m
THRUST
Elapsed: 107 s, Fuel: 199 l, Velocity: -31.6200 m/s, 6060 m
```

```
Elapsed: 108 s, Fuel: 199 l, Velocity: -33.2400 m/s, 6027 m
Elapsed: 109 s, Fuel: 199 l, Velocity: -34.8600 m/s, 5992 m
Elapsed: 110 s, Fuel: 199 l, Velocity: -36.4800 m/s, 5956 m
Elapsed: 111 s, Fuel: 199 l, Velocity: -38.1000 m/s, 5917 m
Elapsed: 112 s, Fuel: 199 l, Velocity: -39.7200 m/s, 5878 m
THRUST
Elapsed: 113 s, Fuel: 198 l, Velocity: -31.3400 m/s, 5836 m
Elapsed: 114 s, Fuel: 198 l, Velocity: -32.9600 m/s, 5803 m
Elapsed: 115 s, Fuel: 198 l, Velocity: -34.5800 m/s, 5769 m
Elapsed: 116 s, Fuel: 198 l, Velocity: -36.2000 m/s, 5733 m
Elapsed: 117 s, Fuel: 198 l, Velocity: -37.8200 m/s, 5695 m
```

The velocity is gradually slowed, as the neural network decides to fire the thrusters every six seconds. This keeps the velocity around -35 m/s.

```
THRUST
Elapsed: 118 s, Fuel: 197 l, Velocity: -29.4400 m/s, 5655 m
Elapsed: 119 s, Fuel: 197 l, Velocity: -31.0600 m/s, 5624 m
Elapsed: 120 s, Fuel: 197 l, Velocity: -32.6800 m/s, 5592 m
Elapsed: 121 s, Fuel: 197 l, Velocity: -34.3000 m/s, 5557 m
Elapsed: 122 s, Fuel: 197 l, Velocity: -35.9200 m/s, 5521 m
THRUST
Elapsed: 123 s, Fuel: 196 l, Velocity: -27.5400 m/s, 5484 m
Elapsed: 124 s, Fuel: 196 l, Velocity: -29.1600 m/s, 5455 m
Elapsed: 125 s, Fuel: 196 l, Velocity: -30.7800 m/s, 5424 m
Elapsed: 126 s, Fuel: 196 l, Velocity: -32.4000 m/s, 5392 m
Elapsed: 127 s, Fuel: 196 l, Velocity: -34.0200 m/s, 5358 m
Elapsed: 128 s, Fuel: 196 l, Velocity: -35.6400 m/s, 5322 m
THRUST
```

As the craft gets closer to the lunar surface, this maximum allowed velocity begins to decrease. The pilot is slowing the craft, as it gets closer to the lunar surface. At around 4,274 meters above the surface the neural network decides it should now thrust ever five seconds. This slows the descent to around -28 m/s.

```
THRUST
Elapsed: 163 s, Fuel: 189 l, Velocity: -22.3400 m/s, 4274 m
Elapsed: 164 s, Fuel: 189 l, Velocity: -23.9600 m/s, 4250 m
Elapsed: 165 s, Fuel: 189 l, Velocity: -25.5800 m/s, 4224 m
Elapsed: 166 s, Fuel: 189 l, Velocity: -27.2000 m/s, 4197 m
Elapsed: 167 s, Fuel: 189 l, Velocity: -28.8200 m/s, 4168 m
THRUST
Elapsed: 168 s, Fuel: 188 l, Velocity: -20.4400 m/s, 4138 m
Elapsed: 169 s, Fuel: 188 l, Velocity: -22.0600 m/s, 4116 m
```

```
Elapsed: 170 s, Fuel: 188 l, Velocity: -23.6800 m/s, 4092 m
Elapsed: 171 s, Fuel: 188 l, Velocity: -25.3000 m/s, 4067 m
Elapsed: 172 s, Fuel: 188 l, Velocity: -26.9200 m/s, 4040 m
Elapsed: 173 s, Fuel: 188 l, Velocity: -28.5400 m/s, 4011 m
THRUST
```

By occasionally using shorter cycles, the neural pilot slows it even further by the time we are only 906 meters above the surface. The craft has been slowed to -14 meters per second.

```
THRUST
Elapsed: 320 s, Fuel: 162 l, Velocity: -6.6800 m/s, 964 m
Elapsed: 321 s, Fuel: 162 l, Velocity: -8.3000 m/s, 955 m
Elapsed: 322 s, Fuel: 162 l, Velocity: -9.9200 m/s, 945 m
Elapsed: 323 s, Fuel: 162 l, Velocity: -11.5400 m/s, 934 m
Elapsed: 324 s, Fuel: 162 l, Velocity: -13.1600 m/s, 921 m
Elapsed: 325 s, Fuel: 162 l, Velocity: -14.7800 m/s, 906 m
THRUST
Elapsed: 326 s, Fuel: 161 l, Velocity: -6.4000 m/s, 890 m
Elapsed: 327 s, Fuel: 161 l, Velocity: -8.0200 m/s, 882 m
Elapsed: 328 s, Fuel: 161 l, Velocity: -9.6400 m/s, 872 m
Elapsed: 329 s, Fuel: 161 l, Velocity: -11.2600 m/s, 861 m
Elapsed: 330 s, Fuel: 161 l, Velocity: -12.8800 m/s, 848 m
Elapsed: 331 s, Fuel: 161 l, Velocity: -14.5000 m/s, 833 m
THRUST
```

This short cycling continues until the craft has slowed its velocity considerably. It even thrusts to the point of increasing its altitude towards the final seconds of the flight.

```
Elapsed: 899 s, Fuel: 67 l, Velocity: 5.3400 m/s, 2 m
Elapsed: 900 s, Fuel: 67 l, Velocity: 3.7200 m/s, 5 m
Elapsed: 901 s, Fuel: 67 l, Velocity: 2.1000 m/s, 8 m
Elapsed: 902 s, Fuel: 67 l, Velocity: 0.4800 m/s, 8 m
Elapsed: 903 s, Fuel: 67 l, Velocity: -1.1400 m/s, 7 m
Elapsed: 904 s, Fuel: 67 l, Velocity: -2.7600 m/s, 4 m
THRUST
Elapsed: 905 s, Fuel: 66 l, Velocity: 5.6200 m/s, 0 m
Elapsed: 906 s, Fuel: 66 l, Velocity: 4.0000 m/s, 4 m
Elapsed: 907 s, Fuel: 66 l, Velocity: 2.3800 m/s, 6 m
Elapsed: 908 s, Fuel: 66 l, Velocity: 0.7600 m/s, 7 m
Elapsed: 909 s, Fuel: 66 l, Velocity: -0.8600 m/s, 6 m
Elapsed: 910 s, Fuel: 66 l, Velocity: -2.4800 m/s, 4 m
THRUST
Elapsed: 911 s, Fuel: 65 l, Velocity: 5.9000 m/s, 0 m
```

Finally, the craft lands, with a very soft velocity of positive 5.9.

You may be wondering why the lander lands with a velocity of 5.9. This is due to a slight glitch in the program. This "glitch" is left in, because it illustrates an important point. When neural networks are allowed to learn, they are totally on their own, and they will take advantage of everything that they can find.

The final positive velocity is because the program decides if it wants to thrust as the last part of a simulation cycle. The program has already decided the craft's altitude is below zero, and it has landed. But the neural network "sneaks in" that one final thrust. Even though the craft is already landed, and this thrust does no actual good. However, the final thrust does increase the score of the neural network.

Recall equation 8.1. For every negative meter per second of velocity, at landing, the program score is decreased by 1,000. The program figured out that the opposite is also true. For every positive meter per second of velocity it also gains 1,000 points. By learning about this little quirk in the program, the neural pilot can obtain even higher scores.

The neural pilot learned some very interesting things. We did not devise a strategy and force it upon the network. The network learned what it wanted to do. Specifically, this pilot decided the following:

- Freefall for some time to take advantage of terminal velocity.
- At a certain point, break the freefall and slow the craft.
- Slowly lose speed as you approach the surface.
- Give one final thrust, after landing, to maximize score.

The neural pilot in this example was trained using a genetic algorithm. Genetic algorithms and simulated annealing will be discussed later in this chapter. First, we will see how the Lander was simulated, and how its score is actually calculated.

Examining the Lunar Lander Simulator

We will now examine how the Lunar Lander example was created. We will look both at how the physical simulation is accomplished, as well as how the neural network actually pilots the spacecraft. Finally, we will see how the neural network learns to be a better pilot.

Simulating the Lander

First, we need a class that will simulate the "physics" of lunar landing. I use the term physics very loosely. The purpose of this example is more on how a neural network adapts to an artificial environment, than any sort of realistic physical simulation.

All of the physical simulation code is contained in the **LanderSimulator** class. This class is shown in Listing 8.1.

Listing 8.1: Simulating the Lander

```
package org.encog.examples.neural.lunar;

import java.text.NumberFormat;

public class LanderSimulator {

  public static final double GRAVITY = 1.62;
  public static final double THRUST = 10;
  public static final double TERMINAL_VELOCITY = 40;

  private int fuel;
  private int seconds;
  private double altitude;
  private double velocity;

  public LanderSimulator() {
    this.fuel = 200;
    this.seconds = 0;
    this.altitude = 10000;
    this.velocity = 0;
  }

  public void turn(boolean thrust) {
    this.seconds++;
    this.velocity -= GRAVITY;
    this.altitude += this.velocity;

    if (thrust && this.fuel > 0) {
      this.fuel--;
      this.velocity += THRUST;
    }

    this.velocity = Math.max(-TERMINAL_VELOCITY, this.velocity);
```

```
    this.velocity = Math.min(TERMINAL_VELOCITY, this.velocity);

    if (this.altitude < 0)
      this.altitude = 0;
    }

    public String telemetry() {
    NumberFormat nf = NumberFormat.getNumberInstance();
    nf.setMinimumFractionDigits(4);
    nf.setMaximumFractionDigits(4);
    StringBuilder result = new StringBuilder();
    result.append("Elapsed: ");
    result.append(seconds);
    result.append(" s, Fuel: ");
    result.append(this.fuel);
    result.append(" l, Velocity: ");
    result.append(nf.format(velocity));
    result.append(" m/s, ");
    result.append((int) altitude);
    result.append(" m");
    return result.toString();
  }

  public int score() {
    return (int) ((this.fuel * 10)
      + this.seconds + (this.velocity * 1000));
  }

  public int getFuel() {
    return fuel;
  }

  public int getSeconds() {
    return seconds;
  }

  public double getAltitude() {
    return altitude;
  }

  public double getVelocity() {
    return velocity;
  }

  public boolean flying() {
    return (this.altitude > 0);
```

```
  }
}
```

This class begins by defining some constants that will be important to the simulation.

```
public static final double GRAVITY = 1.62;
public static final double THRUST = 10;
public static final double TERMINAL_VELOCITY = 40;
```

The **GRAVITY** constant defines the acceleration on the moon that is due to gravity. It is set to 1.62, and is measured in meters per second. The **THRUST** constant specifies how the number of meters per second by which the gravity acceleration will be countered. The **TERMINAL_VELOCITY** is the fastest speed that the spacecraft can travel either upward or downward.

In addition to these constants, the simulator program will need several instance variables to maintain state. These variables are listed below as follows:

```
private int fuel;
private int seconds;
private double altitude;
private double velocity;
```

The **fuel** variable holds the amount of fuel remaining. The **seconds** variable holds the number of seconds aloft. The **altitude** variable holds the current altitude, in meters. The velocity variable holds the current velocity. Positive numbers indicate that the craft is moving upwards. Negative numbers indicate that the craft is moving downwards.

The simulator sets the values to reasonable starting values in the following constructor:

```
public LanderSimulator() {
  this.fuel = 200;
  this.seconds = 0;
  this.altitude = 10000;
  this.velocity = 0;
}
```

The craft is given 200 liters to start with. The altitude is set to 10,000 meters above ground.

The **turn** method processes each "turn". A turn is one second in the simulator. The **thrust** parameter indicates whether the spacecraft wishes to thrust during this turn.

```
public void turn(boolean thrust) {
```

First, increase the number of seconds elapsed by one. Decrease the velocity by the **GRAVITY** constant to simulate the fall.

```
  this.seconds++;
  this.velocity-=GRAVITY;
```

The current velocity increases the altitude. Of course, if the velocity is negative, the altitude will decrease.

```
  this.altitude+=this.velocity;
```

If thrust is applied during this turn, then decrease the **fuel** by one and increase the velocity by the **THRUST** constant.

```
  if( thrust && this.fuel>0 ) {
    this.fuel--;
    this.velocity+=THRUST;
  }
```

We must impose the terminal velocity. We do not want to fall or ascend faster than the terminal velocity. The following line makes sure that we are not ascending faster than the terminal velocity.

```
  this.velocity = Math.max(
    -TERMINAL_VELOCITY,
    this.velocity);
```

The following line makes sure that we are not descending faster than the terminal velocity.

```
  this.velocity = Math.min(
    TERMINAL_VELOCITY,
    this.velocity);
```

The following line makes sure that the altitude does not drop below zero. We do not want to simulate the craft hitting so hard that it goes underground.

```
if( this.altitude<0)
    this.altitude = 0;
}
```

In addition to the simulation code, the **LanderSimulator** also provides two utility functions. The first calculates the score. It should only be called after the spacecraft has landed. This method is shown here.

```
public int score() {
  return (int)((this.fuel*10)
    + this.seconds
    + (this.velocity*1000));
}
```

The **score** method implements Equation 8.1. As you can see it uses **fuel**, **seconds** and **velocity** to calculate the score, according to the earlier equation.

Additionally, a method is provided to determine if the spacecraft is still flying. If the altitude is greater than zero, it is still flying.

```
public boolean flying() {
  return(this.altitude>0);
}
```

In the next section we will see how the neural network actually flies the spacecraft and is given a score.

Calculating the Score

The **PilotScore** class implements the code necessary for the neural network to fly the spacecraft. This class also calculates the final score after the craft has landed. This class is shown in Listing 8.2.

Listing 8.2: Calculating the Lander Score

```
package org.encog.examples.neural.lunar;

import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.training.CalculateScore;
```

```
public class PilotScore implements CalculateScore {

  public double calculateScore(BasicNetwork network) {
    NeuralPilot pilot = new NeuralPilot(network, false);
    return pilot.scorePilot();
  }

  public boolean shouldMinimize() {
    return false;
  }
}
```

As you can see from the following line, the **PilotScore** class implements the **CalculateScore** interface.

```
public class PilotScore implements CalculateScore {
```

The **CalculateScore** interface is used by both Encog simulated annealing and genetic algorithms. It is used to determine how effective a neural network is at solving the given problem. A low score could be either bad or good, depending on the problem.

The **CalculateScore** interface requires two methods. This first method is named **calculateNetworkScore**. This method accepts a neural network and returns a **double** that represents the score of the network.

```
public double calculateNetworkScore(

  BasicNetwork network) {

  NeuralPilot pilot = new NeuralPilot(network, false);
  return pilot.scorePilot();
}
```

The second method returns a value to indicate if the score should be minimized.

```
public boolean shouldMinimize() {
  return false;
}
```

For this example we would like to maximize the score. As a result the **shouldMinimize** method returns **false**.

Flying the Spacecraft

In this section we will see how the neural network actually flies the spacecraft. The neural network will be fed environmental information, such as fuel remaining, altitude and current velocity. The neural network will then output a single value that will indicate if the neural network wishes to thrust. The **NeuralPilot** class performs this flight. You can see the **NeuralPilot** class in Listing 8.3.

Listing 8.3: Flying the Spacecraft

```
package org.encog.examples.neural.lunar;

import org.encog.neural.data.NeuralData;
import org.encog.neural.networks.BasicNetwork;
import org.encog.normalize.DataNormalization;
import org.encog.normalize.input.BasicInputField;
import org.encog.normalize.input.InputField;
import org.encog.normalize.output.OutputFieldRangeMapped;

public class NeuralPilot {

  private BasicNetwork network;
  private DataNormalization norm;
  private boolean track;

  public NeuralPilot(BasicNetwork network, boolean track) {
    InputField fuelIN;
    InputField altitudeIN;
    InputField velocityIN;

    this.track = track;
    this.network = network;

    norm = new DataNormalization();
    norm.addInputField(fuelIN = new BasicInputField());
    norm.addInputField(altitudeIN = new BasicInputField());
    norm.addInputField(velocityIN = new BasicInputField());
    norm.addOutputField(new OutputFieldRangeMapped(
      fuelIN,-0.9,0.9));
    norm.addOutputField(new OutputFieldRangeMapped(
      altitudeIN,-0.9,0.9));
    norm.addOutputField(new OutputFieldRangeMapped(velocityIN,
    -0.9,0.9));
    fuelIN.setMax(200);
```

```
    fuelIN.setMin(0);
    altitudeIN.setMax(10000);
    altitudeIN.setMin(0);
    velocityIN.setMin(-LanderSimulator.TERMINAL_VELOCITY);
    velocityIN.setMax(LanderSimulator.TERMINAL_VELOCITY);
  }

  public int scorePilot() {
    LanderSimulator sim = new LanderSimulator();

  while(sim.flying()) {
    double[] data = new double[3];
    data[0] = sim.getFuel();
    data[1] = sim.getAltitude();
    data[2] = sim.getVelocity();

    NeuralData input = this.norm.buildForNetworkInput(data);
    NeuralData output = this.network.compute(input);
    double value = output.getData(0);

    boolean thrust;

    if( value > 0 ) {
      thrust = true;

    if( track )
      System.out.println("THRUST");
    }
    else
      thrust = false;
    sim.turn(thrust);

    if( track )
      System.out.println(sim.telemetry());
    }
    return(sim.score());
  }
}
```

The **NeuralPilot** constructor sets up the pilot to fly the spacecraft. The constructor is passed a network to fly the spacecraft, as well as a Boolean that indicates if telemetry should be tracked to the screen.

```
public NeuralPilot(
  BasicNetwork network, boolean track) {
```

The constructor begins by setting up a **DataNormalization** object. The following field types are defined as local variables:

```
InputField fuelIN;
InputField altitudeIN;
InputField velocityIN;
OutputFieldRangeMapped fuelOUT;
OutputFieldRangeMapped altitudeOUT;
OutputFieldRangeMapped velocityOUT;
```

We save the operating parameters. The track variable is saved to the instance level so that the program will later know if it should display telemetry.

```
this.track = track;
this.network = network;
```

In the last chapter we used normalization to transform a raw CSV file into a normalized CSV file. In this chapter we will use normalization in "real-time". To do this, we begin by creating a normalization object.

```
norm = new DataNormalization();
```

The neural pilot will have three input neurons and one output neuron. These three input neurons will communicate the following three fields to the neural network.

- Current fuel level
- Current altitude
- Current velocity

These three input fields will produce one output field that indicates if the neural pilot would like to fire the thrusters.

To normalize these three fields, you define them as three **BasicInputField** objects. These fields are then added to the normalization class.

```
norm.addInputField(fuelIN = new BasicInputField());
norm.addInputField(altitudeIN = new BasicInputField());
norm.addInputField(velocityIN = new BasicInputField());
```

We use the **BasicInputField** because these fields are very simple; we will provide the data. The data are not coming from a CSV file, an array, or some other more complex structure. We will simply place the raw values directly into the input fields.

We also add three output fields. Recall from the previous chapter, that input and output fields are a matter of perspective. These three "output fields" are simply the output from the normalization. They will be the "input" to the neural network.

All three fields are normalized using the **OutputFieldRangeMapped** object. This will map the raw data into the ranges specified here. In this case it will be between -0.9 and +0.9.

```
norm.addOutputField(fuelOUT =
  new OutputFieldRangeMapped(fuelIN,-0.9,0.9));
norm.addOutputField(altitudeOUT =
  new OutputFieldRangeMapped(altitudeIN,-0.9,0.9));
norm.addOutputField(velocityOUT =
  new OutputFieldRangeMapped(velocityIN,-0.9,0.9));
```

We must also set the minimum and maximum raw data values for each of the three fields. This allows the normalization object to know how their true range so that they can be mapped. In this last chapter, this was done automatically. However, because we don't have all of the training data up front, and we are training in "real time", we must make some estimate of the minimum and maximum raw data values.

```
fuelIN.setMax(200);
fuelIN.setMin(0);
altitudeIN.setMax(10000);
altitudeIN.setMin(0);
velocityIN.setMin(-LanderSimulator.TERMINAL_VELOCITY);
velocityIN.setMax(LanderSimulator.TERMINAL_VELOCITY);
}
```

For this example, the primary purpose of flying the space craft is to receive a score. The **scorePilot** method calculates this score. It will simulate a flight from the point that the space craft is dropped from the orbiter to the point that it lands. The **scorePilot** method calculates this score.

```
public int scorePilot() {
```

This method begins by creating a **LanderSimulator** object. This object will simulate the very simple physics used by this program.

```
LanderSimulator sim = new LanderSimulator();
```

We now enter the main loop of the **scorePilot** method. It will continue looping, so long as the space craft is still flying. The space craft is still flying, so long as its altitude is greater than zero.

```
while(sim.flying()) {
```

We begin by creating an array to hold the raw data. The raw data is obtained directly from the simulator.

```
double[] data = new double[3];
data[0] = sim.getFuel();
data[1] = sim.getAltitude();
data[2] = sim.getVelocity();
```

The input to the neural network is constructed from the normalization object.

```
NeuralData input = this.norm.buildForNetworkInput(data);
```

This data is fed to the neural network, and the output is gathered.

```
NeuralData output = this.network.compute(input);
```

This single output neuron will determine if the thrusters should be fired.

```
double value = output.getData(0);

boolean thrust;
```

If the value is greater than zero, then the thrusters will be fired. If we are tracking, then also display that the thrusters were fired.

```
if( value > 0 ) {
  thrust = true;
  if( track )
    System.out.println("THRUST");
}
else
  thrust = false;
```

Process the next "turn" in the simulator, and thrust if necessary. Also display telemetry if we are tracking.

```
  sim.turn(thrust);
  if( track )
    System.out.println(sim.telemetry());
}
```

The space craft has now landed. Return the score based on the criteria previously discussed.

```
  return(sim.cost());
```

We will now look at how to train the neural pilot.

Training the Neural Pilot

This example can train the neural pilot using either a genetic algorithm or simulated annealing. Encog treats both genetic algorithms and simulated annealing very similarly. On one hand, you can simply provide a training set and use simulated annealing or you can use a genetic algorithm just like you did for a propagation network. We will see an example of this later in the chapter as we apply these two techniques to the XOR problem, and you see how similar they can be to propagation training.

On the other hand, genetic algorithms and simulated annealing can do something that propagation training cannot. They can allow you to train without a training set. It is still supervised training, because you will use the scoring class, developed earlier in this chapter. However, you do not need to come up with training data. You just need to tell the neural network how good of a job it is doing. If you can provide this scoring function, simulated annealing or a genetic algorithm can train the neural network. Both methods will be discussed. We will begin with a genetic algorithm.

What is a Genetic Algorithm

Genetic algorithms attempt to simulate Darwinian evolution to create a better neural network. The neural network is reduced to an array of **double** variables. This array becomes the genetic sequence.

The genetic algorithm begins by creating a population of random neural networks. All neural networks in this population have the same structure, meaning they have the same number of neurons and layers. However, they all have different random weights.

These neural networks are sorted according their "scores". Their scores are provided by the scoring method, discussed in the last section. In the case of the neural pilot, this score indicates how softly the ship landed.

The top neural networks are selected to "breed". The bottom neural networks "die". When two networks breed, we simulate nature by splicing their DNA. In this case, splices are taken from the double array from each network and spliced together to create a new offspring neural network. The offspring neural networks take up the places vacated by the dying neural networks.

Some of the offspring will be "mutated". That is, some of the genetic material will be random, and not from either parent. This introduces needed variety into the gene pool and simulates the natural process of mutation.

The population is sorted, and the process begins again. Each iteration provides one cycle. As you can see, there is no need for a training set. All that is needed is an object to score each neural network. Of course you can use training sets. To do this you simply a provided scoring object that uses a training set to score each network.

Using a Genetic Algorithm

Using the genetic algorism is very easy. The **NeuralGeneticAlgorithm** class is used to do this. The **NeuralGeneticAlgorithm** class implements the **Train** interface. Therefore, once constructed, it is used in the same way as any other Encog training class.

The following code creates a new **NeuralGeneticAlgorithm** to train the neural pilot.

```
train = new NeuralGeneticAlgorithm(
  network, new FanInRandomizer(),
  new PilotScore(),500, 0.1, 0.25);
```

The base network is provided to communicate the structure of the neural network to the genetic algorithm. The genetic algorithm will disregard weights currently set by the neural network.

The randomizer is provided so that the neural network can create a new random population. The **FanInRandomizer** attempts to produce starting weights that are less extreme, and more trainable, than the regular **RangeRandomizer** that is usually used. However, either randomizer could be used.

The value of 500 specifies the population size. Larger populations will train better, but will take more memory and processing time. The 0.1 is used to mutate 10% of the offspring. The 0.25 value is used to choose the mating population from the top 25% of the population.

```
int epoch = 1;
```

Now that the trainer has been set up we can train the neural network just like any Encog training object. Here we only iterate 50 times. This is usually enough to produce a skilled neural pilot.

```
for(int i=0;i<50;i++) {
  train.iteration();
  System.out.println(
  "Epoch #" + epoch + " Score:" + train.getError());
  epoch++;
}
```

We could have also trained using the **EncogUtility** class, as was done in the previous chapter. Just for simple training the **EncogUtility** is usually the preferred method. However, if your program needs to do something after each iteration, the more manual approach, shown above, may be preferable.

What is Simulated Annealing

Simulated annealing can also be used to train the neural pilot. Simulated annealing is similar to a genetic algorithm in that you should provide a scoring object. However, internally, it works quite differently. Simulated annealing simulates the metallurgical process of annealing.

Annealing is the process by which a very hot molten metal is slowly cooled. This slow cooling process causes the metal to produce a strong consistent

molecular structure. Annealing is a process that allows metals to be produced that are less likely to fracture or shatter.

A similar process can be performed on neural networks. To implement simulated annealing, the neural network is converted to an array of double values. This is exactly the same process as was done for the genetic algorithm.

Randomness is used to simulate the heat and cooling effect. While the neural network is still really "hot" the existing weights of the neural network increase in speed. As the network cools, this randomness slows down. Only changes that produce a positive effect on the network's score are kept.

Using Simulated Annealing

To use simulated annealing to train the neural pilot, pass the argument anneal on the command line when running this example. It is very simple for the example to use annealing rather than a genetic algorithm. They both use the same scoring function and are interchangeable. The following lines of code make use of the simulated annealing algorithm for this example.

```
if( args.length>0 && args[0].equalsIgnoreCase("anneal"))
{
  train = new NeuralSimulatedAnnealing(
      network, new PilotScore(), 10, 2, 100);
}
```

The simulated annealing object **NeuralSimulatedAnnealing** is used to train the neural pilot. The neural network is passed, along with the same scoring object that was used to train using a genetic algorithm.

The values of ten and two are the starting and stopping temperatures, respectively. They are not true temperatures, in terms of Fahrenheit or Celsius. A higher number will produce more randomness; a lower number produces less randomness. The following code shows how this temperature, or factor is applied.

```
public double randomize(final double d) {
  return d + (this.factor - (Math.random() * this.factor * 2));
}
```

The number 100 specifies how many cycles, per iteration, that it should take to go from the higher temperature to the lower temperature. Generally, the more cycles you have, the more accurate the results will be. However, the higher the number, the longer it takes to train.

There are no simple rules for how to set these values. Generally, you will need to experiment with different values to see which trains your particular neural network the best.

Using the Training Set Score Class

You can also use training sets with genetic algorithms and simulated annealing. Used this way, simulated annealing and genetic algorithms are a little different than propagation training based on usage. There is no scoring function when used this way. You simply use the **TrainingSetScore** object, which takes the training set and uses it to score the neural network.

Generally resilient propagation will outperform genetic algorithms or simulated annealing, when used in this way. Genetic algorithms or simulated annealing really excel when using a scoring method instead of a training set. Furthermore, you can use simulated annealing sometimes to push backpropagation out of a local minimum. We will see an example of this in the chapter on recurrent neural networks.

Listing 8.4 shows an example of training a neural network for the XOR operator using a training set-based genetic algorithm.

Listing 8.4: XOR and the Genetic Algorithm

```
package org.encog.examples.neural.xorgenetic;

import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.CalculateScore;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.TrainingSetScore;
import org.encog.neural.networks.training.genetic.
  NeuralGeneticAlgorithm;
import org.encog.util.logging.Logging;
```

```
import org.encog.util.randomize.FanInRandomizer;

public class XorGenetic {
  public static double XOR_INPUT[][] = {
  { 0.0, 0.0 },
  { 1.0, 0.0 },
  { 0.0, 1.0 },
  { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
  { 0.0 },
  { 1.0 },
  { 1.0 },
  { 0.0 } };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();

    BasicNetwork network = new BasicNetwork();
    network.addLayer(new BasicLayer(2));
    network.addLayer(new BasicLayer(3));
    network.addLayer(new BasicLayer(1));
    network.getStructure().finalizeStructure();
    network.reset();

    NeuralDataSet trainingSet = new BasicNeuralDataSet(
      XOR_INPUT, XOR_IDEAL);

    CalculateScore score = new TrainingSetScore(trainingSet);
    // train the neural network
    final Train train = new NeuralGeneticAlgorithm(
      network, new FanInRandomizer(), score, 5000, 0.1, 0.25);

    int epoch = 1;

    do {
      train.iteration();
      System.out.println("Epoch #" + epoch
        + " Error:" + train.getError());
      epoch++;
    } while ((epoch < 5000) && (train.getError() > 0.001));

    network = train.getNetwork();

    // test the neural network
```

```
    System.out.println("Neural Network Results:");
    for(NeuralDataPair pair: trainingSet ) {
    final NeuralData output = network.compute(pair.getInput());
    System.out.println(pair.getInput().getData(0) + ","
      + pair.getInput().getData(1)
      + ", actual=" + output.getData(0) + ",ideal="
      + pair.getIdeal().getData(0));
    }
  }
}
```

The following lines create a training set-based genetic algorithm. First, create a **TrainingSetScore** object.

```
CalculateScore score = new TrainingSetScore(trainingSet);
```

This object can then be used with either a genetic algorithm or simulated annealing. The following code shows it being used with a genetic algorithm.

```
final Train train = new NeuralGeneticAlgorithm(
  network, new FanInRandomizer(), score, 5000, 0.1, 0.25);
```

To use the **TrainingSetScore** object with simulated annealing simply pass it to the simulated annealing constructor, as was done above.

Summary

In this chapter you saw how to use genetic algorithms and simulated annealing to train a neural network. Both of these techniques can use a scoring object, rather than training sets. Both algorithms can also use a training set, if desired.

Genetic algorithms attempt to simulate Darwinian evolution. Neural networks are sorted based on fitness. Better neural networks are allowed to breed; inferior networks die. The next generation takes genetic material from the fittest neural networks.

Simulated annealing simulates the metallurgical process of annealing. The weights of the network are taken from a high temperature to a low. As the temperature is lowered, the best networks are chosen. This produces a neural network that is suited to getting better scores.

So far we have only seen how to use supervised training. In supervised training a neural network is given feedback on the success of its solutions. This can be in the form of a training set or a scoring function. Unsupervised training gives the neural network no such guidance. The next chapter will discuss unsupervised training.

Questions

1. Are simulated annealing and genetic algorithms considered supervised training or unsupervised training? Why?
2. Which is more desirable from a scoring function, a high score, or a low score?
3. How do you use a training set together with simulated annealing, or a genetic algorithm?
4. How is "randomness" used by simulated annealing?
5. What is the role of mutation in a genetic algorithm?
6. What advantages do simulated annealing and genetic algorithms have over propagation training?
7. When used with a training set, which will perform better, a genetic algorithm or resilient propagation?
8. How is "randomness" used by a genetic algorithm?
9. Do Encog genetic algorithms alter the structure of a neural network?
10. What are the negative effects of having a population size that is too large?

Terms

Annealing Cycles

Crossover

Ending Temperature

Genetic Algorithms

Lunar Lander Game

Mutation

Score

Simulated Annealing

Staring Temperature

Terminal Velocity

Chapter 9: Unsupervised Training Methods

- What is a Self Organizing Map?
- Mapping colors with a SOM
- Training a SOM
- Applying the SOM to the forest cover data

This chapter focuses on using Encog to implement a Self Organizing Map (SOM). A SOM is a special type of neural network that is used to classify data. Typically, a SOM will take higher resolution data and map them to a single or multidimensional output. This can be very useful for creating a neural network to see the similarities among its input data. Dr. Teuvo Kohonen, of the Academy of Finland, created the SOM. Because of this, the SOM is sometimes called a Kohonen neural network.

A SOM is trained using a competitive, unsupervised training algorithm. Encog implements this training algorithm using the **CompetitiveTraining** class. This is a completely different type of training then those previously used in this book. The SOM does not use a training set or scoring object. There are no clearly defined objectives provided to the neural network at all. The only type of "objective" that the SOM has is to group similar inputs together.

The example that we will examine in this chapter will take colors as input and map similar colors together. This GUI example program will show, visually, how similar colors are grouped together by the self-organizing map.

The output from a self-organizing map is topological. This output is usually viewed in an n-dimensional way. Usually, the output is single dimensional, but it can also be two-dimensional, three-dimensional, even four-dimensional or higher. What this means is that the "position" of the output neurons is important. If two output neurons are closer to each other, they will be trained together more so than two neurons that are not as close.

All of the neural networks that we have examined so far in this book have not been topological. In previous examples from this book, the distance between neurons was unimportant. Output neuron number two was just as significant to output neuron number one as was output neuron number 100.

The Structure and Training of a SOM

An Encog SOM is implemented as a two-layer, neural network. The SOM simply has an input layer and an output layer. The input layer maps data to the output layer. As patterns are presented to the input layer, the output neuron with the highest activation is considered the winner.

There are no threshold values in the SOM network, only weights from the input layer to the output layer. Additionally, only a linear activation function is used. Figure 9.1 shows a SOM created in the Encog Workbench.

Figure 9.1: A Self-Organizing Map

The SOM represented by the illustration above will be used later in this chapter as an example. It has three input neurons, which will represent color components of red, green and blue. It has 2,500 output neurons, which represents a 50x50 output grid.

Structuring a SOM

We will now look at how the above SOM will be structured. This SOM will be given several colors to train on. These colors will be expressed as RGB vectors. The individual red, green and blue values can range between -1 and +1. Where -1 is no color, or black, and +1 is full intensity of red, green or blue. These three-color components make up the input to the neural network.

The output is a grid of 2,500 neurons arranged into 50 rows by 50 columns. This SOM will organize similar colors near each other in this output grid. Figure 9.2 shows this output.

Figure 9.2: The Output Grid

The above figure may not be as clear in black and white editions of this book as it is in color. However, you can see similar colors grouped near each other. A single, color-based SOM is a very simple example, but it allows you to visually see the grouping capabilities of the SOM.

Training a SOM

We will now look at how the SOM is actually trained. The training process will update the weight matrix. The weight matrix is a 3 x 2,500 matrix. We initialize the weight matrix to random values to start. Then 15 training colors are chosen. These are simply random colors.

Just like previous examples, training will progress through a series of iterations. However, unlike feedforward neural networks, SOM networks are usually trained with a fixed number of iterations. For the colors example in this chapter we will use 1,000 iterations.

We will begin with the color sample that we wish to train for. We will choose one random color sample per iteration. We will pick one output

neuron whose weights most closely match the color on which we are training. The training pattern is a vector of three numbers. The weights between each of the 2,500 output neurons and the three input neurons are also a vector of three numbers. We calculate the Euclidean distance between the weight and training pattern. Both are a vector of three numbers. This is done with Equation 9.1.

Equation 9.1: The Euclidean Distance between Weight and Output Neuron

$$distance = \sqrt{(p_1 - w_1)^2 + (p_2 - w_2)^2 + (p_3 - w_3)^2}$$

In the above equation the variable **p** represents the input pattern. The variable **w** represents the weight vector. By squaring the differences between each of the vector components and then taking the square root of the resulting sum, we are given the Euclidean distance. This measures how different each weight vector is from the input training pattern.

This distance is calculated for every output neuron. The output neuron that has the shortest distance is called the Best Matching Unit (BMU). The BMU is the neuron that will learn the most from the training pattern. The neighbors of the BMU will learn less. Now that we have a BMU, we loop over all of the weights in the matrix. We will update every weight according to Equation 9.2.

Equation 9.2: SOM Learning Function

$$Wv(t+1) = Wv(t) + \theta(v,t)\alpha(t)(D(t) - Wv(t))$$

In the above equation the variable t represents time, or the iteration number. The purpose of the equation is to calculate the resulting weight vector **Wv(t+1)**. The next weight will be calculated by adding to the current weight, which is **Wv(t)**. We are essentially going to calculate how different the current weight is from the input vector. The clause **D(T)-Wv(t)** gives us this amount. If we simply added this value to the weight, the weight would exactly match the input vector. We don't want to do this. As a result, we scale it by multiplying it by two ratios. The first ratio, represented by theta, is the neighborhood function. The second ratio is a monotonically decreasing learning rate.

The neighborhood function considers how close a neighbor the output neuron we are training, is from the Best Matching Unit (BMU). For closer neurons, the neighborhood function will be close to one. For distant neighbors the neighborhood function will return zero. This controls how near and far neighbors are trained. We will look at how the neighborhood function determines this in the next section.

The learning rate also scales how much the output neuron will learn. This learning rate is similar to the learning rate used in backpropagation training. However, the learning rate should decrease as the training progresses. This learning rate must decrease monotonically. To decrease monotonically simply means that the function output only decreases or remains the same as time progresses. The output from the function will never increase at any interval as time increases.

Understanding Neighborhood Functions

The neighborhood function determines to what degree each output neuron should receive training from the current training pattern. The neighborhood function will return a value of one for the Best Matching Unit (BMU). This indicates that it should receive the most training of any neurons. Neurons further from the BMU will receive less training. It is the job of the neighborhood function to determine this percent.

If the output is arranged in only one dimension, then a simple one-dimensional neighborhood function should be used. A single dimension, self-organizing map treats the output as one long array of numbers. For instance, a single dimension network might have 100 output neurons, and they are simply treated as a long, single dimension array of 100 values.

A two dimensional SOM might take these same 100 values and treat them as a grid, perhaps a grid of 10 rows and 10 columns. The actual structure remains the same; the neural network has 100 output neurons. The only difference is the neighborhood function. The first would use a single dimensional neighborhood function; the second would use a two dimensional neighborhood function. The function would need to be able to consider this additional dimension and factor it into the distance returned.

It is also possible to have three, four, and even more dimensional functions for the neighborhood function. Two dimensions is the most popular choice.

Single dimensional neighborhood functions are also somewhat common. Three or more dimensions are more unusual. It really comes down to computing how many ways an output neuron can be close to another. Encog supports any number of dimensions, though each additional dimension adds greatly to the amount of memory and processing power needed.

The Gaussian function is a popular choice for a neighborhood function. The Gaussian function has single and multi-dimensional forms. The Single Dimension Gaussian function is shown in Equation 9.3.

Equation 9.3: The One-Dimensional Gaussian Function

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

The graph of the Gaussian function is shown in Figure 9.3.

Figure 9.3: A One-Dimensional Gaussian Function

From the above chart you can see why the Gaussian function is a popular choice for a neighborhood function. If the current output neuron is the BMU, then its distance (x-axis) will be zero. As a result, the training percent (y-

axis) is 100%. As the distance increases either positively or negatively, the training percentage decreases. Once the distance is great enough, the training percent is near-zero.

There are several constants in Equation 9.3 that govern the shape of the Gaussian function. The constant **a** determines the peak, or height of the Gaussian function. The constant **b** determines the center of the Gaussian function. The constant **c** determines the width of the curve. The variable **x** represents the distance that the current neuron is from the BMU.

The above Gaussian function is only useful for a one-dimensional output array. If you would like to use a two-dimensional output grid, you should use the two-dimensional form of the Gaussian function. Equation 9.4 shows the two-dimensional form of the Gaussian functions.

Equation 9.4: A Two-Dimensional Gaussian Function

$$f(x,y) = ae^{-\left(\frac{(x-b_1)^2}{2c_1^2} + \frac{(y-b_2)^2}{2c_2^2}\right)}$$

The graph form of the two-dimensional form of the Gaussian function is shown in Figure 9.3.

Figure 9.4: A Two Dimensional Gaussian Function

The two dimensional form of the Gaussian function takes a single peak variable, but you can specify separate values for the position and width of the curve. The equation does not need to be symmetrical.

You may be wondering how to set the Gaussian constants for use with a neural network. The peak is almost always one. If, for some reason, you wanted to unilaterally decrease the effectiveness of training, you could set the peak to something below one. However, this is more the role of the learning rate. The center is almost always zero, because you will want to center the curve on the origin. If you did change the center, then a neuron, other than the BMU, would receive the full learning. It is unlikely you would ever want to do this. For a multi-dimensional Gaussian, you would likely set all centers to zero, to truly center the curve at the origin.

This leaves the width of the Gaussian function. The width should be set to something slightly less than the entire width of the grid or array. Then the width should be gradually decreased. The width should be decreased monotonically, just like the learning rate.

Forcing a Winner

An optional feature to Encog SOM competitive training is the ability to force a winner. By default, Encog does not force a winner. However, this feature can be enabled for SOM training. Forcing a winner will try to ensure that each output neuron is winning for at least one of the training samples. This can cause a more even distribution of winners. However, it can also skew the data, as it does somewhat “engineer” the neural network. Because of this, it is disabled by default.

Calculating Error

In propagation training we could measure the success of our training by examining the current error of the neural network. In a SOM there is no direct error, because there is no expected output. Yet, the Encog interface **Train** exposes an error property. This property does return an estimation of the error of a SOM.

The error is defined to be the "worst", or longest Euclidean distance of any of the BMU's. This value should be minimized, as learning progresses. This gives a general approximation of how well the SOM has been trained.

Implementing the Colors SOM in Encog

We will now see how the color matching SOM is implemented. There are two classes that make up this example. They are listed here.

- **MapPanel**
- **SomColors**

The **MapPanel** class is used to display the weight matrix to the screen. The **SomColors** class extends the **JPanel** class and adds the **MapPanel** to itself for display. We will examine both classes, starting with the **MapPanel**.

Displaying the Weight Matrix

The **MapPanel** class draws the GUI display for the SOM as it progresses. This relatively simple class is shown in Listing 9.1.

Listing 9.1: Drawing the SOM

```
package org.encog.examples.neural.gui.som;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.synapse.Synapse;

public class MapPanel extends JPanel {

  /**
   *
   */
  private static final long serialVersionUID
    = 7528474872067939033L;
  public static final int CELL_SIZE = 8;
  public static final int WIDTH = 50;
  public static final int HEIGHT = 50;

  private Synapse synapse;

  public MapPanel(SomColors som) {
    this.synapse = som.getNetwork().getLayer(
      BasicNetwork.TAG_INPUT).getNext().get(0);
  }

  private int convertColor(double d) {
    double result = 128*d;
    result+=128;
    result = Math.min(result, 255);
    result = Math.max(result, 0);
    return (int)result;
  }

  @Override
  public void paint(Graphics g) {
```

```
    for(int y = 0; y< HEIGHT; y++) {

      for(int x = 0; x< WIDTH; x++) {
        int index = (y*WIDTH)+x;
        int red = convertColor(this.synapse.getMatrix()
         .get(0, index));
        int green = convertColor(this.synapse.getMatrix()
         .get(1, index));
        int blue = convertColor(this.synapse.getMatrix()
         .get(2, index));
        g.setColor(new Color(red,green,blue));
        g.fillRect(x*CELL_SIZE, y*CELL_SIZE,
           CELL_SIZE, CELL_SIZE);
      }
    }
  }
}
```

The **convertColor** function is very important. It converts a double that contains a range of -1 to +1 into the 0 to 255 range that an RGB component requires. A neural network deals much better with -1 to +1 than 0 to 255. As a result, this normalization is needed.

```
private int convertColor(double d) {
  double result = 128*d;
  result+=128;
  result = Math.min(result, 255);
  result = Math.max(result, 0);
  return (int)result;
}
```

The number 128 is the midpoint between 0 and 255. We multiply the **result** by 128 to get it to the proper range and then add 128 to diverge from the midpoint. We ensure that the result is in the proper range.

Using the **convertColor** method the **paint** method can properly draw the state of the SOM. The output from this function will be a color map of all of the weights in the neural network. Each output neuron, all 2,500 of them, are shown on a grid. Their color is determined by the weight between that output neuron and the three input neurons. These three weights are treated as RGB color components. The **convertColor** method is shown here.

```
public void paint(Graphics g)
{
```

We begin by looping through all 50 rows and columns.

```
  for(int y = 0; y< HEIGHT; y++)  {
    for(int x = 0; x< WIDTH; x++)  {
```

We wish to think of the output neurons as being in a two-dimensional grid. However, they are all stored as a one-dimensional array. We must calculate the current one-dimensional index from the two-dimensional **x** and **y** values.

```
      int index = (y*WIDTH)+x;
```

We obtain the three weight values from the matrix and use the **convertColor** method to convert these to RGB components.

```
      int red = convertColor(this.synapse.getMatrix().
        get(0, index));
      int green = convertColor(this.synapse.getMatrix().
        get(1, index));
      int blue = convertColor(this.synapse.getMatrix().
        get(2, index));
```

These three components are used to create a new **Color** object.

```
      g.setColor(new Color(red,green,blue));
```

A filled rectangle is displayed to display the neuron.

```
      g.fillRect(x*CELL_SIZE, y*CELL_SIZE, CELL_SIZE, CELL_SIZE);
    }
  }
}
```

Once the loops complete, the entire weight matrix has been displayed to the screen.

Training the Color Matching SOM

The **SomColors** class acts as the main **JPanel** for the application. It also provides all of the training for the neural network. This class can be seen in Listing 9.2.

Listing 9.2: Training the SOM

```
package org.encog.examples.neural.gui.som;
```

```
import java.util.ArrayList;
import java.util.List;

import javax.swing.JFrame;

import org.encog.neural.data.NeuralData;
import org.encog.neural.data.basic.BasicNeuralData;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.training.competitive
  .CompetitiveTraining;
import org.encog.neural.networks.training.competitive
  .neighborhood.NeighborhoodGaussianMulti;
import org.encog.neural.pattern.SOMPattern;
import org.encog.util.randomize.RangeRandomizer;

public class SomColors extends JFrame implements Runnable {

  /**
   *
   */
  private static final long serialVersionUID =
    -6762179069967224817L;
  private MapPanel map;
  private BasicNetwork network;
  private Thread thread;
  private CompetitiveTraining train;
  private NeighborhoodGaussianMulti gaussian;

  public SomColors() {
    this.setSize(640, 480);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.network = createNetwork();
    this.getContentPane().add(map = new MapPanel(this));
    this.gaussian = new NeighborhoodGaussianMulti(MapPanel.WIDTH,
      MapPanel.HEIGHT);
    this.train = new CompetitiveTraining(this.network,
      0.01, null, gaussian);
    train.setForceWinner(false);
    this.thread = new Thread(this);
    thread.start();
  }

  public BasicNetwork getNetwork() {
    return this.network;
  }
```

```
  private BasicNetwork createNetwork() {
    BasicNetwork result = new BasicNetwork();
    SOMPattern pattern = new SOMPattern();
    pattern.setInputNeurons(3);
    pattern.setOutputNeurons(MapPanel.WIDTH * MapPanel.HEIGHT);
    result = pattern.generate();
    result.reset();
    return result;
  }

  public static void main(String[] args) {
    SomColors frame = new SomColors();
    frame.setVisible(true);
  }

  public void run() {

    List<NeuralData> samples = new ArrayList<NeuralData>();
    for (int i = 0; i < 15; i++) {
    NeuralData data = new BasicNeuralData(3);
    data.setData(0, RangeRandomizer.randomize(-1, 1));
    data.setData(1, RangeRandomizer.randomize(-1, 1));
    data.setData(2, RangeRandomizer.randomize(-1, 1));
    samples.add(data);
  }

  this.train.setAutoDecay(1000, 0.8, 0.003, 30, 5);

  for (int i = 0; i < 1000; i++) {
    int idx = (int) (Math.random() * samples.size());
    NeuralData c = samples.get(idx);

    this.train.trainPattern(c);
    this.train.autoDecay();
    this.map.repaint();
    System.out.println("Iteration " + i + ","
      + this.train.toString());
    }
  }
}
```

The **CompetitiveTraining** class must be setup so that the neural network will train. However, we first need a neighborhood function. For this

example, we are going to use the **NeighborhoodGaussianMulti** neighborhood function. This neighborhood function is capable of supporting a multi-dimensional Gaussian neighborhood function. The following line of code creates this neighborhood function.

```
this.gaussian = new NeighborhoodGaussianMulti(
  MapPanel.WIDTH,MapPanel.HEIGHT,1,5,0);
```

The constructor being used here creates a two-dimensional Gaussian neighborhood function. The first two parameters specify the height and width of the grid.

There are other constructors that can create higher dimensional Gaussian functions. Additionally, there are other neighborhood functions provided by Encog. The two most common are the **NeighborhoodGaussian** and **NeighborhoodGaussianMulti**, which respectively implements a one-dimensional and a multi-dimensional Gaussian neighborhood function. The complete list of neighborhood functions is listed here.

- **NeighborhoodBubble**
- **NeighborhoodGaussian**
- **NeighborhoodGaussianMulti**
- **NeighborhoodSingle**

The **NeighborhoodBubble** only provides one-dimensional neighborhood functions. A radius is specified, and anything falling within that radius will get the full effect of training. The **NeighborhoodSingle** functions as a single-dimensional neighborhood function and will allow only the BMU to receive the effects of training.

We must also create a **CompetitiveTraining** object to make use of the neighborhood function.

```
this.train = new CompetitiveTraining(
  this.network,0.01,null,gaussian);
```

The first parameter specifies the network to train. The second parameter is the learning rate. We will automatically decrease the learning rate, so the learning rate specified here is not important. The third parameter is the training set. We will randomly feed colors to the neural network, so the

training set it's not needed. Finally, the fourth parameter is the neighborhood function that was just created.

The SOM training is provided for this example by a background thread. This allows the training to progress while the user watches. The background thread is implemented in the **run** method. The **run** method is shown here.

```
public void run() {
```

The **run** method begins by creating the 15 random colors for which the neural network will be trained. These random samples will be stored in the **samples** variable, which is a **List**.

```
  List<NeuralData> samples = new ArrayList<NeuralData>();
```

The random colors are generated. They have random numbers for the RGB components.

```
  for(int i=0;i<15;i++) {
    NeuralData data = new BasicNeuralData(3);
    data.setData(0, RangeRandomizer.randomize(-1,1));
    data.setData(1, RangeRandomizer.randomize(-1,1));
    data.setData(2, RangeRandomizer.randomize(-1,1));
    samples.add(data);
  }
```

The following line sets the parameters for the automatic decay of the learning rate and the radius.

```
  this.train.setAutoDecay(1000, 0.8, 0.003, 30, 5);
```

We must provide the anticipated number of iterations. For this example, this is 1,000. For SOM neural networks you should know the number of iterations up front. This is different than propagation training where we trained for either a specific amount of time or until below a specific error rate.

The parameters 0.8 and 0.003 are the beginning and ending learning rates. The error rate will be uniformly decreased from 0.8 to 0.003 over each iteration. It should reach close to 0.003 by the last iteration.

Likewise, the parameters 30 and 5 represent the beginning and ending radius. The radius will start at 30 and should be near 5 by the final

iteration. If more than the planned 1,000 iterations are performed, the radius and learning rate will not fall below their minimums.

```
  for(int i=0;i<1000;i++) {
```

For each competitive learning iteration, you have two choices. First, you can choose to simply provide a **NeuralDataSet** that contains the training data, and call the **iteration** method of the **CompetitiveTraining**.

Next we choose a random color index, and obtain that color.

```
    int idx = (int)(Math.random()*samples.size());
    NeuralData c = samples.get(idx);
```

The **trainPattern** method will train the neural network for this random color pattern. The BMU will be located and updated as described earlier in this chapter.

```
    this.train.trainPattern(c);
```

Alternatively, the colors could have been loaded into a **NeuralDataSet** object and the **iteration** method could have been used. However, training the patterns one at a time, and using a random pattern looks better when displayed on the screen.

Next, we call the **autoDecay** method that will decrease the learning rate and radius, according to the parameters previously specified.

```
    this.train.autoDecay();
```

The screen is repainted.

```
    this.map.repaint();
```

Finally, we display information about the current iteration.

```
    System.out.println("Iteration " + i + ","
      + this.train.toString());
  }
}
```

This process continues for 1,000 iterations. By the final iteration, the colors will be grouped.

Summary

Up to this point in the book all of the neural networks have been trained using a supervised training algorithm. This chapter introduced unsupervised training. Unsupervised training provides no feedback to the neural network like the error rates we previously saw.

A very common neural network type that can be used with unsupervised training is the Self Organizing Map (SOM), or Kohonen neural network. This neural network type has only an input and output layer. This is a competitive neural network; the neuron that has the highest output is considered the winning neuron.

A SOM trains by taking an input pattern and seeing which output neuron has the closest weight values to this input pattern. The closest matching neuron, called the Best Matching Unit (BMU) is then trained. All neighboring neurons are also trained. The degree to which neighbors are trained, and which neurons are even neighbors, is determined by a neighborhood function. The most commonly used neighborhood functions are variants of the Gaussian function.

Neural networks are very adept at recognizing patterns. Most of the examples illustrated in the book so far have focused on pattern recognition. Neural networks can also recognize patterns in time. This allows the neural network make predictions. The next chapter will focus on predictive neural networks.

Questions for Review

1. What is another common name for the Self Organizing Map (SOM)?
2. What is the purpose of a neighborhood function to the training of a Self Organizing Map?
3. Calculate the Euclidean distance between [1, 2, 3] and [-3, -2, -1].
4. How is the “error rate” calculated for a Self Organizing Map (SOM)?
5. What will most neighborhood functions return for the Best Matching Unit (BMU)?

6. If the learning rate were 0.5 and the neighborhood function returned 0.25, what percent of the training would be applied to this neuron?
7. Do SOM neural networks have threshold values?
8. Which activation function is used with a SOM?
9. The Gaussian function allows you to specify a width, center and peak. What center and peak values are typically used for a SOM neighborhood function?
10. Which neighborhood function has no radius and only allows the BMU to learn?

Terms

Best Matching Unit (BMU)

Competitive Training

Gaussian Neighborhood Function

Kohonen Neural Network

Neighborhood Function

Self Organizing Map (SOM)

Chapter 10: Using Temporal Data

- How a Predictive Neural Network Works
- Using the Encog Temporal Dataset
- Attempting to Predict Sunspots
- Using the Encog Market Dataset
- Attempting to Predict the Stock Market

Prediction is another common use for neural networks. A predictive neural network will attempt to predict future values based on present and past values. Such neural networks are called temporal neural networks, because they operate over time. This chapter will introduce temporal neural networks and the support classes that Encog provides for them.

In this chapter, you will see two applications of Encog temporal neural networks. First, we will look at how to use Encog to predict sunspots. Sunspots are reasonably predictable, and the neural network should be able to learn future patterns by analyzing past data. Next, we will examine a simple case of applying a neural network to making stock market predictions.

Before we look at either example we must see how a temporal neural network actually works. Simple, recurrent neural networks will be discussed later in this book. A temporal neural network is usually either a feedforward or simple recurrent network. Structured properly, the feedforward neural networks we have seen so far could be structured as a temporal neural network. It is the meaning that we assign to the input and output neurons that make a network a temporal neural network.

How a Predictive Neural Network Works

A predictive neural network uses its inputs to accept information about current data and uses its outputs to predict future data. It uses two "windows", a future window and a past window. Both windows must have a window size, which is the amount of data that is either predicted or is needed to predict. To see the two windows in action, consider the following data.

```
Day 1: 100
Day 2: 102
Day 3: 104
Day 4: 110
Day 5: 99
```

```
Day 6: 100
Day 7: 105
Day 8: 106
Day 9: 110
Day 10: 120
```

Consider a temporal neural network with a past window size of five and a future window size of two. This neural network would have five input neurons and two output neurons. We would break the above data among these windows to produce training data. The following data shows one such element of training data.

```
Input 1: Day 1: 100 (input neuron)
Input 2: Day 2: 102 (input neuron)
Input 3: Day 3: 104 (input neuron)
Input 4: Day 4: 110 (input neuron)
Input 5: Day 5: 99  (input neuron)
Ideal 1: Day 6: 100 (output neuron)
Ideal 2: Day 7: 105 (output neuron)
```

Of course the data above would need to be normalized in some way before it can be fed to the neural network. The above illustration simply shows how the input and output neurons are mapped to the actual data. To get additional data both windows are simply slid forward. The next element of training data would be as follows.

```
Input 1: Day 2: 102 (input neuron)
Input 2: Day 3: 104 (input neuron)
Input 3: Day 4: 110 (input neuron)
Input 4: Day 5: 99  (input neuron)
Input 5: Day 6: 100 (input neuron)
Ideal 1: Day 7: 105 (output neuron)
Ideal 2: Day 8: 106 (output neuron)
```

You would continue sliding the past and future windows forward as you generate more training data. Encog contains specialized classes to prepare data in this format. You simply specify the size of the past, or input, window and the future, or output, window. These specialized classes will be discussed in the next section.

Using the Encog Temporal Dataset

The Encog temporal dataset is contained in the following package:

```
org.encog.neural.data.temporal
```

There are a total of four classes that make up the Encog temporal dataset. These classes are as follows:

- **TemporalDataDescription**
- **TemporalError**
- **TemporalNeuralDataSet**
- **TemporalPoint**

The **TemporalDataDescription** class describes one unit of data that is either used for prediction or output. The **TemporalError** class is an exception that is thrown if there is an error while processing the temporal data. The **TemporalNeuralDataSet** class operates just like any Encog dataset and allows the temporal data to be used for training. The **TemporalPoint** class represents one point of temporal data.

To begin using a **TemporalNeuralDataSet** we must instantiate it. This is done as follows:

```
TemporalNeuralDataSet result =
  new TemporalNeuralDataSet(
    [past window size] , [future window size] );
```

The above instantiation specifies both the size of the past and future windows. You must also define one or more **TemporalDataDescription** objects. These define the individual items inside of the past and future windows. One single **TemporalDataDescription** object can function as both a past and a future window element as illustrated in the code below.

```
TemporalDataDescription desc =
  new TemporalDataDescription(
    [calculation type] , [use for past] , [use for future] );

result.addDescription(desc);
```

To specify that a **TemporalDataDescription** object functions as both a past and future element, use the value **true** for the last two parameters. There are several calculation types that you can specify for each data description. These types are summarized here.

- TemporalDataDescription.RAW,
- TemporalDataDescription.PERCENT_CHANGE,
- TemporalDataDescription.DELTA_CHANGE,

The **RAW** type specifies that the data points should be passed on, unmodified, to the neural network. The **PERCENT_CHANGE** specifies that each point should be passed on as a percentage change. The **DELTA_CHANGE** specifies that each point should be passed on as the actual change between the two values. If you are normalizing the data yourself, you would use the RAW type. Otherwise, it is very likely you would use the **PERCENT_CHANGE** type.

Next you must provide the raw data to train the temporal network from. To do this, create **TemporalPoint** objects and add them to the temporal dataset. Each **TemporalPoint** object can contain multiple values. You should have the same number of values in each of your temporal data points as you had **TemporalDataDescription** objects. The following code shows how to define a temporal data point.

```
TemporalPoint point = new TemporalPoint( [number of values] );
point.setSequence( [a sequence number] );
point.setData(0, [ value 1 ] );
point.setData(1, [ value 2 ] );
result.getPoints().add(point);
```

Every data point should have a sequence number. This allows the data points to be sorted. The **setData** method calls allow the individual values to be set. The number of **setData** method calls should match the specified number of values in the constructor.

Finally, you should call the **generate** method. This method takes all of the temporal points and creates the training set. After **generate** has been called, the **TemportalDataSet** object can be use for training.

```
result.generate();
```

In the next section we will see how to make use of a **TemportalDataSet** object to predict sunspots.

Application to Sunspots

In this section we will see how to use Encog to predict sunspots. Sunspots are fairly periodic and predictable. A neural network can learn this pattern and predict the number of sunspots with reasonable accuracy. The output from the sunspot prediction program is shown here. You can see that the neural network first begins training. The network trains until the error rate falls below six percent.

```
Epoch #1 Error:0.39165411390480664
Epoch #2 Error:1.2907898518116008
Epoch #3 Error:1.275679853982214
Epoch #4 Error:0.8026954615095163
Epoch #5 Error:0.4999305514145095
Epoch #6 Error:0.468223450164209
Epoch #7 Error:0.22034289938540677
Epoch #8 Error:0.2406776630699879
...
Epoch #128 Error:0.06025613803011326
Epoch #129 Error:0.06002069579351901
Epoch #130 Error:0.059830227113598734
Year  Actual      Predict     Closed Loop Predict
1960  0.5723      0.5547      0.5547
1961  0.3267      0.4075      0.3918
1962  0.2577      0.1837      0.2672
1963  0.2173      0.1190      0.0739
1964  0.1429      0.1738      0.1135
1965  0.1635      0.2631      0.3650
1966  0.2977      0.2327      0.4203
1967  0.4946      0.2870      0.1342
1968  0.5455      0.6167      0.3533
1969  0.5438      0.5111      0.6415
1970  0.5395      0.3830      0.4011
1971  0.3801      0.4072      0.2469
```

```
1972  0.3898      0.2148      0.2342
1973  0.2598      0.2533      0.1788
1974  0.2451      0.1686      0.2163
1975  0.1652      0.1968      0.2064
1976  0.1530      0.1470      0.2032
1977  0.2148      0.1533      0.1751
1978  0.4891      0.3579      0.1014
```

Once the network has been trained, it tries to predict the number of sunspots between 1960 and 1978. It does this with at least some degree of accuracy. The number displayed is normalized, and simply provides an idea of the relative number of sunspots. A larger number indicates more sunspot activity; a lower number indicates less sunspot activity.

There are two prediction numbers given: the regular prediction, and the closed-loop prediction. Both prediction types use a past window of 30 and a future window of 1. The regular prediction simply uses the last 30 values from real data. The closed loop starts this way. But its own predictions become the input as the window slides forward. This usually results in a less accurate prediction because any mistakes the neural network makes are compounded.

We will now examine how this program was implemented. Listing 10.1 shows the code for the sunspot prediction program.

Listing 10.1: Predicting Sunspots

```
package org.encog.examples.neural.predict.sunspot;

import java.text.NumberFormat;

import org.encog.NullStatusReportable;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralData;
import org.encog.neural.data.temporal.TemporalDataDescription;
import org.encog.neural.data.temporal.TemporalNeuralDataSet;
import org.encog.neural.data.temporal.TemporalPoint;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.resilient
  .ResilientPropagation;
import org.encog.normalize.DataNormalization;
import org.encog.normalize.input.InputField;
```

```
import org.encog.normalize.input.InputFieldArray1D;
import org.encog.normalize.output.OutputFieldRangeMapped;
import org.encog.normalize.target.NormalizationStorageArray1D;
import org.encog.util.logging.Logging;

public class PredictSunspot {

  public final static double[] SUNSPOTS = {
    0.0262,  0.0575,  0.0837,  0.1203,  0.1883,  0.3033,
    0.1517,  0.1046,  0.0523,  0.0418,  0.0157,  0.0000,
    0.0000,  0.0105,  0.0575,  0.1412,  0.2458,  0.3295,
    0.3138,  0.2040,  0.1464,  0.1360,  0.1151,  0.0575,
    0.1098,  0.2092,  0.4079,  0.6381,  0.5387,  0.3818,
    0.2458,  0.1831,  0.0575,  0.0262,  0.0837,  0.1778,
    0.3661,  0.4236,  0.5805,  0.5282,  0.3818,  0.2092,
    0.1046,  0.0837,  0.0262,  0.0575,  0.1151,  0.2092,
    0.3138,  0.4231,  0.4362,  0.2495,  0.2500,  0.1606,
    0.0638,  0.0502,  0.0534,  0.1700,  0.2489,  0.2824,
    0.3290,  0.4493,  0.3201,  0.2359,  0.1904,  0.1093,
    0.0596,  0.1977,  0.3651,  0.5549,  0.5272,  0.4268,
    0.3478,  0.1820,  0.1600,  0.0366,  0.1036,  0.4838,
    0.8075,  0.6585,  0.4435,  0.3562,  0.2014,  0.1192,
    0.0534,  0.1260,  0.4336,  0.6904,  0.6846,  0.6177,
    0.4702,  0.3483,  0.3138,  0.2453,  0.2144,  0.1114,
    0.0837,  0.0335,  0.0214,  0.0356,  0.0758,  0.1778,
    0.2354,  0.2254,  0.2484,  0.2207,  0.1470,  0.0528,
    0.0424,  0.0131,  0.0000,  0.0073,  0.0262,  0.0638,
    0.0727,  0.1851,  0.2395,  0.2150,  0.1574,  0.1250,
    0.0816,  0.0345,  0.0209,  0.0094,  0.0445,  0.0868,
    0.1898,  0.2594,  0.3358,  0.3504,  0.3708,  0.2500,
    0.1438,  0.0445,  0.0690,  0.2976,  0.6354,  0.7233,
    0.5397,  0.4482,  0.3379,  0.1919,  0.1266,  0.0560,
    0.0785,  0.2097,  0.3216,  0.5152,  0.6522,  0.5036,
    0.3483,  0.3373,  0.2829,  0.2040,  0.1077,  0.0350,
    0.0225,  0.1187,  0.2866,  0.4906,  0.5010,  0.4038,
    0.3091,  0.2301,  0.2458,  0.1595,  0.0853,  0.0382,
    0.1966,  0.3870,  0.7270,  0.5816,  0.5314,  0.3462,
    0.2338,  0.0889,  0.0591,  0.0649,  0.0178,  0.0314,
    0.1689,  0.2840,  0.3122,  0.3332,  0.3321,  0.2730,
    0.1328,  0.0685,  0.0356,  0.0330,  0.0371,  0.1862,
    0.3818,  0.4451,  0.4079,  0.3347,  0.2186,  0.1370,
    0.1396,  0.0633,  0.0497,  0.0141,  0.0262,  0.1276,
    0.2197,  0.3321,  0.2814,  0.3243,  0.2537,  0.2296,
    0.0973,  0.0298,  0.0188,  0.0073,  0.0502,  0.2479,
    0.2986,  0.5434,  0.4215,  0.3326,  0.1966,  0.1365,
    0.0743,  0.0303,  0.0873,  0.2317,  0.3342,  0.3609,
```

```
    0.4069,  0.3394,  0.1867,  0.1109,  0.0581,  0.0298,
    0.0455,  0.1888,  0.4168,  0.5983,  0.5732,  0.4644,
    0.3546,  0.2484,  0.1600,  0.0853,  0.0502,  0.1736,
    0.4843,  0.7929,  0.7128,  0.7045,  0.4388,  0.3630,
    0.1647,  0.0727,  0.0230,  0.1987,  0.7411,  0.9947,
    0.9665,  0.8316,  0.5873,  0.2819,  0.1961,  0.1459,
    0.0534,  0.0790,  0.2458,  0.4906,  0.5539,  0.5518,
    0.5465,  0.3483,  0.3603,  0.1987,  0.1804,  0.0811,
    0.0659,  0.1428,  0.4838,  0.8127 };

    public final static int STARTING_YEAR = 1700;
    public final static int WINDOW_SIZE = 30;
    public final static int TRAIN_START = WINDOW_SIZE;
    public final static int TRAIN_END = 259;
    public final static int EVALUATE_START = 260;
    public final static int EVALUATE_END = SUNSPOTS.length-1;

    /**
     * This really should be lowered, I am setting it to a
     * level here that will train in under a minute.
     */
    public final static double MAX_ERROR = 0.01;

    private double[] normalizedSunspots;
    private double[] closedLoopSunspots;

    public void normalizeSunspots(double lo,double hi) {
      InputField in;
      // create arrays to hold the normalized sunspots
      normalizedSunspots = new double[SUNSPOTS.length];
      closedLoopSunspots = new double[SUNSPOTS.length];

      // normalize the sunspots
      DataNormalization norm = new DataNormalization();
      norm.setReport(new NullStatusReportable());
      norm.addInputField(in =
        new InputFieldArray1D(true,SUNSPOTS));
      norm.addOutputField(
        new OutputFieldRangeMapped(in, lo, hi));
      norm.setTarget(
        new NormalizationStorageArray1D(normalizedSunspots));
      norm.process();
      System.arraycopy(normalizedSunspots, 0,
        closedLoopSunspots, 0, normalizedSunspots.length);
    }
```

```
    public NeuralDataSet generateTraining() {
      TemporalNeuralDataSet result =
  new TemporalNeuralDataSet(WINDOW_SIZE,1);

  TemporalDataDescription desc =
    new TemporalDataDescription(
      TemporalDataDescription.Type.RAW,true,true);
      result.addDescription(desc);

  for(int year = TRAIN_START;year<TRAIN_END;year++) {
    TemporalPoint point = new TemporalPoint(1);
    point.setSequence(year);
    point.setData(0, this.normalizedSunspots[year]);
    result.getPoints().add(point);
  }

  result.generate();
  return result;
}

public BasicNetwork createNetwork() {
  BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(WINDOW_SIZE));
  network.addLayer(new BasicLayer(10));
  network.addLayer(new BasicLayer(1));
  network.getStructure().finalizeStructure();
  network.reset();
  return network;
}

public void train(BasicNetwork network,NeuralDataSet training) {
  final Train train = new ResilientPropagation(
    network, training);

  int epoch = 1;

  do {
    train.iteration();
    System.out.println("Epoch #" + epoch
    + " Error:" + train.getError());
    epoch++;
  } while(train.getError() > MAX_ERROR);
}

public void predict(BasicNetwork network) {
  NumberFormat f = NumberFormat.getNumberInstance();
```

```
  f.setMaximumFractionDigits(4);
  f.setMinimumFractionDigits(4);

  System.out.println(
    "Year\tActual\tPredict\tClosed Loop Predict");

  for(int year=EVALUATE_START;year<EVALUATE_END;year++) {
  // calculate based on actual data
  NeuralData input = new BasicNeuralData(WINDOW_SIZE);

  for(int i=0;i<input.size();i++) {
    input.setData(i,
      this.normalizedSunspots[(year-WINDOW_SIZE)+i]);
  }

  NeuralData output = network.compute(input);
  double prediction = output.getData(0);
  this.closedLoopSunspots[year] = prediction;

  // calculate "closed loop", based on predicted data
  for(int i=0;i<input.size();i++) {
    input.setData(i,
      this.closedLoopSunspots[(year-WINDOW_SIZE)+i]);
  }

  output = network.compute(input);
  double closedLoopPrediction = output.getData(0);

  // display
  System.out.println((STARTING_YEAR+year)
    +"\t"+f.format(this.normalizedSunspots[year])
    +"\t"+f.format(prediction)
    +"\t"+f.format(closedLoopPrediction));

  }
}

public void run() {
  normalizeSunspots(0.1,0.9);
  BasicNetwork network = createNetwork();
  NeuralDataSet training = generateTraining();
  train(network,training);
  predict(network);
}

public static void main(String args[]) {
```

```
    Logging.stopConsoleLogging();
    PredictSunspot sunspot = new PredictSunspot();
    sunspot.run();
  }

}
```

As you can see, the program has Sunspot data hardcoded near the top of the file. This data was taken from a C-based, neural network example program. You can find the original application at the following URL:

http://www.neural-networks-at-your-fingertips.com/bpn.html

The older, C-based neural network example was modified to make use of Encog. You will notice that the Encog version is much shorter than the C-based version. This is because much of what the example did was already implemented in Encog. Further, the Encog version trains the network faster, because it makes use of resilient propagation, whereas the C-based example makes use of backpropagation.

This example goes through a two-step process for using the data. First, the raw data is normalized. Then, this normalized data is loaded into a **TemportalDataSet** object for temporal training. The **normalizeSunspots** method is called to normalize the sunspots. This method is shown below.

```
public void normalizeSunspots(double lo,double hi) {
```

The **hi** and **lo** parameters specify the high and low range to which the sunspots should be normalized. This is a range-mapped normalization, as discussed in the Chapter 6, "Obtaining Data for Encog". For this example the **lo** value is 0.1 and the **high** value is 0.9.

First, arrays are created to hold the normalized sunspots. The regular prediction and training will use the **normalizedSunspots** array. The closed loop prediction will use the **closedLoopsunspots** array.

```
  InputField in;

  // create arrays to hold the normalized sunspots
  normalizedSunspots = new double[SUNSPOTS.length];
  closedLoopSunspots = new double[SUNSPOTS.length];
```

The **DataNormalize** object will be used to normalize the sunspots. The **DataNormalize** object was discussed in Chapter 6.

```
  // normalize the sunspots
  DataNormalization norm = new DataNormalization();
```

First we set to use a **NullStatusReportable** report object. We do not care about the status updates from the normalization. Because the dataset is fairly small, normalization will happen very quickly.

```
  norm.setReport(new NullStatusReportable());
```

We will add a single input field and a single output field. The input will come from an array, and the output will be ranged mapped between the high and low values.

```
  norm.addInputField(in = new InputFieldArray1D(true,SUNSPOTS));
  norm.addOutputField(new OutputFieldRangeMapped(in, lo, hi));
```

The output will be to a one-dimensional array, named **normalizedSunspots**.

```
  norm.setTarget(new
NormalizationStorageArray1D(normalizedSunspots));
```

Calling the **process** method begins the normalization.

```
  norm.process();
```

Once the normalization is complete, the array should be copied to the closed-loop array.

```
  System.arraycopy(normalizedSunspots, 0,
    closedLoopSunspots, 0, normalizedSunspots.length);

}
```

Initially, the closed-loop array starts out the same as the regular prediction. However, its predictions will used to fill this array.

Now that the sunspot data has been normalized, it should be converted to temporal data. This is done by calling the **generateTraining** method, which is shown below.

```
public NeuralDataSet generateTraining() {
```

This method will return an Encog dataset that can be used for training. First a **TemporalNeuralDataSet** is created. The past and future window sizes are specified.

```
TemporalNeuralDataSet result = new
TemporalNeuralDataSet(WINDOW_SIZE,1);
```

We will have a single data description. Because we already normalized the data, this will be of type RAW. This data description will be used for both input, and prediction, as the last two parameters specify. Finally, we add this description to the dataset.

```
TemporalDataDescription desc = new TemporalDataDescription(
  TemporalDataDescription.Type.RAW,true,true);
result.addDescription(desc);
```

It is now necessary to create all of the data points. We will loop between the starting and ending year. These are the years that are used to train the neural network. Other years will be used to test the neural network's predictive ability.

```
for(int year = TRAIN_START;year<TRAIN_END;year++) {
```

Each data point will have only one value. We are using a single value to predict the sunspots. The sequence is the year, because we have only one sunspot sample per year.

```
  TemporalPoint point = new TemporalPoint(1);
  point.setSequence(year);
```

The one value we are using is the normalized number of sunspots. This number is both what we use to predict, from past values, as well as what we hope to predict in the future.

```
  point.setData(0, this.normalizedSunspots[year]);
  result.getPoints().add(point);
}
```

Finally, we generate the training set and return it.

```
result.generate();

return result;
```

```
}
```

The data is now ready for training. This dataset is trained using resilient propagation. This process is the same as we have done many times earlier in this book. Once training is complete we will attempt to predict sunspots using the application. This is done with the **predict** method, which is shown here.

```
public void predict(BasicNetwork network)
{
```

First, we create a **NumberFormat** object so that the numbers can be properly formatted. We will display four decimal places.

```
  NumberFormat f = NumberFormat.getNumberInstance();
  f.setMaximumFractionDigits(4);
  f.setMinimumFractionDigits(4);
```

We display the heading for the table and begin to loop through the evaluation years.

```
  System.out.println("Year\tActual\tPredict\tClosed Loop
Predict");

  for(int year=EVALUATE_START;year<EVALUATE_END;year++) {
```

We create input into the neural network based on actual data. This will be the actual prediction. We extract 30 years worth of data for the past window.

```
  NeuralData input = new BasicNeuralData(WINDOW SIZE);
  for(int i=0;i<input.size();i++) {
    input.setData(i,this.normalizedSunspots[(year-
WINDOW_SIZE)+i]);
  }
```

The neural network is presented with the data, and we retrieve the prediction.

```
  NeuralData output = network.compute(input);
  double prediction = output.getData(0);
```

The prediction is saved to the closed-loop array for use with future predictions.

```
  this.closedLoopSunspots[year] = prediction;
```

We will now calculate the closed-loop value. The calculation is essentially the same, except that the closed-loop data, which is continually modified, is used. Just as before, we grab 30 years worth of data.

```
  for(int i=0;i<input.size();i++) {
    input.setData(i,this.closedLoopSunspots[
      (year-WINDOW_SIZE)+i]);
  }
```

We compute the output.

```
  output = network.compute(input);
  double closedLoopPrediction = output.getData(0);
```

Finally, we display the closed-loop prediction, the regular prediction and the actual value.

```
  System.out.println((STARTING_YEAR+year)
    +"\t"+f.format(this.normalizedSunspots[year])
    +"\t"+f.format(prediction)
    +"\t"+f.format(closedLoopPrediction)
    );

  }
}
```

This will display a list of all of the sunspot predictions made by Encog. In the next section we will see how Encog can automatically pull current market information and attempt to predict stock market directions.

Using the Encog Market Dataset

Encog also includes a dataset specifically designed for stock market data. This dataset is capable of downloading data from external sources. Currently, the only external source included in Encog is Yahoo Finance. The Encog market dataset is built on top of the temporal dataset. Most classes in the Encog market dataset descend directly from corresponding classes in the temporal data set.

The following classes make up the Encog Market Dataset package:

- **MarketDataDescription**
- **MarketDataType**
- **MarketError**
- **MarketNeuralDataSet**
- **MarketPoint**
- **TickerSymbol**

The **MarketDataDescription** class represents one piece of market data that is part of either the past or future window. It descends from the **TemporalDataDescription** class. It consists primarily of a **TickerSymbol** object and a **MarketDataType** enumeration. The ticker symbol specifies the security we are to include, and the **MarketDataType** specifies the type of data, from this security, that we would like to use. The types of data available are listed below.

- **OPEN** - The market open for the day.
- **CLOSE** - The market close for the day.
- **VOLUME** - The volume for the day.
- **ADJUSTED_CLOSE** - The adjusted close. Adjusted for splits and dividends.
- **HIGH** - The high for the day.
- **LOW** - The low for the day.

These are the market data types criteria currently supported by Encog. They are all represented inside of the **MarketDataType** enumeration.

The **MarketNeuralDataSet** class is descended from the **TemporalNeuralDataSet**. This is the main class you will deal with when creating market-based training data for Encog. This class is an Encog dataset, and can be trained from. If any errors occur, the **MarketError** exception will be thrown.

The **MarketPoint** class descends from the **TemporalPoint**. You will usually not deal with this object directly, as you will usually have Encog download market data for you from Yahoo Finance. The following code shows

the general format for using the **MarketNeuralDataSet** class. First we must create a loader. Currently, the **YahooFinanceLoader** is the only public loader available for Encog.

```
MarketLoader loader = new YahooFinanceLoader();
```

Next, we create the market dataset. We pass the loader, as well as the size of the past and future windows.

```
MarketNeuralDataSet market = new MarketNeuralDataSet(
loader,
[past window size],
[future window size] );
```

Next we create a **MarketDataDescription** object. To do this we specify the ticker symbol and data type we need. The last two true values at the end specify that this item is used both for past and predictive purposes.

```
final MarketDataDescription desc = new MarketDataDescription(
[ticker], [data type needed] , true, true);
```

We add this data description to the dataset.

```
market.addDescription(desc);
```

We can add additional descriptions as needed. Next, we load the market data and generate the training data.

```
market.load( [begin date], [end date] );
market.generate();
```

As you can see from the code, the beginning and ending dates must be specified. This tells Encog the range from which to generate training data.

Application to the Stock Market

We will now look at an example of applying Encog to stock market prediction. This program attempts to predict the direction of a single stock based on past performance. This is a very simple stock market example, and is not meant to offer any sort of investment advice.

We will begin by seeing how to run this example. There are three distinct modes in which this example can be run, depending on the command line argument that was passed. These arguments are summarized below.

- **generate** – Download financial data and generate training file.
- **train** – Train the neural network.
- **evaluate** – Evaluate the neural network.

To begin the example you should run the main class, which is named **MarketPredict**. In the following sections we will see how this example generates data, trains, and then evaluates the resulting neural network.

Generating Training Data

The first step is to generate the training data. The example is going to download about eight years worth of financial information to train with. It takes some time to download and process this information. The data is downloaded and written to an Encog EG file. The class **MarketBuildTraining** provides this functionality. The **MarketBuildTraining** class is shown in Listing 10.2.

Listing 10.2: Generating Training Data

```
public class MarketBuildTraining {

  public static void generate() {

    final MarketLoader loader = new YahooFinanceLoader();
    final MarketNeuralDataSet market =
      new MarketNeuralDataSet(loader,
      Config.INPUT_WINDOW, Config.PREDICT_WINDOW);
    final MarketDataDescription desc =
      new MarketDataDescription(
      Config.TICKER, MarketDataType.ADJUSTED_CLOSE, true, true);
    market.addDescription(desc);

    market.load(Config.TRAIN_BEGIN.getTime(),
      Config.TRAIN_END.getTime());
    market.generate();
    market.setDescription("Market data for: " +
      Config.TICKER.getSymbol());

    // create a network
```

```
    final BasicNetwork network = new BasicNetwork();
    network.addLayer(new BasicLayer(market.getInputSize()));
    network.addLayer(new BasicLayer(Config.HIDDEN1_COUNT));

    if (Config.HIDDEN2_COUNT != 0) {
    network.addLayer(new BasicLayer(Config.HIDDEN2_COUNT));
  }

  network.addLayer(new BasicLayer(market.getIdealSize()));
  network.getStructure().finalizeStructure();
  network.reset();

  // save the network and the training
  final EncogPersistedCollection encog =
    new EncogPersistedCollection(
  Config.FILENAME);
  encog.create();
  encog.add(Config.MARKET_TRAIN, market);
  encog.add(Config.MARKET_NETWORK, network);
  }
}
```

All work performed by this class is in the **static** method named **generate**. This method is shown below.

```
public static void generate() {
```

This method begins by creating a **YahooFinanceLoader** that will load the requested financial data.

```
  final MarketLoader loader = new YahooFinanceLoader();
```

A new **MarketNeuralDataSet** object is created that will use the **loader** and a specified size for the past and future windows. By default, the program uses a future window size of one and a past window size of ten. These constants are all defined in the **Config** class. You control how the network is structured and trained by changing any of the values in the **Config** class.

```
  final MarketNeuralDataSet market = new
MarketNeuralDataSet(loader,
    Config.INPUT_WINDOW, Config.PREDICT_WINDOW);
```

The program uses a single market value from which to make predictions. It will use the adjusted closing price of the specified security. The security that the program is trying to predict is specified in the **Config** class.

```
  final MarketDataDescription desc = new MarketDataDescription(
    Config.TICKER, MarketDataType.ADJUSTED_CLOSE, true, true);
  market.addDescription(desc);
```

The market data is now loaded between the specified beginning and ending dates. Later dates will be used to evaluate the neural network's performance.

```
  market.load(Config.TRAIN_BEGIN.getTime(),
Config.TRAIN_END.getTime());
  market.generate();
```

We will set the description of the training data. This will be displayed if the training data is ever opened with the Encog Workbench.

```
  market.setDescription("Market data for: " +
Config.TICKER.getSymbol());
```

We also create a network to save to the EG file. This network is a simple feedforward neural network that may have one or two hidden layers. The sizes of the hidden layers are specified in the **Config** class.

```
  final BasicNetwork network = new BasicNetwork();
  network.addLayer(new BasicLayer(market.getInputSize()));
  network.addLayer(new BasicLayer(Config.HIDDEN1_COUNT));
  if (Config.HIDDEN2_COUNT != 0) {
    network.addLayer(new BasicLayer(Config.HIDDEN2_COUNT));
  }
  network.addLayer(new BasicLayer(market.getIdealSize()));
  network.getStructure().finalizeStructure();
  network.reset();
```

We now create the EG file and store both the network and training data to this file. It is important to note that **TemporalDataSet** or any of its derived classes will persist as a **BasicNeuralDataSet**. Only the generated data will be saved, not the other support objects, such as the **MarketDataDescription** objects.

```
  final EncogPersistedCollection encog = new
EncogPersistedCollection(
    Config.FILENAME);
  encog.create();
  encog.add(Config.MARKET_TRAIN, market);
  encog.add(Config.MARKET_NETWORK, network);
```

Later phases of the program, such as the training and evaluation phases, will make use of this file.

Training the Neural Network

Training the neural network is very simple. The network and training data have already been created and stored in an EG file. All that the training class needs to do is load both of these resources from the EG file and begin training. The **MarketTrain** class does this. This class is shown in Listing 10.3.

Listing 10.3: Training the Neural Network

```
package org.encog.examples.neural.predict.market;

import java.io.File;

import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.propagation.resilient
  .ResilientPropagation;
import org.encog.persist.EncogPersistedCollection;

public class MarketTrain {

  public static void train() {

  final File file = new File(Config.FILENAME);

  if (!file.exists()) {
    System.out.println("Can't read file: "
      + file.getAbsolutePath());
    return;
  }

  final EncogPersistedCollection encog =
    new EncogPersistedCollection(
```

```
    file);

  final NeuralDataSet trainingSet = (NeuralDataSet) encog
    .find(Config.MARKET_TRAIN);

  final BasicNetwork network =
    (BasicNetwork) encog.find(Config.MARKET_NETWORK);

  // train the neural network
  final Train train =
    new ResilientPropagation(network, trainingSet);

  int epoch = 1;
  final long startTime = System.currentTimeMillis();
  int left = 0;

  do {
    final int running =
    (int) ((System.currentTimeMillis() - startTime) / 60000);
    left = Config.TRAINING_MINUTES - running;
    train.iteration();
    System.out.println("Epoch #" + epoch + " Error:"
      + (train.getError() * 100.0) + "%," + " Time Left: " + left
      + " Minutes");
    epoch++;
    } while ((left >= 0) && (train.getError() > 0.001));

    network.setDescription("Trained neural network");
    encog.add(Config.MARKET_NETWORK, network);
  }
}
```

The **static** method **train** performs all of the training. This method is shown here.

```
public static void train() {
```

The method begins by verifying whether the Encog EG file is present. Training data and the network will be loaded from here.

```
  final File file = new File(Config.FILENAME);

  if (!file.exists()) {
    System.out.println("Can't read file: " +
file.getAbsolutePath());
      return;
```

Next, we create an **EncogPersistedCollection** object to load the EG file. We will extract a network and training set from this file.

```
  final EncogPersistedCollection encog = new
EncogPersistedCollection(file);
  final NeuralDataSet trainingSet = (NeuralDataSet) encog.find(
    Config.MARKET_TRAIN);
```

Next, we load the network from the EG file. This network will be used for training.

```
  final BasicNetwork network =
    (BasicNetwork) encog.find(Config.MARKET_NETWORK);
```

We are now ready to train the neural network. We will use **ResilientPropagation** training and loop for the number of minutes specified in the **Config** class.

```
  final Train train = new ResilientPropagation(network,
trainingSet);

  int epoch = 1;
  final long startTime = System.currentTimeMillis();
  int left = 0;

  do {
    final int running =
      (int) ((System.currentTimeMillis() - startTime) / 60000);
    left = Config.TRAINING_MINUTES - running;
    train.iteration();
    System.out.println("Epoch #" + epoch + " Error:"
      + (train.getError() * 100.0) + "%," + " Time Left: " + left
      + " Minutes");
    epoch++;
  } while ((left >= 0) && (train.getError() > 0.001));
```

Finally, the neural network is saved back to the EG file.

```
network.setDescription("Trained neural network");
encog.add(Config.MARKET_NETWORK, network);
```

At this point, the neural network is trained. You can run the training again to further train the neural network, or move on to evaluating the neural network. If you train the same neural network again, using resilient

propagation, the error rate will initially spike. This is because the resilient propagation algorithm must reestablish proper delta values for training.

Evaluating the Neural Network

We are now ready to evaluate the neural network. We will use the trained neural network from the last section and see how well it performs on actual current stock market data. The **MarketEvaluate** class contains all of the evaluation code. This class is shown in Listing 10.4.

Listing 10.4: Evaluating the Neural Network

```
package org.encog.examples.neural.predict.market;

import java.io.File;
import java.text.DecimalFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataPair;
import org.encog.neural.data.market.MarketDataDescription;
import org.encog.neural.data.market.MarketDataType;
import org.encog.neural.data.market.MarketNeuralDataSet;
import org.encog.neural.data.market.loader.MarketLoader;
import org.encog.neural.data.market.loader.YahooFinanceLoader;
import org.encog.neural.networks.BasicNetwork;
import org.encog.persist.EncogPersistedCollection;

public class MarketEvaluate {

  enum Direction {
    up, down
  };

  public static Direction determineDirection(double d) {
    if (d < 0)
      return Direction.down;
    else
      return Direction.up;
  }

  public static MarketNeuralDataSet grabData() {
    MarketLoader loader = new YahooFinanceLoader();
    MarketNeuralDataSet result = new MarketNeuralDataSet(loader,
```

```
      Config.INPUT_WINDOW,
      Config.PREDICT_WINDOW);
    MarketDataDescription desc =
      new MarketDataDescription(Config.TICKER,
        MarketDataType.ADJUSTED_CLOSE, true, true);
        result.addDescription(desc);

    Calendar end = new GregorianCalendar();// end today
    Calendar begin = (Calendar) end.clone();// begin 30 days ago
    begin.add(Calendar.DATE, -60);

    result.load(begin.getTime(), end.getTime());
    result.generate();

    return result;

  }

  public static void evaluate() {

    File file = new File(Config.FILENAME);

    if (!file.exists()) {
      System.out.println(
        "Can't read file: " + file.getAbsolutePath());
      return;
    }

    EncogPersistedCollection encog =
      new EncogPersistedCollection(file);
    BasicNetwork network =
      (BasicNetwork) encog.find(Config.MARKET_NETWORK);

    if (network == null) {
      System.out.println("Can't find network resource: "
        + Config.MARKET_NETWORK);
      return;
    }

    MarketNeuralDataSet data = grabData();

    DecimalFormat format = new DecimalFormat("#0.00");

    int count = 0;
    int correct = 0;
    for (NeuralDataPair pair : data) {
```

```
      NeuralData input = pair.getInput();
      NeuralData actualData = pair.getIdeal();
      NeuralData predictData = network.compute(input);

      double actual = actualData.getData(0);
      double predict = predictData.getData(0);
      double diff = Math.abs(predict - actual);

      Direction actualDirection = determineDirection(actual);
      Direction predictDirection = determineDirection(predict);

      if (actualDirection == predictDirection)
        correct++;

      count++;

      System.out.println("Day " + count + ":actual="
        + format.format(actual)
        + "(" + actualDirection + ")"
        + ",predict=" + format.format(predict) + "("
        + actualDirection + ")"
        + ",diff=" + diff);

      }

    double percent = (double) correct / (double) count;
    System.out.println("Direction correct:" + correct
      + "/" + count);
    System.out.println("Directional Accuracy:"
      + format.format(percent * 100) + "%");

  }
}
```

There are two important methods that are used during the evaluation process. The first is the **determineDirection** class. We are not attempting to determine the actual percent change for a security, but rather, which direction it will move the next day.

```
public static Direction determineDirection(final double d) {
  if( d<0 )
    return Direction.down;
  else
    return Direction.up;
}
```

This method simply returns an enumeration that specifies whether the stock price moved up or down.

We will need some current market data to evaluate against. The **grabData** method obtains the necessary market data. It makes use of a **MarketNeuralDataSet**, just as the training does, to obtain some market data. This method is shown here.

```
public static MarketNeuralDataSet grabData() {
```

Just like the training data generation, market data is loaded from a **YahooFinanceLoader** object.

```
  MarketLoader loader = new YahooFinanceLoader();
  MarketNeuralDataSet result = new MarketNeuralDataSet(
    loader,
    Config.INPUT_WINDOW,
    Config.PREDICT_WINDOW);
```

We create exactly the same data description as was used for training. We want the adjusted close for the specified ticker symbol. We also want the data both for past and future. We will feed the past data to the neural network and see how well the output matches the future data.

```
  MarketDataDescription desc = new MarketDataDescription(
    Config.TICKER,
    MarketDataType.ADJUSTED_CLOSE,
    true,
    true);
  result.addDescription(desc);
```

We need to choose what date range we would like to use to evaluate the network. We will grab the last 60 days worth of data.

```
  Calendar end = new GregorianCalendar();// end today
  Calendar begin = (Calendar)end.clone();// begin 60 days ago
  begin.add(Calendar.DATE, -60);
```

The market data is now loaded and generated by using the **load** method call.

```
  result.load(begin.getTime(), end.getTime());
  result.generate();
```

```
  return result;
}
```

The resulting data is returned to the calling method. Now that we have covered the support methods, we will see how the actual training occurs. The **static** method **evaluate** performs the actual evaluation. This method is shown below.

```
public static void evaluate() {
```

First, we make sure that the Encog EG file exists.

```
File file = new File(Config.FILENAME);

if( !file.exists() ) {
  System.out.println("Can't read file: " + file.getAbsolutePath()
);
  return;
}

  EncogPersistedCollection encog = new
EncogPersistedCollection(file);
```

Then, we load the neural network from the EG file. This is the neural network that was trained in the previous section.

```
  BasicNetwork network = (BasicNetwork)
encog.find(Config.MARKET_NETWORK);

  if( network==null ) {
    System.out.println("Can't find network resource: "
      + Config.MARKET_NETWORK );
    return;
  }
```

We load the market data with which we will use to evaluate the network. This is done using the **grabData** method discussed earlier in this section.

```
  MarketNeuralDataSet data = grabData();
```

We will use a formatter to format the percentages.

```
  DecimalFormat format = new DecimalFormat("#0.00");
```

As we perform the evaluation, we will keep count of how many cases we examined, as well as how many were correct.

```
int count = 0;
int correct = 0;
```

We begin looping over all of the market data we loaded.

```
for(NeuralDataPair pair: data) {
```

We retrieve one training pair and obtain the actual data as well as what was predicted. We get the predicted data by running the network using the **compute** method.

```
NeuralData input = pair.getInput();
NeuralData actualData = pair.getIdeal();
NeuralData predictData = network.compute(input);
```

We now retrieve the actual and predicted data and calculate the difference. This is how far off the neural network was on predicting the actual price change.

```
double actual = actualData.getData(0);
double predict = predictData.getData(0);
double diff = Math.abs(predict-actual);
```

We also calculate the direction the network predicted security takes, as well as the direction the security actually took.

```
Direction actualDirection = determineDirection(actual);
Direction predictDirection = determineDirection(predict);
```

If the direction was correct, then increment the correct count by one. Either way, increment the total count by one.

```
if( actualDirection==predictDirection )
  correct++;

count++;
```

Display the results for each case examined.

```
System.out.println("Day " + count+":actual="
  +format.format(actual)+"("+actualDirection+")"
  +",predict="
  +format.format(predict)+"("+actualDirection+")"
```

```
    +",diff="+diff);

  }
```

Finally, display stats on the overall accuracy of the neural network.

```
double percent = (double)correct/(double)count;
System.out.println("Direction correct:" + correct + "/" + count);
System.out.println("Directional
Accuracy:"+format.format(percent*100)+"%");
}
```

The following code snippet shows the output of this application when launched once. Because it is using data preceding the current date, the results will be different when you run it. These results occur because the program is attempting to predict percent movement on Apple Computer's stock price.

```
Day 1:actual=0.05(up),predict=-0.09(up),diff=0.1331431391626865
Day 2:actual=-
0.02(down),predict=0.15(down),diff=0.1752316137707985
Day 3:actual=-0.04(down),predict=-
0.08(down),diff=0.04318588896364293
Day 4:actual=0.04(up),predict=-0.13(up),diff=0.167230163960771
Day 5:actual=0.04(up),predict=0.08(up),diff=0.041364210497886064
Day 6:actual=-0.05(down),predict=-
0.15(down),diff=0.09856291235302134
Day 7:actual=0.03(up),predict=0.02(up),diff=0.0121349208067498
Day 8:actual=0.06(up),predict=0.14(up),diff=0.07873950162422072
Day 9:actual=0.00(up),predict=-0.04(up),diff=0.044884229765456175
Day 10:actual=-0.02(down),predict=-
0.11(down),diff=0.08800357702537594
Day 11:actual=-
0.03(down),predict=0.10(down),diff=0.1304932331559785
Day 12:actual=0.03(up),predict=-0.00(up),diff=0.03830226924277358
Day 13:actual=-0.04(down),predict=-
0.03(down),diff=0.006017023124087514
Day 14:actual=0.01(up),predict=-
0.00(up),diff=0.011094798099546017
Day 15:actual=-
0.07(down),predict=0.10(down),diff=0.1634993352860712
Day 16:actual=0.00(up),predict=0.09(up),diff=0.08529079398874763
Day 17:actual=0.01(up),predict=0.08(up),diff=0.07476901867409716
Day 18:actual=-
0.05(down),predict=0.10(down),diff=0.14462998342498684
```

```
Day
19:actual=0.01(up),predict=0.01(up),diff=0.0053944458622837204
Day 20:actual=-
0.02(down),predict=0.16(down),diff=0.17692298105888082
Day 21:actual=0.01(up),predict=0.01(up),diff=0.003908063600862748
Day 22:actual=0.01(up),predict=0.05(up),diff=0.04043842368088156
Day 23:actual=-
0.00(down),predict=0.05(down),diff=0.05856519756505361
Day 24:actual=-0.01(down),predict=-
0.01(down),diff=0.0031913517175624975
Day 25:actual=0.06(up),predict=0.03(up),diff=0.02967685979492382
Day 26:actual=0.04(up),predict=-0.01(up),diff=0.05155871532643232
Day 27:actual=-0.02(down),predict=-
0.09(down),diff=0.06931714317358993
Day 28:actual=-0.02(down),predict=-
0.04(down),diff=0.019323500655091908
Day 29:actual=0.02(up),predict=0.06(up),diff=0.04364949212592098
Day 30:actual=-0.02(down),predict=-
0.06(down),diff=0.036886336426948246
Direction correct:18/30
Directional Accuracy:60.00%
```

Here, the program had an accuracy of 60%, which is actually very good for this simple neural network. I've seen accuracy rates in the 30 to 40 percent range, when this program was run at different intervals.

This is a very simple stock market predictor. By no means should it be used for any sort of actual investing. It shows the foundation of how you structure a neural network to predict market direction.

Summary

In this chapter you saw how Encog could process temporal neural networks. Temporal networks are used to predict something that will occur in the future.. The first example in this chapter showed how to use Encog to predict sunspots. The second example showed how to use Encog to attempt to predict stock price movements.

The sunspot example made use of the **TemporalDataSet**; this is a low-level temporal dataset that is designed to model any "window-based" prediction neural network. A past window is used to provide several values to the neural network from which to make predictions. A future window

specifies the number of elements the neural network should predict into the future.

The stock market example made use of the **MarketNeuralDataSet** class. This class is based on the **TemporalDataSet**. It extends the **TemporalDataSet** to provide the ability to automatically download financial information from Yahoo Finance. This is a very simple example to show the foundation of applying neural networks to the stock market. Investment decisions should not be made based on this network.

This chapter shows how to use datasets that are specifically designed for predictive neural networks. Encog includes a number of specialized datasets for this purpose. In the next chapter you will see how to use datasets that are specifically designed for images.

Questions for Review

1. Is there any difference in structure between a temporal neural network and the feedforward neural networks that have been used in earlier chapters of this book?

2. You would like to attempt to predict the next number in the sequence [1,8,2,5,2,6,7,8]. You are going to use a past window of 4 and a future window of 2. Do not worry about normalization. What are the first two inputs and ideal elements in your training set.

3. What is the purpose of a **TemporalDataDescription** object?

4. What financial data items can Encog currently use for stock market prediction?

5. Why is a small, future window size of one is generally acceptable, yet the past window size must be considerably bigger.

6. Why will the error rate initially spike when training a neural network using resilient propagation for the second time?

7. In what form is any **TemporalDataSet** training set stored to an EG file?

8. What is the purpose of a **TemporalPoint** object?

9. When would you use the RAW type when dealing with a **TemporalDataDescription**?

10. It can be very difficult for a stock market neural network to predict exact price shifts. What is something that is easier to predict, and how is this value calculated?

Terms

Future Window

Past Window

Temporal Data

Temporal Neural Network

Window

Chapter 11: Using Image Data

- Processing Images
- Finding the Bounds
- Downsampling
- Using the Image Dataset

Using images and image recognition in neural networks is very common. In this chapter we will see how to use images with Encog. By using the same feedforward and self-organizing maps, as seen in earlier chapters, we can create neural networks that recognize certain images. Specialized datasets simply make it easier to get image data into the neural network.

Encog provides specialized datasets to make it easier to process different types of data. The same type of underlying neural network handles each type. Actually getting the data converted to a form that is usable by the neural network can be a challenge. In the last chapter you saw how the **TemporalNeuralDataSet** and **MarketDataSet** classes made it easier to process temporal and market data.

This chapter will introduce the **ImageNeuralDataSet**. This class can accept a list of images that will be loaded and processed into a form that is useful for Encog. The **ImageNeuralDataSet** is based upon the **BasicNeuralDataSet**, which is really just an array of double values for input and idea. The **ImageNeuralDataSet** simply adds special functions to load images into arrays of doubles.

There are several important issues to consider when loading image data into a neural network. The **ImageNeuralDataSet** takes care of two important aspects of this. The first aspect is detecting boundaries. It is important to find the boundaries for what you are actually trying to recognize. The second is downsampling. Images are usually formatted in high-resolution and must be downsampled to a consistent lower resolution to be fed to the neural network.

Finding the Bounds

An image is a rectangular region. This is the data that your neural network has to deal with. Only a part of the image may be useful to you. Ideally, the actual image you are trying to recognize takes up the entire physical image that is provided to your neural network. Such is the case with Figure 11.1.

Figure 11.1: An X Drawn Over the Entire Drawing Area

As you can see in the above figure, the letter "X" was drawn over nearly the entire physical image. This image would require minimal, if any, boundary detection.

Images will not always be so perfectly created. Consider the image presented in Figure 11.2.

Figure 11.2: An Off-Center, Off-Scale X

Here the letter "X" is scaled differently than in the previous image. Figure 11.2 is also off-center. We need to find the bounds of the second letter "X" to properly recognize it. Figure 11.3 shows a bounding box around the letter "X". Only data inside of the bounding box will be used to recognize the image.

Figure 11.3: The X with its Bounds Detected

As you can see, the bounds have been detected for the letter "X". The bounding box signifies that only data inside of that box will be recognized. Now the "X" is in approximately the same orientation as Figure 11.1.

Downsampling an Image

Even with bounding boxes, images may not be of a consistent size. The letter "X" in Figure 11.3 is considerably smaller than Figure 11.1. When we recognize the image, we will essentially draw a grid over the image and line up each grid cell to an input neuron. To do this, the images must be of a consistent size. Further, most images have a resolution that is too high to be used with a neural network.

Downsampling solves both of these problems. By using downsampling we both reduce the image to a lower resolution and scale all images to a consistent size. To see this in action, consider Figure 11.4. This figure shows the Encog logo at full resolution.

Figure 11.4: The Encog Logo at Full Resolution

Figure 11.5 shows this same image downsampled.

Figure 11.5: The Encog Logo Downsampled

Do you notice the grid-like pattern? It has been reduced to 32x32 pixels. These pixels would form the input to a neural network. This neural network would require 1,024 input neurons, if the network were to only look at the intensity of each square. Looking at the intensity only causes the neural network to see in "black and white".

If you would like the neural network to see in color, then it is necessary to provide red, green and blue (RGB) values for each of these pixels. This would mean three input neurons for each pixel, which would push our input neuron count to 3,072.

The Encog image dataset provides both boundary detection, as well as RGB and intensity downsampling. In the next section, the Encog image dataset will be introduced.

What to do with the Output Neurons

The output neurons should represent the groups that these images will fall into. For example, if you are writing an OCR application, you could have one output neuron for every character to be recognized. You might also use equilateral encoding, as discussed in Chapter 6, "Obtaining Data for Encog".

If you are using supervised training you will also generate ideal output data for each of the images. These ideal outputs train the neural network for what the image actually is. Whether you are using supervised or unsupervised training, the output neurons will tell you what the neural network thought that the image was.

Using the Encog Image Dataset

Before you can instantiate an **ImageNeuralDataSet** object you must create a downsampled object. All Encog downsampled objects must implement the interface **Downsample**. Encog currently supports two downsample classes. These classes are listed here.

- **RGBDownsample**
- **SimpleIntensityDownsample**

The **SimpleIntensityDownsample** does not take color into consideration. It simply calculates the brightness or darkness of a pixel. The number of input neurons will be height times width, as there is only one input neuron needed per pixel.

The **RGBDownsample** is the more advanced of the two. This downsample object converts to the resolution that you specify and turns every pixel into a three-color (RGB) input. The total number of input neuron values

produced by this object will be height times width times three. The following code instantiates a **SimpleIntensityDownsample** object. This object will be used to create the training set.

```
downsample = new SimpleIntensityDownsample();
```

Now that you have a downsample object, you are ready to begin using an **ImageNeuralDataSet** class. It must be instantiated with several parameters. The following code does this.

```
this.training = new ImageNeuralDataSet(
  downsample, false, 1, -1);
```

The values 1 and -1 specify the range to which the colors will be normalized. Either the intensity color or the three RGB colors individually will be normalized to this range. The **false** value means that we do not want the dataset to attempt to detect the edges. If this value were **true**, Encog would attempt to detect the edges.

The current Encog edge detection is not very advanced. It looks for one consistent color around the sides of an image and attempts to remove as much of that region as it can. More advanced edge detection will likely be built into future versions of Encog. If you need advanced edge detection, you should attempt to trim the images before sending them to the **ImageNeuralDataSet** object.

Now that the **ImageNeuralDataSet** object has been created, we should add some images to it. To add images to this dataset an **ImageNeuralData** object must be created for each image. The following lines of code will add one image from a file.

```
Image img = ImageIO.read(  [filename of image]  );
ImageNeuralData data = new ImageNeuralData( img );
this.training.add( data, [ideal output] );
```

The image is loaded from a file using the Java **ImageIO** class. This class is provided by Java to read images from files. Any valid Java image object can be used by the dataset.

The ideal output should be specified if you are using supervised training. If you are using unsupervised training, this parameter can be omitted. Once

the **ImageNeuralData** object is instantiated it is added to the dataset. These steps are repeated for every image to be added.

Once all of the images have been loaded, they are ready to be downsampled. To downsample the images call the **downsample** method.

```
this.training.downsample( [downsample height] ,
  [downsample width] );
```

You must specify the downsample height and width. All of the images will be downsampled to this size. After calling the **downsample** method the training data has been generated and can be used to train a neural network.

Image Recognition Example

We will now see how to tie all of the Encog image classes together into an example. Here we will present a generic image recognition program. This example could easily become the foundation of a much more complex image recognition program. This example is driven from a script file. Listing 11.1 shows the type of script file that you might use to drive this program.

Listing 11.1: Image Recognition Script

```
CreateTraining: width:16,height:16,type:RGB
Input: image:./coins/dime.png, identity:dime
Input: image:./coins/dollar.png, identity:dollar
Input: image:./coins/half.png, identity:half dollar
Input: image:./coins/nickle.png, identity:nickle
Input: image:./coins/penny.png, identity:penny
Input: image:./coins/quarter.png, identity:quarter
Network: hidden1:100, hidden2:0
Train: Mode:console, Minutes:1, StrategyError:0.25,
StrategyCycles:50
Whatis: image:./coins/dime.png
Whatis: image:./coins/half.png
Whatis: image:./coins/testcoin.png
```

The syntax used by this script file is very simple. There is a command, followed by a colon. This command is followed by a comma-separated list of parameters. Each parameter is a name-value pair that is also separated by a colon. There are five commands in all: **CreateTraining**, **Input**, **Network**, **Train** and **WhatIs**.

The **CreateTraining** command creates a new training set. Here you specify the downsample height and width. The type of downsample is also specified. It can be RGB or Brightness.

The **Input** command inputs a new image for training. Each input command specifies the image, as well as the identity of the image. Multiple images can have the same identity. For example, the above script could have provided a second image of a dime by causing the second **Input** command to also have the identity of "dime".

The **Network** command creates a new neural network for training and recognition. There are two parameters that specify the size of the first and second hidden layers. If you do not wish to have a second hidden layer, specify zero for the **hidden2** parameter.

The **Train** command trains the neural network. The mode specifies whether you want console or GUI training. The **minutes** parameter specifies how many minutes are required to train the network. This parameter is only used with console training; for GUI training this parameter should be set to zero. The strategy tells the training algorithm how many cycles to wait to reset the neural network if the error level has not dropped below the specified amount.

The **WhatIs** command accepts an image and tries to recognize it. The example will print the identity of the image that it thought was most similar.

We will now take a look at the image recognition example. This example is shown in Listing 11.2.

Listing 11.2: The Image Recognition Example

```
package org.encog.examples.neural.image;

import java.awt.Image;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;

import javax.imageio.ImageIO;

import org.encog.EncogError;
import org.encog.neural.data.NeuralData;
import org.encog.neural.data.basic.BasicNeuralData;
import org.encog.neural.data.image.ImageNeuralData;
import org.encog.neural.data.image.ImageNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.training.propagation.resilient
  .ResilientPropagation;
import org.encog.neural.networks.training.strategy.ResetStrategy;
import org.encog.util.downsample.Downsample;
import org.encog.util.downsample.RGBDownsample;
import org.encog.util.downsample.SimpleIntensityDownsample;
import org.encog.util.logging.Logging;
import org.encog.util.simple.EncogUtility;

public class ImageNeuralNetwork {

  class ImagePair {

    private final File file;
    private final int identity;

    public ImagePair(final File file, final int identity) {
      super();
      this.file = file;
      this.identity = identity;
    }

    public File getFile() {
      return this.file;
    }

    public int getIdentity() {
      return this.identity;
    }
  }

  public static void main(final String[] args) {
    Logging.stopConsoleLogging();
    if (args.length < 1) {
```

```
    System.out.println(
      "Must specify command file.  See source for format.");
    } else {
      try {
        final ImageNeuralNetwork program =
          new ImageNeuralNetwork();
        program.execute(args[0]);
      } catch (final Exception e) {
        e.printStackTrace();
      }
    }
  }

  private final List<ImagePair> imageList =
    new ArrayList<ImagePair>();
  private final Map<String, String> args =
    new HashMap<String, String>();
  private final Map<String, Integer> identity2neuron =
    new HashMap<String, Integer>();
  private final Map<Integer, String> neuron2identity =
    new HashMap<Integer, String>();
  private ImageNeuralDataSet training;
  private String line;
  private int outputCount;
  private int downsampleWidth;
  private int downsampleHeight;
  private BasicNetwork network;

  private Downsample downsample;

  private int assignIdentity(final String identity) {

  if (this.identity2neuron.containsKey(identity.toLowerCase())) {
    return this.identity2neuron.get(identity.toLowerCase());
  }

  final int result = this.outputCount;
  this.identity2neuron.put(identity.toLowerCase(), result);
  this.neuron2identity.put(result, identity.toLowerCase());
  this.outputCount++;
  return result;
  }

  public void execute(final String file) throws IOException {
  final FileInputStream fstream = new FileInputStream(file);
  final DataInputStream in = new DataInputStream(fstream);
```

```
  final BufferedReader br = new BufferedReader(
    new InputStreamReader(in));

  while ((this.line = br.readLine()) != null) {
    executeLine();
  }

  in.close();
}

private void executeCommand(final String command,
  final Map<String, String> args) throws IOException {

  if (command.equals("input")) {
    processInput();
  } else if (command.equals("createtraining")) {
    processCreateTraining();
  } else if (command.equals("train")) {
    processTrain();
  } else if (command.equals("network")) {
    processNetwork();
  } else if (command.equals("whatis")) {
    processWhatIs();
  }
}

public void executeLine() throws IOException {
  final int index = this.line.indexOf(':');

  if (index == -1) {
    throw new EncogError("Invalid command: " + this.line);
  }

  final String command = this.line.substring(
    0, index).toLowerCase().trim();

  final String argsStr = this.line.substring(index + 1).trim();
  final StringTokenizer tok = new StringTokenizer(argsStr, ",");
  this.args.clear();

  while (tok.hasMoreTokens()) {
    final String arg = tok.nextToken();
    final int index2 = arg.indexOf(':');
    if (index2 == -1) {
      throw new EncogError("Invalid command: " + this.line);
  }
```

```
  final String key = arg.substring(0,
    index2).toLowerCase().trim();

  final String value = arg.substring(index2 + 1).trim();
  this.args.put(key, value);
  }

executeCommand(command, this.args);
}

private String getArg(final String name) {

  final String result = this.args.get(name);
  if (result == null) {
    throw new EncogError("Missing argument " + name
      + " on line: " + this.line);
  }
  return result;
}

private void processCreateTraining() {
  final String strWidth = getArg("width");
  final String strHeight = getArg("height");
  final String strType = getArg("type");

  this.downsampleHeight = Integer.parseInt(strWidth);
  this.downsampleWidth = Integer.parseInt(strHeight);

  if (strType.equals("RGB")) {
    this.downsample = new RGBDownsample();
  } else {
    this.downsample = new SimpleIntensityDownsample();
  }

  this.training = new ImageNeuralDataSet(this.downsample,
    false, 1, -1);

  System.out.println("Training set created");
}

private void processInput() throws IOException {

  final String image = getArg("image");
  final String identity = getArg("identity");
```

```
  final int idx = assignIdentity(identity);
  final File file = new File(image);

  this.imageList.add(new ImagePair(file, idx));

  System.out.println("Added input image:" + image);
}

private void processNetwork() throws IOException {

  System.out.println("Downsampling images...");

  for (final ImagePair pair : this.imageList) {

    final NeuralData ideal =
      new BasicNeuralData(this.outputCount);
    final int idx = pair.getIdentity();

    for (int i = 0; i < this.outputCount; i++) {
      if (i == idx) {
        ideal.setData(i, 1);
      } else {
        ideal.setData(i, -1);
      }
    }

    final Image img = ImageIO.read(pair.getFile());
    final ImageNeuralData data = new ImageNeuralData(img);
    this.training.add(data, ideal);
  }

  final String strHidden1 = getArg("hidden1");
  final String strHidden2 = getArg("hidden2");

  this.training.downsample(
    this.downsampleHeight, this.downsampleWidth);

  final int hidden1 = Integer.parseInt(strHidden1);
  final int hidden2 = Integer.parseInt(strHidden2);

  this.network = EncogUtility.simpleFeedForward(
    this.training.getInputSize(), hidden1, hidden2,
    this.training.getIdealSize(), true);
  System.out.println("Created network: "
    + this.network.toString());
}
```

```
private void processTrain() throws IOException {

  final String strMode = getArg("mode");
  final String strMinutes = getArg("minutes");
  final String strStrategyError = getArg("strategyerror");
  final String strStrategyCycles = getArg("strategycycles");

  System.out.println("Training Beginning... Output patterns="
    + this.outputCount);

  final double strategyError =
    Double.parseDouble(strStrategyError);
  final int strategyCycles = Integer.parseInt(strStrategyCycles);

  final ResilientPropagation train =
    new ResilientPropagation(this.network, this.training);
    train.addStrategy(
      new ResetStrategy(strategyError, strategyCycles));

  if (strMode.equalsIgnoreCase("gui")) {
    EncogUtility.trainDialog(train, this.network, this.training);
  } else {
    final int minutes = Integer.parseInt(strMinutes);
    EncogUtility.trainConsole(train, this.network, this.training,
    minutes);
  }
  System.out.println("Training Stopped...");
}

public void processWhatIs() throws IOException {
  final String filename = getArg("image");
  final File file = new File(filename);
  final Image img = ImageIO.read(file);
  final ImageNeuralData input = new ImageNeuralData(img);
    input.downsample(this.downsample, false,
    this.downsampleHeight, this.downsampleWidth, 1, -1);
  final int winner = this.network.winner(input);
  System.out.println("What is: "
    + filename + ", it seems to be: "
    + this.neuron2identity.get(winner));
  }
}
```

Some of the code in the above listing deals with parsing the script file and arguments. Because string parsing is not really the focus of this book we will focus on how each of the commands is carried out and how the neural network is constructed. In the next sections we will see how each of these commands was implemented.

Creating the Training Set

The **CreateTraining** command is implemented by the **processCreateTraining** method. This method is shown here.

```
private void processCreateTraining() {
```

The **CreateTraining** command takes three parameters. The following lines read these parameters.

```
  final String strWidth = getArg("width");
  final String strHeight = getArg("height");
  final String strType = getArg("type");
```

The **width** and **height** parameters are both integers and need to be parsed.

```
  this.downsampleHeight = Integer.parseInt(strWidth);
  this.downsampleWidth = Integer.parseInt(strHeight);
```

We must now create the downsample object we are to use. If the mode is RGB then use **RGBDownsample**, otherwise use **SimpleIntensityDownsample**.

```
  if (strType.equals("RGB")) {
    this.downsample = new RGBDownsample();
  } else {
    this.downsample = new SimpleIntensityDownsample();
  }
```

The **ImageNeuralDataSet** can now be created.

```
  this.training = new ImageNeuralDataSet(
    this.downsample, false, 1, -1);
  System.out.println("Training set created");
}
```

Now that we have created the training set we can begin inputting images. The next section describes how this is done.

Inputting an Image

The **Input** command is implemented by the **processInput** method. This method is shown here.

```
private void processInput() throws IOException {
```

The **Input** command takes two parameters. The following lines read these parameters.

```
  final String image = getArg("image");
  final String identity = getArg("identity");
```

The identity is a text string that represents what the image is. We keep count of how many unique identities there are and assign an increasing number to each. These unique identities will form the output layer of the neural network. Each unique identity will be assigned an output neuron. When images are later presented to the neural network, the output neuron with the highest output will represent the identification of the image to the network. The **assignIdentity** method is a simple method that assigns this increasing count, and keeps a mapping of the identity strings to their neuron index.

```
final int idx = assignIdentity(identity);
```

A **File** object is created to hold the image. This will later be used to also read the image.

```
  final File file = new File(image);
```

At this point we do not wish to actually load the individual images. We will simply make note of the image, by saving an **ImagePair** object. The **ImagePair** object links the image to its output neuron index number. The **ImagePair** class is not something built into Encog. It is a structure used by this example to map the images.

```
  this.imageList.add(new ImagePair(file, idx));
```

Finally, we display a message that tells us that the image has been added.

```
  System.out.println("Added input image:" + image);
}
```

Once the images have all been added, we know how many output neurons we have. We can now create the actual neural network. Creating the neural network is handled in the next section.

Creating the Network

The **Network** command is implemented by the **processInput** method. This method is shown here.

```
private void processNetwork() throws IOException {
```

We will begin by downsampling the images. We will loop over every **ImagePair** previously created.

```
  System.out.println("Downsampling images...");

  for (final ImagePair pair : this.imageList) {
```

We create a new **BasicNeuralData** to hold the ideal output for each of the output neurons.

```
    final NeuralData ideal = new
      BasicNeuralData(this.outputCount);
```

The output neuron that corresponds to the identity of the image currently being trained will be set to 1. All other output neurons will be set to -1.

```
    final int idx = pair.getIdentity();
    for (int i = 0; i < this.outputCount; i++) {
    if (i == idx) {
      ideal.setData(i, 1);
    }
    else {
      ideal.setData(i, -1);
    }
  }
```

The input data for this training set item will be the downsampled image. We first load the image into a Java **Image** object.

```
  final Image img = ImageIO.read(pair.getFile());
```

We create an **ImageNeuralData** object to hold this image, and add it to the training set.

```
  final ImageNeuralData data = new ImageNeuralData(img);
  this.training.add(data, ideal);
  }
```

There are two parameters provided to the **Network** command. They specify the number of neurons in each of the two hidden layers. If the second hidden layer has no neurons, then we have a single hidden layer.

```
  final String strHidden1 = getArg("hidden1");
  final String strHidden2 = getArg("hidden2");
  final int hidden1 = Integer.parseInt(strHidden1);
  final int hidden2 = Integer.parseInt(strHidden2);
```

We are now ready to downsample all of the images.

```
  this.training.downsample(
    this.downsampleHeight, this.downsampleWidth);
```

Finally, we create the new neural network according to the parameters specified. The final true parameter specifies that we would like to use a hyperbolic tangent activation function.

```
  this.network = EncogUtility.simpleFeedForward(
    this.training.getInputSize(),
    hidden1, hidden2,
    this.training.getIdealSize(), true);
```

Once the network is created, we report that it is done by printing a message.

```
  System.out.println("Created network: " +
    this.network.toString());
}
```

Now that the network has been created, it can be trained. Training is handled in the next section.

Training the Network

The **Train** command is implemented by the **processTrain** method. This method is shown here.

```
private void processTrain() throws IOException {
```

The **Train** command takes four parameters. The following lines read these parameters.

```
  final String strMode = getArg("mode");
  final String strMinutes = getArg("minutes");
  final String strStrategyError = getArg("strategyerror");
  final String strStrategyCycles = getArg("strategycycles");
```

Once the parameters are read, we display a message stating that training has begun.

```
  System.out.println("Training Beginning... Output patterns="
   + this.outputCount);
```

We parse the two strategy parameters.

```
  final double strategyError =
Double.parseDouble(strStrategyError);
  final int strategyCycles = Integer.parseInt(strStrategyCycles);
```

The neural network is initialized to random weight and threshold values. Sometimes the random set of weights and thresholds will cause the neural network training to stagnate. In this situation, it is best to reset to a new set of random values and begin training again.

We begin training by creating a new **ResilientPropagation** trainer. RPROP training was covered in Chapter 5, "Propagation Training".

```
final ResilientPropagation train =
  new ResilientPropagation(this.network, this.training);
```

Encog allows training strategies to be added to handle situations such as this. One particularly useful training strategy is the **ResetStrategy**. This strategy takes two parameters. The first states the minimum error that the network must achieve before it will be automatically reset to new random values. The second parameter specifies the number of cycles that the

network is allowed to achieve this error rate. If the specified number of cycles is reached, and the network is not at the required error rate, then the weights and thresholds will be randomized.

Encog supports a number of different training strategies. Training strategies enhance whatever training method you are using. They allow minor adjustments to be made as training progresses. Encog supports the following strategies:

- **Greedy**
- **HybridStrategy**
- **ResetStrategy**
- **SmartLearningRate**
- **SmartMomentum**
- **StopTrainingStrategy**

The **Greedy** strategy only allows a training iteration to save its weight and threshold changes if the error rate was improved. The **HybridStrategy** allows a backup training method to be used, if the primary training method stagnates. The hybrid strategy will be explained in Chapter 12, "Recurrent Neural Networks". The **ResetStrategy** resets the network if it stagnates. The **SmartLearningRate** and **SmartMomentum** strategies are used with backpropagation training to attempt to automatically adjust momentum and learning rate. The **StopTrainingStrategy** stops training if it has reached a certain level.

The following lines of code add a reset strategy.

```
train.addStrategy(
   new ResetStrategy(strategyError, strategyCycles));
```

If we are truing using the GUI, then we must use **trainDialog**, otherwise we should use **trainConsole**.

```
 if (strMode.equalsIgnoreCase("gui")) {
   EncogUtility.trainDialog(train, this.network, this.training);
 } else {
   final int minutes = Integer.parseInt(strMinutes);
   EncogUtility.trainConsole(train, this.network, this.training,
     minutes);
```

```
  }
```

Notify the user that training has stopped by displaying a message such as the one shown below. The training process stops when it is canceled by the dialog, or in the case of GUI mode, has been canceled.

```
System.out.println("Training Stopped...");
}
```

Once the neural network is trained, we are ready to recognize images. This is discussed in the next section.

Recognizing Images

The **WhatIs** command is implemented by the **processWhatIs** method. This method is shown here.

```
public void processWhatIs() throws IOException {
```

The **WhatIs** command takes one parameter. The following lines read this parameter.

```
  final String filename = getArg("image");
```

The image is then loaded into an **ImageNeuralData** object.

```
  final File file = new File(filename);
  final Image img = ImageIO.read(file);
  final ImageNeuralData input = new ImageNeuralData(img);
```

The image is downsampled to the correct dimensions.

```
  input.downsample(this.downsample, false, this.downsampleHeight,
    this.downsampleWidth, 1, -1);
```

The downsampled image is presented to the neural network, and we find out what neuron is the winner. The winning neuron is the neuron with the greatest output for the pattern that was presented. This is simple "one-of" normalization, as we discussed in Chapter 6. Chapter 6 also introduced equilateral normalization, which could also be used.

```
  final int winner = this.network.winner(input);
 System.out.println("What is: " + filename + ", it seems to be: "
  + this.neuron2identity.get(winner));
}
```

Finally, we display what the neural network recognized the pattern as.

This example demonstrated a simple script-based image recognition program. This application could easily be used as the starting point for other more advanced image recognition applications. One very useful extension that could be made to this application would be the ability to load and save the trained neural network.

Summary

In this chapter you saw how to use images as input into Encog. Nearly any of the neural network types discussed in this book can be used to recognize images. The classes provided by Encog primarily process the image data into a form that is usable for a neural network, rather than defining the actual structure of the neural network.

The classes provided by Encog for image handling provide several very important functions. Bounds detection and downsampling are two very important image functions that are provided by Encog.

Bounds detection is the process where unimportant parts of an image are trimmed out. Encog supports simple bounds checking where a background of a consistent color can be removed. This prevents the placement of an object, within the input image, from impairing the ability of the neural network to recognize the image. If bounds detection is used, it should not matter if the image to recognize is in the upper left or bottom right corner.

Downsampling is the process where the resolution of an image is decreased. Images can be very high-resolution. They also usually consist of a large amount of color. Encog provides downsampling to deal with both issues. Images can be decreased to a much lower resolution so that we do not need quite as many input neurons to deal with all of the information. Downsampling can also discard color information and deal only with intensity.

So far, all of the neural networks we have seen have been either feedforward or self-organizing maps. Feedforward neural networks form connections in a forward manor. This is not the only type of neural network structure. Connections can also be made backwards, to previous layers. Such neural networks are called recurrent neural networks. In the next chapter you will see how to construct recurrent neural networks with Encog.

Questions for Review

1. Describe the purpose of border detection in image processing.

2. You are creating a neural network that will recognize images downsampled to 32x32 using RGB encoding. How many input neurons would such a neural network have?

3. You are creating a neural network that will recognize images downsampled to 16x16 using intensity only without colors. It should recognize 10 digits. How many input and output neurons would such a neural network have?

4. You would like to create a neural network that would recognize the 50 state flags of the United States of America. You would like to use equilateral normalization to represent each of the 50 individual flags. How many output neurons would you have?

5. What is the purpose of the reset training strategy in Encog?

6. Describe the purpose of downsampling in image processing.

7. The methods **winner** and **compute** are both available from the **BasicNetwork** class. What is the difference between these two methods?

8. What is the purpose of an Encog training strategy?

9. You would like to create a neural network that recognizes the following US coins: penny, nickel, dime, quarter, and half-dollar. You have several samples of each coin. How many output neurons would you have, using simple "one-of" normalization? How would you represent multiple samples of each coin?

10. You have a high-resolution image that is 10,000 x 10,000 pixels. If you were downsample it to 16x16 using RGB, how many output neurons would it require?

Terms

Bounding Box

Crop

Downsample

Intensity Downsample

RGB

RGB Downsample

Scaling

Chapter 12: Recurrent Neural Networks

- Implementing a Recurrent Neural Network in Encog
- Understanding Thermal Neural Networks
- Understanding the Elman Neural Network
- Understanding the Jordan Neural Network

We have primarily looked at feedforward neural networks so far in this book. Even the self-organizing map networks only formed forward connections. All connections in a neural network do not need to be forward. It is also possible to create recurrent connections. A recurrent connection is a connection from one layer to either itself, or some other pervious layer.

There are a total of four different common recurrent neural network architectures that are directly supported by Encog. These are only the commonly recognized recurrent architectures; there are many additional network architectures that can be supported by combining Encog's layer and synapse types. The four common recurrent neural network types supported by Encog are listed here.

- Boltzmann Machine
- Elman Neural Network
- Hopfield Neural Network
- Jordan Neural Network

Of these neural network types, two of the simplest are the Hopfield and Boltzmann machine neural networks. These are commonly referred to as thermal neural networks, because the energy level of the neural network plays an important role in their operation. Thermal neural networks will be covered in this chapter.

Two more complex types of recurrent neural network are the Elman and Jordan neural networks. They are both similar recurrent neural networks that differ slightly on how the recurrent connection is made. Later in this chapter you learn how to create both an Elman and a Jordan neural network.

All four of these recurrent neural network types will be covered in this chapter. We will begin with the thermal neural network types.

Encog Thermal Neural Networks

The thermal neural networks supported by Encog include the Hopfield and Boltzmann Machine neural networks. Both of these two neural network types are recurrent in the fact that they are self-connected. Every neuron in a layer is connected to the other neurons in the same layer. In this section we will look at both the Hopfield and Boltzmann Machine neural networks.

The thermal neural networks used by Encog are different from the networks seen so far in how input is presented to them. The thermal neural networks are given an initial pattern, this pattern is presented to the input neurons, and a new pattern is received from the output neurons. So far, this is the same process used in the networks we have seen thus far. However, this output is then fed back into the input. Because the input layer and output layers are the same, this is always possible, as they will have the same number of neurons.

This cycle continues. As a result, the thermal networks have a sort of "state" that is kept. The output pattern is continually cycled through the thermal network. The thermal network will eventually converge to a constant pattern. Thermal networks also have an energy or temperature. As the network progresses to the final stabilized pattern, the temperature decreases.

What determines how this stabilized pattern is reached, depends on the network being used. Hopfield and Boltzmann machine neural networks both handle this differently.

The Encog thermal networks operate on bipolar numbers. Bipolar numbers are a mathematical way of representing true and false. In bipolar form, true is represented as 1 and false is represented as -1. This allows the binary numbers to be represented numerically in a way that true and false are opposites. Both Hopfield and Boltzmann machine neural networks make use of bipolar numbers. In the next section, we will look at Hopfield neural networks.

Understanding Hopfield Neural Networks

The Hopfield neural network is a very simple form of neural network. A Hopfield neural network consists of a single layer of neurons that are fully

connected to each other. The layer does not have threshold values. However, there is no connection between a neuron and itself. Figure 12.1 shows a four-neuron, Hopfield neural network.

Figure 12.1: Neurons of a Hopfield Neural Network

As you can see, the above neural network has 12 connections. Every neuron is connected to every other neuron. But there are no self-connections. The Hopfield neural network looks very simple when modeled in the Encog Workbench as shown in Figure 12.2.

Figure 12.2: The Hopfield Neural Network

As you can see from the above diagram, there is a single layer that functions as both the input and output layer of the neural network. The activation function used is bipolar. This means that the neural network will only output -1 or 1 for each of the output neurons.

A Hopfield neural network can be trained for a number of patterns. The Hopfield network will then converge towards one of these trained patterns when presented with a new pattern. We will now look at an example neural network that converges to a trained pattern. This example as introduced back in Chapter 2 as Listing 2.4. We will now examine how this example works.

This program trains the neural network for a number of patterns. The program then tests the Hopfield neural network by presenting these patterns back. The Hopfield network always converges back to the training pattern that most closely matches the input. Therefore, when presented with the exact input of a training pattern, the training pattern that is recognized is returned. This is called auto-association. The network returns the entire contents of the pattern that is associated with the input.

In addition to presenting the training patterns back to the network, some distorted versions of the training patterns are also returned. The program takes the distorted image and converges to the closest training pattern match. The output from this program is as follows:

```
Cycles until stable(max 100): 1, result=
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
----------------------
Cycles until stable(max 100): 1, result=
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
  OO  OO     ->     OO  OO
  OO  OO     ->     OO  OO
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
  OO  OO     ->     OO  OO
  OO  OO     ->     OO  OO
OO  OO  OO   ->   OO  OO  OO
OO  OO  OO   ->   OO  OO  OO
----------------------
Cycles until stable(max 100): 1, result=
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
OOOOO        ->   OOOOO
     OOOOO   ->        OOOOO
```

```
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
     OOOOO   ->        OOOOO
----------------------
Cycles until stable(max 100): 1, result=
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
 O  O  O     ->    O  O  O
  O  O  O    ->     O  O  O
O  O  O  O   ->   O  O  O  O
----------------------
Cycles until stable(max 100): 1, result=
OOOOOOOOOO   ->   OOOOOOOOOO
O        O   ->   O        O
O OOOOOO O   ->   O OOOOOO O
O O    O O   ->   O O    O O
O O OO O O   ->   O O OO O O
O O OO O O   ->   O O OO O O
O O    O O   ->   O O    O O
O OOOOOO O   ->   O OOOOOO O
O        O   ->   O        O
OOOOOOOOOO   ->   OOOOOOOOOO
----------------------
Cycles until stable(max 100): 2, result=
             ->   O O O O O
             ->    O O O O O
             ->   O O O O O
             ->    O O O O O
             ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
O O O O O    ->   O O O O O
 O O O O O   ->    O O O O O
----------------------
Cycles until stable(max 100): 2, result=
OOO O    O   ->   OO  OO  OO
 O  OOO OO   ->   OO  OO  OO
  O O OO O   ->     OO  OO
 OOO   O     ->     OO  OO
```

```
OO  O  OOO   ->   OO  OO  OO
 O OOO   O   ->   OO  OO  OO
O OO  O  O   ->     OO  OO
   O OOO     ->     OO  OO
OO OOO  O    ->   OO  OO  OO
 O  O  OOO   ->   OO  OO  OO
----------------------
Cycles until stable(max 100): 2, result=
OOOOO        ->   OOOOO
O   O OOO    ->   OOOOO
O   O OOO    ->   OOOOO
O   O OOO    ->   OOOOO
OOOOO        ->   OOOOO
     OOOOO   ->        OOOOO
 OOO O   O   ->        OOOOO
 OOO O   O   ->        OOOOO
 OOO O   O   ->        OOOOO
     OOOOO   ->        OOOOO
----------------------
Cycles until stable(max 100): 2, result=
O  OOOO  O   ->   O  O  O  O
OO  OOOO     ->    O  O  O
OOO  OOOO    ->     O  O  O
OOOO  OOOO   ->   O  O  O  O
 OOOO  OOO   ->    O  O  O
  OOOO  OO   ->     O  O  O
O  OOOO  O   ->   O  O  O  O
OO  OOOO     ->    O  O  O
OOO  OOOO    ->     O  O  O
OOOO  OOOO   ->   O  O  O  O
----------------------
Cycles until stable(max 100): 2, result=
OOOOOOOOOO   ->   OOOOOOOOOO
O        O   ->   O        O
O        O   ->   O OOOOOO O
O        O   ->   O O    O O
O   OO   O   ->   O O OO O O
O   OO   O   ->   O O OO O O
O        O   ->   O O    O O
O        O   ->   O OOOOOO O
O        O   ->   O        O
OOOOOOOOOO   ->   OOOOOOOOOO
----------------------
```

You will notice, from the output above, that only one cycle is needed for the nondistorted patterns. This is because they are identical to the training pattern. Two cycles are needed for the distorted images to converge to a training pattern. For more heavily distorted patterns it may take even more cycles.

This program deals with 10x10 patterns. This size pattern uses a 100 neuron Hopfield network. These patterns are represented as string arrays to make it easier. For example, one pattern is represented as follows.

```
public String[][] PATTERN  = { {
            "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O",
        "O O O O O ",
        " O O O O O"  },
```

This pattern has 100 characters in it. To convert it to a form that is useful for the neural network, every space becomes a -1 and every “0” becomes a 1.

Most of the processing in this example is done inside of the **Execute** method.

```
public void Execute(IExampleInterface app)
{
  this.app = app;
```

To create the Hopfield neural network we will use an Encog pattern. There are Encog patterns for most common neural network architectures supported by Encog. It is not necessary to use a pattern. We could add, and properly connect, all of the layers and synapses to create the network. However, a pattern makes it easier. The following code uses a pattern to create a Hopfield neural network.

```
  HopfieldPattern pattern = new HopfieldPattern();
  pattern.InputNeurons = WIDTH * HEIGHT;
  BasicNetwork hopfield = pattern.Generate();
```

If you would like to see the actual code necessary to create a Hopfield neural network, without a pattern, refer to Appendix C. This appendix shows every pattern that Encog supports, along with the actual code needed to create any of these neural network types.

Recall from Chapter 2, "The Parts of an Encog Neural Network" that Encog neural networks make use of a logic class to tell Encog how to process the neural network. Hopfield neural networks make use of the class **HopfieldLogic**.

```
  HopfieldLogic hopfieldLogic = (HopfieldLogic)hopfield.Logic;
```

Next, we train the Hopfield neural network. Training a Hopfield neural network is very different from the sort of training that we performed with other network types in this book. Training a Hopfield neural network does not involve the many repetitions that other networks require. Due to the very simple, single-layer nature of the Hopfield neural network they are trained with a simple mathematical process. This is performed by the **AddPattern** method. The following lines of code do this.

```
for (int i = 0; i < PATTERN.Length; i++)
{
  hopfieldLogic.AddPattern(ConvertPattern(PATTERN, i));
}
```

The **PATTERN** array contains the patterns on which the network will be trained. The **ConvertPattern** method converts the pattern strings into bipolar -1 and 1 values.

Finally, we call the **Evaluate** method twice to evaluate both the training patterns and the distorted patterns.

```
  Evaluate(hopfield,PATTERN);
  Evaluate(hopfield,PATTERN2);
}
```

The **Evaluate** method simply loops over an array and presents every pattern to the neural network. The **Evaluate** method is shown here.

```
public void Evaluate(BasicNetwork hopfield, String[][] pattern)
{
```

Most neural networks we have seen so far make use of the **Compute** method of the **BasicNetwork** class. This method could be used to run a single cycle of a Hopfield neural network. However, the **HopfieldLogic** class provides a convenience method called **RunUntilStable**. This is the method we will use here. We begin by obtaining the **HopfieldLogic** object used by the network.

```
HopfieldLogic hopfieldLogic = (HopfieldLogic)hopfield.Logic;
```

We loop over every pattern that was sent to this method.

```
for (int i = 0; i < pattern.Length; i++)
{
```

We convert the pattern into a bipolar array.

```
  BiPolarNeuralData pattern1 = ConvertPattern(pattern, i);
```

The Hopfield network treats the pattern that is cycled through the input and output as the current state. We set this state to be the pattern that we are to evaluate.

```
  hopfieldLogic.CurrentState = pattern1;
```

We will run the Hopfield network until the pattern stabilizes. The pattern is stable once we have had two cycles that did not change the current state. The parameter 100 specifies that we should give it at most 100 cycles to stabilize.

```
  int cycles = hopfieldLogic.RunUntilStable(100);
```

We now look at the stabilized neural network.

```
  BiPolarNeuralData pattern2 =
   (BiPolarNeuralData)hopfieldLogic.getCurrentState();
```

Finally, we display the output from the network, as well as the number of cycles it took to get there.

```
  BiPolarNeuralData pattern2 =
    (BiPolarNeuralData)hopfieldLogic.CurrentState;

  Console.WriteLine("Cycles until stable(max 100): "
    + cycles + ", result=");
```

```
    Display(pattern1, pattern2);
    Console.WriteLine("----------------------");
```

The Hopfield neural network is useful for recognizing simple bipolar patterns. For more complex recognitions, the more advanced feedforward and self-organizing maps, seen earlier in this book, are needed.

Understanding Boltzmann Machines

A Boltzmann machine is a stochastic, recurrent neural network. Boltzmann machines are the stochastic, generative counterparts of Hopfield neural networks. However, unlike a Hopfield neural network, a Boltzmann machine does have threshold values.

Stochastic means that the Boltzmann machine is somewhat random. The degree to which the Boltzmann machine is random depends on the current temperature of the network. As the network cools, it becomes less random. This is a form of the simulated annealing that we saw in Chapter 8, "Other Supervised Training Methods".

Boltzmann machines work with bipolar numbers, just like Hopfield neural networks. Because of this, the Boltzmann machine could be used for the same sort of pattern recognition as seen in the last section. However, for this example we are going to apply it to an optimization problem. We are going to apply it to the traveling salesman problem (TSP) as explained in the next section.

The Traveling Salesman Problem

The traveling salesman problem involves a "traveling salesman" who must visit a certain number of cities. The task is to identify the shortest route for the salesman to travel between the cities. The salesman is allowed to begin and end at any city, but must visit each city once. The salesman may not visit a city more than once.

This may seem like an easy task for a normal iterative program; however, consider the speed with which the number of possible combinations grows as the number of cities increases. If there are one or two cities, only one step is required. Three increases the possible routes to six. The following list shows how quickly these combinations can grow.

```
1 city causes 1 combination.
2 cities cause 1 combination.
3 cities cause 6 combinations.
4 cities cause 24 combinations.
5 cities cause 120 combinations.
6 cities cause 720 combinations.
7 cities cause 5,040 combinations.
8 cities cause 40,320 combinations.
9 cities cause 362,880 combinations.
10 cities cause 3,628,800 combinations.
11 cities cause 39,916,800 combinations.
12 cities cause 479,001,600 combinations.
13 cities cause 6,227,020,800 combinations.
...
50 cities cause 3.041 * 10^64 combinations.
```

The formula behind the above table is the factorial. The number of cities, n, is calculated using the factorial operator (!). The factorial of some arbitrary value n is given by n * (n – 1) * (n – 2) * ... * 3 * 2 * 1. As you can see from the above table, these values become incredibly large when a program must do a "brute force" search. The sample program that we will examine in the next section finds a solution to a 10-city problem in a matter of minutes. We accomplish this by using a Boltzmann machine, rather than a normal, brute-force approach.

Using a Boltzmann Machine for the TSP

We will apply a Boltzmann machine to the TSP. The Boltzmann TSP example is shown in Listing 12.1.

Listing 12.1: Boltzmann Machine and the TSP

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Encog.Neural.NeuralData.Bipolar;
using Encog.Neural.Networks;
using ConsoleExamples.Examples;
using Encog.Neural.Networks.Logic;
using Encog.Neural.Networks.Pattern;

namespace Encog.Examples.Boltzmann
{
    public class BoltzTSP:IExample
```

```
    {

        public const int NUM_CITIES = 10;
        public const int NEURON_COUNT = NUM_CITIES * NUM_CITIES;

        private double gamma = 7;
        private double[][] distance;

        private IExampleInterface app;

        public static ExampleInfo Info
        {
            get
            {
                ExampleInfo info = new ExampleInfo(
                    typeof(BoltzTSP),
                    "tsp-boltzmann",
                    "Boltzmann Machine for the Traveling Salesman
(TSP)",
                    "Use a Boltzmann machine to provide a
solution for the Traveling Salesman Problem (TSP).");
                return info;
            }
        }

        public double Sqr(double x)
        {
            return x * x;
        }

        public void CreateCities()
        {
            double x1, x2, y1, y2;
            double alpha1, alpha2;

            this.distance = new double[NUM_CITIES][];

            for (int n1 = 0; n1 < NUM_CITIES; n1++)
            {
                this.distance[n1] = new double[NUM_CITIES];
                for (int n2 = 0; n2 < NUM_CITIES; n2++)
                {
                    alpha1 = ((double)n1 / NUM_CITIES) * 2
                      * Math.PI;
                    alpha2 = ((double)n2 / NUM_CITIES)
                      * 2 * Math.PI;
```

```
                x1 = Math.Cos(alpha1);
                y1 = Math.Sin(alpha1);
                x2 = Math.Cos(alpha2);
                y2 = Math.Sin(alpha2);
                distance[n1][n2] = Math.Sqrt(Sqr(x1 - x2)
                  + Sqr(y1 - y2));
            }
        }
    }

    public bool IsValidTour(BiPolarNeuralData data)
    {
        int cities, stops;

        for (int n1 = 0; n1 < NUM_CITIES; n1++)
        {
            cities = 0;
            stops = 0;
            for (int n2 = 0; n2 < NUM_CITIES; n2++)
            {
                if (data.GetBoolean(n1 * NUM_CITIES + n2))
                {
                    if (++cities > 1)
                        return false;
                }
                if (data.GetBoolean(n2 * NUM_CITIES + n1))
                {
                    if (++stops > 1)
                        return false;
                }
            }
            if ((cities != 1) || (stops != 1))
                return false;
        }
        return true;
    }

    public double LengthOfTour(BiPolarNeuralData data)
    {
        double result;
        int n1, n2, n3;

        result = 0;
        for (n1 = 0; n1 < NUM_CITIES; n1++)
        {
            for (n2 = 0; n2 < NUM_CITIES; n2++)
```

```
                {
                    if (data.GetBoolean(
                        ((n1) % NUM_CITIES) * NUM_CITIES + n2))
                        break;
                }
                for (n3 = 0; n3 < NUM_CITIES; n3++)
                {
                    if (data.GetBoolean(((n1 + 1) % NUM_CITIES)
                        * NUM_CITIES + n3))
                        break;
                }
                result += distance[n2][n3];
            }
            return result;
        }

        String DisplayTour(BiPolarNeuralData data)
        {
            StringBuilder result = new StringBuilder();

            int n1, n2;
            bool first;

            for (n1 = 0; n1 < NUM_CITIES; n1++)
            {
                first = true;
                result.Append("[");
                for (n2 = 0; n2 < NUM_CITIES; n2++)
                {
                    if (data.GetBoolean(n1 * NUM_CITIES + n2))
                    {
                        if (first)
                        {
                            first = false;
                            result.Append(n2);
                        }
                        else
                        {
                            result.Append(", " + n2);
                        }
                    }
                }
                result.Append("]");
                if (n1 != NUM_CITIES - 1)
                {
                    result.Append(" -> ");
```

```
            }
        }
        return result.ToString();
    }

    public void CalculateWeights(BasicNetwork network)
    {
        int n1, n2, n3, n4;
        int i, j;
        int predN3, succN3;
        double weight;

        BoltzmannLogic logic = (BoltzmannLogic)network.Logic;

        for (n1 = 0; n1 < NUM_CITIES; n1++)
        {
            for (n2 = 0; n2 < NUM_CITIES; n2++)
            {
                i = n1 * NUM_CITIES + n2;
                for (n3 = 0; n3 < NUM_CITIES; n3++)
                {
                    for (n4 = 0; n4 < NUM_CITIES; n4++)
                    {
                        j = n3 * NUM_CITIES + n4;
                        weight = 0;
                        if (i != j)
                        {
                            predN3 = (n3 == 0 ? NUM_CITIES
                              - 1 : n3 - 1);
                            succN3 = (n3 == NUM_CITIES - 1
                              ? 0 : n3 + 1);
                            if ((n1 == n3) || (n2 == n4))
                                weight = -gamma;
                            else if ((n1 == predN3) ||
                                 (n1 == succN3))
                                weight = -distance[n2][n4];
                        }
                        logic.ThermalSynapse.WeightMatrix
                          [i, j] = weight;
                    }
                }
                logic.ThermalLayer.Threshold[i] =
                  (-gamma / 2);
            }
        }
    }
```

```
        public void Execute(IExampleInterface app)
        {
            this.app = app;
            BoltzmannPattern pattern = new BoltzmannPattern();
            pattern.InputNeurons = NEURON_COUNT;
            BasicNetwork network = pattern.Generate();
            BoltzmannLogic logic = (BoltzmannLogic)network.Logic;

            CreateCities();
            CalculateWeights(network);

            logic.Temperature = 100;
            do
            {
                logic.EstablishEquilibrium();
                app.WriteLine(logic.Temperature + " : "
                  + DisplayTour(logic.CurrentState));
                logic.DecreaseTemperature(0.99);
            } while (!IsValidTour(logic.CurrentState));

            app.WriteLine("Final Length: "
              + this.LengthOfTour(logic.CurrentState));
        }
    }
}
```

The output from the program is as follows:

```
100.0 : [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> []
99.0 : [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> []
98.01 : [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> []
97.0299 : [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> []
96.059601 : [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> [] -> []
95.09900499 : [] -> [] -> [] -> [] -> [] -> [2] -> [] -> [] -> [] -> []
…
0.697888643387: [] -> [] -> [5] -> [] -> [] -> [] -> [] -> [2] -> [] -> []
0.690909756953: [7] -> [] -> [] -> [4] -> [3] -> [2] -> [] -> [0] -> [] -> [8]
0.684000659383: [9] -> [] -> [] -> [] -> [7] -> [] -> [3] -> [4] -> [] -> [8]
0.677160652789: [0] -> [] -> [8] -> [7] -> [6] -> [4] -> [5] -> [] -> [2] -> [1]
0.670389046262: [7] -> [] -> [6] -> [] -> [] -> [3] -> [] -> [9] -> [0] -> [8]
0.663685155799: [] -> [] -> [6] -> [] -> [] -> [] -> [] -> [] -> [2] -> [1]
0.657048304241: [8] -> [9] -> [0] -> [1] -> [2] -> [3] -> [4] -> [5] -> [6] -> [7]
Final Length: 6.180339887498949
```

As you can see from the above output, the temperature is slowly decreased until an optimum path through the cities is found. Many paths considered by the network are not even valid, and are shown above with blank cities, which

are square brackets that do not contain numbers. The final, optimized length of the trip is shown.

Structuring the TSP Neural Network

One of the first things we must consider is how to encode the traveling salesman problem as a neural network. We are using a Boltzmann machine, so we have a single layer of neurons. Somehow this single layer of neurons must encode a tour through 10 cities.

To do this, we will use 100 neurons. We have 10 stops and also 10 cities. We will treat the input neurons as a 10x10 grid. The rows will represent the tour stops, and the columns the cities. For one tour stop only one city will be visited. Therefore in each row, only one column should have a true (1) value, the rest should be false (-1) values.

Neurons 1 through 10 would form the first stop. As a result, only one of those neurons should have a true value. Likewise, neurons 11 through 20 will hold the second stop. Neurons 11 through 20 should have a single true neuron and the rest false. This repeats for all 100 neurons.

One very important consideration in this network configuration is that there are many invalid states for the neural network. There are several rules that define a valid neural network. They are summarized here.

- At least one neuron per row must be true.
- No more than one neuron per row can be true.
- Every row must have a unique neuron true; no multiple stops to the same city are permitted.

Checking to see that the state of the current neural network is an important part of this program.

Implementing the Boltzmann Machine

The cities will be arranged in a circle. This allows us to determine if the Boltzmann machine has truly found the optimal path. The Boltzmann machine has no concept of a circle, or how to follow one, so the cities may just as well be randomly placed. However, if you look at the above output, you will see that the cities are visited in order, given their circular nature. The

Boltzmann machine can start with any city it likes. However, if it is following an optimal path, it must visit them sequentially.

The **createCities** method creates a two-dimensional array that holds the distances between the cities. This method is shown here.

```
public void createCities() {
  double x1, x2, y1, y2;
  double alpha1, alpha2;
```

We begin by creating the two dimensional array. This array will end up being symmetrical. That is, the distance between city 1 and city 3 should be the same the other way around (the distance between city 3 and city 1). The array is a simple table. The variable **distance[3][1]** represents the distance between city 2 and city 0. This is because arrays are zero based.

```
  this.distance = new double[NUM_CITIES][NUM_CITIES];
```

We must loop through every array element.

```
  for (int n1 = 0; n1 < NUM_CITIES; n1++) {
    for (int n2 = 0; n2 < NUM_CITIES; n2++) {
```

The variables **alpha1** and **alpha2** represent the angles of the first and second cities. To visualize how we have cities arranged in a circle, visualize the second hand sweeping through a minute. We will take the unit circle, which is 2PI radians, and divide it among the cities.

```
      alpha1 = ((double) n1 / NUM_CITIES) * 2 * Math.PI;
      alpha2 = ((double) n2 / NUM_CITIES) * 2 * Math.PI;
```

Now, using the trigonometric ratios we calculate the x and y coordinates of the two cities.

```
      x1 = Math.cos(alpha1);
      y1 = Math.sin(alpha1);
      x2 = Math.cos(alpha2);
      y2 = Math.sin(alpha2);
```

Finally, we use the distance formula to calculate the distance between the two cities.

```
      distance[n1][n2] = Math.sqrt(sqr(x1 - x2) + sqr(y1 - y2));
      }
```

```
    }
}
```

It is not necessary to understand the trigonometry behind how a circle is calculated. It is simply a convenient way to calculate 10 city locations in a way that allows us to visualize the shortest path. For more information about how the circle was calculated, you can refer to the following Wikipedia article:

http://en.wikipedia.org/wiki/Unit_circle

The distances were calculated using the distance formula. The following Wikipedia article has more information about the distance formula:

http://en.wikipedia.org/wiki/Distance

I will not cover the unit circle and distance formula here, as they have more to do with trigonometry and geometry and they do neural networks.

Now that the distances have been created, we must assign weights to the Boltzmann machine. There are many ways to assign weights to a Boltzmann machine, particularly with the more complex stacked Boltzmann machines. At this point, Encog only supports simple nonstacked/nonrestricted Boltzmann machines. Future versions of Encog will likely enhance Boltzmann machine processing. But for now, we are left with manually assigning weights.

The weights are assigned so that the network will stabilize to a minimum distance among the cities. The method that does this is **calculateWeights**. It is shown here.

```
public void calculateWeights(BasicNetwork network) {
```

First we obtain the **BoltzmannLogic** object. This is an easy way to gain access to the recurrent Boltzmann synapse, the only synapse in the network.

```
  BoltzmannLogic logic = (BoltzmannLogic)network.getLogic();
```

We must now form the weights between all 100 neurons. This forms a 100x100 matrix, or a total of 10,000 numbers. We will form connections between the source and target neurons. We begin by looping over the source neurons. We will loop over every stop on the tour, and all ten cities in each.

```
for (int sourceTour = 0; sourceTour < NUM_CITIES; sourceTour++)
{
  for (int sourceCity = 0; sourceCity < NUM_CITIES;
    sourceCity++) {
```

We need to translate the source tour step number and city number into an index into one of the 100 neurons. The following statement flattens the tour rows and city columns into a neuron number.

```
  int sourceIndex = sourceTour * NUM_CITIES + sourceCity;
```

Likewise, we loop over the target tour stops and target cities.

```
    for (int targetTour = 0; targetTour < NUM_CITIES;
      targetTour++) {
      for (int targetCity = 0; targetCity < NUM_CITIES;
        targetCity++) {
```

We form the same index as we did with the source cities.

```
        int targetIndex = targetTour * NUM_CITIES + targetCity;
```

As the loops progress, the **sourceIndex** and **targetIndex** will visit every combination of the 100 input neurons. There are 10,000 combinations in this matrix.

We will now calculate the weight for this matrix cell. We initialize the **weight** variable to zero.

```
        double weight = 0;
```

If the source and target indexes are equal, then we will calculate no weight. We do not want any self-connected neurons.

```
        if (sourceIndex != targetIndex) {
```

We now calculate the neuron that represents the next stop on the tour, as well as the previous stop on the tour. We handle each of these locally, and calculate towards a "global" solution with all of the steps. However, for the local solution we simply want to minimize the distance between this step and the next/previous steps from it.

```
          int predTargetTour = (targetTour == 0 ?
            NUM_CITIES - 1 : targetTour - 1);
          int succTargetTour = (targetTour == NUM_CITIES - 1
```

```
                ? 0 : targetTour + 1);
```

The constant gamma represents the threshold that a neuron must exceed. If either the tours or the cities match, we set the weight to below the gamma.

```
            if ((sourceTour == targetTour) ||
              (sourceCity == targetCity))
              weight = -gamma;
            else if ((sourceTour == predTargetTour)
              || (sourceTour == succTargetTour))
```

If the target predecessors and successors fall in line, then we set the weight to the inverse of the distance. This will attempt to minimize distance.

```
            weight = -distance[sourceCity][targetCity];
            }
```

Finally, we set the weight to the calculated value.

```
      logic.getThermalSynapse().getMatrix().
        set(sourceIndex, targetIndex, weight);
      }
    }
```

The threshold is set to half of the inverse of gamma. This establishes the amount of input that an output neuron, consisting of a tour step and a city, must exceed to fire.

```
  logic.getThermalLayer().setThreshold(sourceIndex, -gamma / 2);
  }
}
```

This process continues for every connection in the network.

Processing the Boltzmann Machine

The Boltzmann machine is actually processed by the **run** method. This method is shown here.

```
public void run() {
```

First, we create a Boltzmann machine using the **BoltzmannPattern**.

```
  BoltzmannPattern pattern = new BoltzmannPattern();
```

```
  pattern.setInputNeurons(NEURON_COUNT);
  BasicNetwork network = pattern.generate();
  BoltzmannLogic logic = (BoltzmannLogic)network.getLogic();
```

We create the cities, and calculate the weights, as previously discussed.

```
  createCities();
  calculateWeights(network);
```

We set the starting temperature to 100.

```
  logic.setTemperature(100);
```

We will loop until the network settles on a valid tour.

```
  do {
```

The **establishEquilibrium** method is called to perform one cycle of annealing. The weights are randomized according to the decreasing temperature.

```
    logic.establishEquilibrium();
```

Display the current statistics on the temperature and tour components. The temperature is decreased by one percent.

```
    System.out.println(logic.getTemperature()+" : "
      +displayTour(logic.getCurrentState()));
    logic.decreaseTemperature(0.99);
  } while (!isValidTour(logic.getCurrentState()));
```

This process continues until a valid tour is found.

```
  System.out.println("Final Length: "
    + this.lengthOfTour(logic.getCurrentState()) );
}
```

Finally, we display the length of the final tour.

More Complex Boltzmann Machines

At this point, Encog's support of Boltzmann machines is fairly basic. There are two more complex Boltzmann machine concepts that later versions of Encog will likely include. These include restricted Boltzmann machines as well as stacked Boltzmann machines.

Restricted Boltzmann Machines (RBM) divide the neurons into visible and invisible units. It is restrictive in the sense that there are no connections between hidden neurons and other hidden neurons. This greatly increases the efficiency of the Boltzmann machine.

Another technique is stacking. Several RBM's are stacked on top of each other, forming layers. The output from one RMB becomes the input to another. The various layers can be trained independently. This greatly increases the processing power of the RBM.

The Elman Neural Network

The last two neural networks that we looked at made use of bipolar numbers and had a single layer. Elman and Jordan neural networks are a type of recurrent neural network that has additional layers. They function very similarly to the feedforward networks that we saw in previous chapters. They use similar training techniques to feedforward neural networks as well.

Figure 12.3 shows an Elman neural network.

Figure 12.3: The Elman Neural Network

You will notice that the Elman neural network makes use of a **ContextLayer**. The context layer allows feedback. Feedback is when the

output from a previous iteration is used as the input for successive iterations. Notice that a 1:1 layer connects the hidden layer to the context layer. A 1:1 connection requires the same number of neurons in the source and target layers. It has no weights, and thus does not learn. It is simply a conduit for the output from the hidden layer to get to the context layer. The context layer remembers this output and then feeds it back to the hidden layer on the next iteration. Therefore, the context layer is always feeding the hidden layer its own output from the previous iteration.

The connection from the context layer to the hidden layer is weighted. This synapse will learn as the network is trained. You may wonder what value a context layer adds to a neural network. Context layers allow a neural network to recognize context.

To see how important context is to a neural network consider how the previous networks were trained. The order of the training set elements did not really matter. The training set could be jumbled in any way needed, and the network would still train in the same manner. With an Elman or a Jordan neural network the order becomes very important. The training set element previously supported is still affecting the neural network. This becomes very important for predictive neural networks.

This makes Elman neural networks very useful for temporal neural networks. Consider how the temporal neural network in Chapter 10, "Using Temporal Data", was structured. We specified a window size and created input neurons to match this size and created the predictive window. An Elman neural network does not require this predictive window. There is usually a single input neuron for each piece of data used to predict, and a single output neuron for each piece of data predicted.

Dr. Jeffrey Elman created the Elman neural network. Dr. Elman used an XOR pattern to test his neural network. However, he did not use a typical XOR pattern like we've seen in previous chapters. He used a XOR pattern collapsed to just one input neuron. Consider the following XOR truth table.

```
1.0 XOR 0.0 = 1.0
0.0 XOR 0.0 = 0.0
0.0 XOR 1.0 = 1.0
1.0 XOR 1.0 = 0.0
```

We now wish to collapse this to a string of numbers. To do this simply read the numbers left-to-right, line-by-line. This produces the following:

```
1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0
```

We will create a neural network that accepts one number from the above list, and should predict the next number. This same data will be used with a Jordan neural network later in this chapter. Sample input to this neural network would be as follows:

```
Input Neurons: 1.0 → Output Neurons: 0.0
Input Neurons: 0.0 → Output Neurons: 1.0
Input Neurons: 1.0 → Output Neurons: 0.0
Input Neurons: 0.0 → Output Neurons: 0.0
Input Neurons: 0.0 → Output Neurons: 0.0
Input Neurons: 0.0 → Output Neurons: 0.0
```

It would be impossible to train a typical feedforward neural network for this. The training information would be contradictory. Sometimes an input of 0 results in a 1; other times it results in a 0. An input of 1 has similar issues. The neural network needs context; it should look at what comes before. We will look at an example that uses an Elman and a feedforward network to attempt to predict the output.

Listing 12.2: An Elman Neural Network

```
package org.encog.examples.neural.recurrant.elman;

import org.encog.examples.neural.util.TemporalXOR;
import org.encog.neural.activation.ActivationTANH;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.training.CalculateScore;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.TrainingSetScore;
import org.encog.neural.networks.training.anneal.
  NeuralSimulatedAnnealing;
import org.encog.neural.networks.training.propagation.resilient.
  ResilientPropagation;
import org.encog.neural.networks.training.strategy.Greedy;
import org.encog.neural.networks.training.strategy.
  HybridStrategy;
import org.encog.neural.networks.training.strategy.
  StopTrainingStrategy;
import org.encog.neural.pattern.ElmanPattern;
```

```
import org.encog.neural.pattern.FeedForwardPattern;
import org.encog.util.logging.Logging;

public class ElmanXOR {


  static BasicNetwork createElmanNetwork() {
  // construct an Elman type network
    ElmanPattern pattern = new ElmanPattern();
    pattern.setActivationFunction(new ActivationTANH());
    pattern.setInputNeurons(1);
    pattern.addHiddenLayer(2);
    pattern.setOutputNeurons(1);
    return pattern.generate();
  }

  static BasicNetwork createFeedforwardNetwork() {
  // construct a feedforward type network
    FeedForwardPattern pattern = new FeedForwardPattern();
    pattern.setActivationFunction(
      new ActivationTANH());
    pattern.setInputNeurons(1);
    pattern.addHiddenLayer(2);
    pattern.setOutputNeurons(1);
    return pattern.generate();
  }

  public static void main(final String args[]) {
    Logging.stopConsoleLogging();
    final TemporalXOR temp = new TemporalXOR();
    final NeuralDataSet trainingSet = temp.generate(100);

    final BasicNetwork elmanNetwork =
      ElmanXOR.createElmanNetwork();
    final BasicNetwork feedforwardNetwork =
      ElmanXOR.createFeedforwardNetwork();

    final double elmanError = ElmanXOR.trainNetwork(
      "Elman", elmanNetwork, trainingSet);

    final double feedforwardError =
      ElmanXOR.trainNetwork("Feedforward",
      feedforwardNetwork, trainingSet);

    System.out.println(
      "Best error rate with Elman Network: " + elmanError);
```

```
    System.out.println(
      "Best error rate with Feedforward Network: "
                          + feedforwardError);
    System.out.println("Elman should be able to get into the 30%
range,\nfeedforward should not go below 50%.\nThe recurrent
Elment net can learn better in this case.");
    System.out.println("If your results are not as good, try
rerunning, or perhaps training longer.");
  }

  public static double trainNetwork(
    final String what,
    final BasicNetwork network,
    final NeuralDataSet trainingSet) {

    // train the neural network
    CalculateScore score = new TrainingSetScore(trainingSet);
    final Train trainAlt = new NeuralSimulatedAnnealing(
      network, score, 10, 2, 100);

    final Train trainMain =
      new ResilientPropagation(network, trainingSet);

    final StopTrainingStrategy stop = new StopTrainingStrategy();
    trainMain.addStrategy(new Greedy());
    trainMain.addStrategy(new HybridStrategy(trainAlt));
    trainMain.addStrategy(stop);

    int epoch = 0;
    while (!stop.shouldStop()) {
      trainMain.iteration();
      System.out.println("Training " + what + ", Epoch #" + epoch
        + " Error:" + trainMain.getError());
      epoch++;
    }
  return trainMain.getError();
  }
}
```

When run, this program produces the following output:

```
Training Elman, Epoch #0 Error:1.4859292881311306
Training Elman, Epoch #1 Error:1.369308259241034
Training Elman, Epoch #2 Error:1.2018285171816208
Training Elman, Epoch #3 Error:0.9942688094109812
Training Elman, Epoch #4 Error:0.7301339755850338
```

```
Training Elman, Epoch #5 Error:0.5093578717358752
...
Training Elman, Epoch #128 Error:0.3259409438723773
Training Elman, Epoch #129 Error:0.3259409438723773
Training Elman, Epoch #130 Error:0.3259409438723773
Training Feedforward, Epoch #0 Error:0.6920831215854877
Training Feedforward, Epoch #1 Error:0.5539242161742655
Training Feedforward, Epoch #2 Error:0.5066387161431593
Training Feedforward, Epoch #3 Error:0.5038926941365289
Training Feedforward, Epoch #4 Error:0.5003584289169437
Training Feedforward, Epoch #5 Error:0.5003584289169437
...
Training Feedforward, Epoch #160 Error:0.49980139111813937
Training Feedforward, Epoch #161 Error:0.49980139111813937
Training Feedforward, Epoch #162 Error:0.49980139111813937
Best error rate with Elman Network: 0.3259409438723773
Best error rate with Feedforward Network: 0.49980139111813937
Elman should be able to get into the 30% range,
feedforward should not go below 50%.
The recurrent Elment net can learn better in this case.
If your results are not as good, try rerunning, or perhaps
training longer.
```

As you can see, the program attempts to train both a feedforward and an Elman neural network with the temporal XOR data. The feedforward neural network does not learn the data well. The Elman learns better. In this case feedforward neural network gets to 49.9% and Elman neural network gets to 32.5%. The context layer helps considerably.

This program uses random weights to initialize the neural network. If you run it and do not see as good of results, try rerunning. A better set of starting weights can help.

Creating an Elman Neural Network

Calling the **createElmanNetwork** method creates the Elman neural network in this example. This method is shown here.

```
static BasicNetwork createElmanNetwork() {
  // construct an Elman type network
  ElmanPattern pattern = new ElmanPattern();
  pattern.setActivationFunction(new ActivationTANH());
  pattern.setInputNeurons(1);
  pattern.addHiddenLayer(2);
```

```
  pattern.setOutputNeurons(1);
  return pattern.generate();
}
```

As you can see from the above code, the **ElmanPattern** is used to actually create the Elman neural network. To see the actual code used to produce an Elman neural network, without using a pattern, refer to Appendix C.

Training an Elman Neural Network

Elman neural networks tend to be particularly susceptible to local minima. A local minimum is a point where training stagnates. Visualize the weight matrix and thresholds as a landscape with mountains and valleys. To get to the lowest error, you want to find the lowest valley. Sometimes training finds a low valley and searches near this valley for a lower spot. It may fail to find an even lower valley several miles away.

This example's training uses several training strategies to help avoid this situation. The training code for this example is shown here. The same training routine is used for both the feedforward and Elman networks. The same RPROP training technique we used for feedforward networks is used. RPROP is not as efficient as it is for feedforward networks. However, adding a few training strategies helps greatly. The **trainNetwork** method is used to train the neural network. This method is shown here.

```
public static double trainNetwork(final String what,
  final BasicNetwork network, final NeuralDataSet trainingSet) {
```

One of the strategies employed by this program is a **HybridStrategy**. This allows an alternative training technique to be used if the main training technique stagnates. We will use simulated annealing as the alternative training strategy.

```
  CalculateScore score = new TrainingSetScore(trainingSet);
  final Train trainAlt = new NeuralSimulatedAnnealing(
    network, score, 10, 2, 100);
```

As you can see, we use a training set-based scoring object. For more information about simulated annealing, refer to Chapter 8, "Other Supervised Training Methods". The primary training technique is resilient propagation.

```
  final Train trainMain = new ResilientPropagation(
    network, trainingSet);
```

We will use a **StopTrainingStrategy** to tell us when to stop training. The **StopTrainingStrategy** will stop the training when the error rate stagnates. By default stagnation is defined as less than a 0.00001% improvement over 100 iterations.

```
  final StopTrainingStrategy stop = new StopTrainingStrategy();
```

These strategies are added to the main training technique.

```
  trainMain.addStrategy(new Greedy());
  trainMain.addStrategy(new HybridStrategy(trainAlt));
  trainMain.addStrategy(stop);
```

We also make use of a greedy strategy. This strategy will only allow iterations to improve the error rate of the neural network.

```
  int epoch = 0;
  while (!stop.shouldStop()) {
    trainMain.iteration();
  System.out.println("Training " + what + ", Epoch #" + epoch
    + " Error:" + trainMain.getError());
  epoch++;
  }
return trainMain.getError();
}
```

The loop continues until the stop strategy informs us that we should stop.

The Jordan Neural Network

Encog also contains a pattern for a Jordan neural network. The Jordan neural network is very similar to the Elman neural network. Figure 12.4 shows a Jordan neural network.

Figure 12.4: The Jordan Neural Network

As you can see, a context layer is used. However, the output from the output layer is fed back to the context layer, rather than the hidden layer. This small change in the architecture can make the Jordan neural network better for certain temporal prediction tasks.

Short of trial and error, it can be difficult to determine if an Elman or Jordan neural network will perform better. For example, the Jordan neural network presented here does not work nearly as well as the Elman neural network with the XOR example from earlier in this chapter. However, for certain market simulations that I've worked with, the Jordan network sometimes delivered better results than Elman. It really comes down to trial and error.

To construct a Jordan neural network, the **JordanPattern** should be used. The following code demonstrates this.

```
JordanPattern pattern = new JordanPattern();
pattern.setActivationFunction(new ActivationTANH());
pattern.setInputNeurons(1);
pattern.addHiddenLayer(2);
pattern.setOutputNeurons(1);
```

```
  return pattern.generate();
```

The above code would create a Jordan neural network similar to Figure 12.4.

Summary

In this chapter you learned about recurrent neural networks. A recurrent neural network is a neural network that contains connections backwards to previous layers. A recurrent neural network can also contain layers that are self-connected. In this chapter we looked at the Hopfield neural network, the Boltzmann Machine, and Elman/Jordan neural networks.

The Hopfield neural network is a self-connected neural network. Unlike feedforward neural networks the output is fed back into the input for the next cycle. This gives the Hopfield neural network a "state" that changes each cycle. Hopfield neural networks will stabilize on one of the patterns with which they were trained.

The Boltzmann machine is a simple recurrent neural network, similar to the Hopfield neural network. The weights of a Boltzmann machine are structured so that the Boltzmann machine will stabilize on an acceptable solution. In addition to the simple Boltzmann machine provided by Encog, restricted and stacked Boltzmann machines provide additional functionality.

Elman and Jordan neural networks make use of a context layer. This context layer allows them to learn patterns that span several items of training data. This makes them very useful for temporal neural networks.

Many of the neural network types that we have seen so far have hidden layers. It is often a process of trial and error to determine the structure of the hidden layers. Pruning can give some guidance to the structure of the hidden layers. In the next chapter we will look at how to prune a neural network.

Questions for Review

1. What neural network types, covered in this chapter, are self-connected?
2. How is temperature used in a Hopfield neural network?
3. How is temperature used in a Boltzmann machine?
4. What value does a context layer pass on to the next layer it is connected to?
5. You would like to use an Elman neural network to predict price movement in a stock. Why is it not necessary to choose a past window size, as would have been done with a feedforward neural network?
6. What must be true of the two layers that have a one-to-one synapse between them?
7. What is the difference between an Elman and a Jordan neural network?
8. Describe how an Encog hybrid training strategy works. What are the roles of the two training techniques employed by this strategy?
9. Which networks in this chapter had self-connected layers? In these network types are individual neurons, within the layer, connected to themselves?
10. Do Boltzmann machines make use of threshold values? Do Hopfield neural networks make use of threshold values?

Terms

Autoassociative

Boltzmann Machine

Context Layer

Elman Neural Network

Equilibrium

Greedy Training

Hopfield Neural Network

Hybrid Training

Jordan Neural Network

One to One Synapse

Self-Connected Layer

Simple Recurrent Neural Network (SRN)

Thermal Neural Network

Traveling Salesman Problem (TSP)

Chapter 13: Structuring Hidden Layers

- Determining Hidden Layer Structure
- Selective Pruning
- Incremental Pruning

We have seen many neural networks in this book with hidden layers. Not much though has gone into how to structure these hidden layers. This chapter will provide some insight on dealing with the hidden layers of these neural networks. Not all types of neural network have hidden layers. Of the neural networks that we have examined so far, the following neural network types have hidden layers:

- Feedforward Neural Network
- Elman Neural Network
- Jordan Neural Network

Choosing the correct hidden layer structure can have a great deal of impact on the performance of the neural network. Encog provides some automatic capabilities for determining how many hidden layers are necessary. More advanced features will be added to Encog in later versions to further extend this capability.

Understanding Hidden Layer Structure

Most neural networks will have one or two hidden layers. In some cases more than two can be useful. There really are not many rules for determining which structure is going to be optimal. Generally, I have found that there should be more neurons in the first hidden layer than there are in the input layer. I often begin with two times the number of input neurons and work my way down from there.

Deciding if you need a second hidden layer is much more abstract. I have found that a second hidden layer can sometimes be helpful. Later in this chapter you will see how Encog can use a trial-and-error method to help find a neural network architecture that might be effective. You can instruct Encog to try different combinations of hidden layers in an attempt to find a good structure.

The process of adjusting the number of hidden neurons is called pruning. Pruning in Encog typically takes one of two forms.

- Selective Pruning
- Incremental Pruning

Selective pruning is the simpler of the two. By using selective pruning you instruct Encog to either increase or decrease the number of neurons on a level. Incremental pruning starts with a small neural network and tries increasingly larger hidden layer structures in an attempt to find the one that trains the best. Both types of pruning will be covered in this chapter. We will begin with selective pruning.

Using Selective Pruning

Selective pruning is the simpler of the two pruning types offered by Encog. Selective pruning is done using the **PruneSelective** class and is used to either increase or decrease the neuron count for a layer. The Encog Workbench makes extensive use of the **PruneSelective** class to allow the user to change the number of neurons on a layer.

There are four public methods that the **PruneSelective** class makes available.

- **changeNeuronCount**
- **determineNeuronSignificance**
- **prune**
- **magnifyWeakNeurons**

We will see how to use each of them. First, let's examine how to expose the different layers of a neural network. Consider a scenario where you might have created a feedforward neural network with the following pattern:

```
FeedForwardPattern pattern = new FeedForwardPattern();
pattern.setActivationFunction(new ActivationTANH());
pattern.setInputNeurons(1);
pattern.addHiddenLayer(8);
pattern.addHiddenLayer(4);
pattern.setOutputNeurons(1);
```

```
BasicNetwork network = pattern.generate();
```

This would have created a feedforward neural network with two hidden layers, as well as an input and output layer. To prune individual layers, you need to gain access to the individual layers. The following lines of code show you how to access these individual layers.

```
Layer input = network.getLayer(BasicNetwork.TAG_INPUT);
Layer output = network.getLayer(BasicNetwork.TAG_OUTPUT);
Layer hidden1 = input.getNext().get(0).getToLayer();
Layer hidden2 = hidden1.getNext().get(0).getToLayer();
```

Now that you have access to the individual layers, you can selectively prune them. Begin by creating a **PruneSelective** object, as follows.

```
PruneSelective prune = new PruneSelective(network);
```

If you would like to change the neuron count for the first hidden layer, you would use the following code.

```
prune.changeNeuronCount(hidden1, 6);
```

This would change the **hidden1** layer to six neurons. In this case, it would decrease the neuron count, as there were eight neurons before. The **changeNeuronCount** method can either increase or decrease the neuron count. Both of these operations are handled differently, and will be discussed in the next sections.

The **changeNeuronCount** method is usually used only on hidden layers. However, it can be used on the input and output layer as well. However, when you change the input or output layer of a neural network you are changing the fundamental definition of how the neural network is fitted to a problem. You will also have to change your training data and retrain the neural network.

Increasing the Neuron Count

The **changeNeuronCount** method automatically determines if you are increasing, decreasing or not changing the neuron count. If you specify the same number of neurons already present on the layer, then no change occurs. If you increase or decrease the neuron count, the idea is to limit changing the way that the neural network processes data as much as possible.

When the neuron count is increased, new neurons are added to the layer. These neurons will start with a threshold value of zero. There will also be connections to and from the previous and next layers. These weights will be assigned to zero. The new neurons will not affect the neural network. You may wish to call the **magnifyWeakNeurons** method to slightly randomize the weakest neurons on a layer. The weakest neurons would be the neurons that were just added. For example, the following code would magnify the two weakest neurons so that the weights and threshold values were at least ten percent of the other neurons.

```
prune.magnifyWeakNeurons(hidden1, 2, 0.1);
```

This will slightly change the operation of the neural network, but it also increases the likelihood that these new neurons will benefit from further training.

Decreasing the Neuron Count

You can also decrease a layer's neuron count. Decreasing the neuron count of a hidden layer will most likely affect the output of the neural network. Unless the weights and thresholds of the neuron to be removed are zero, this neuron was contributing somehow to the output of the neural network. Removing it will affect the neural network output.

There are two ways to remove a neuron. The first is to actually specify the neuron that you would like to remove. This is done using the **prune** method.

```
prune.prune(hidden1, 0);
```

The above code removes the first neuron from the **hidden1** layer. Unfortunately, you do not know the significance of the first neuron in this layer. It might have been the most important neuron in the layer. To determine a neuron's significance, the **determineNeuronSignificance** method can be used. The following code would display the significance of the first neuron.

```
System.out.println( prune.determineNeuronSignificance (
  hidden1, 0) )
```

The above code checks the significance of the first neuron. The number returned takes into account the connections this neuron has, as well as its

threshold. You should only use this number as a comparison to another significance number. If the significance number is higher for one layer than another, that layer is more significant. A more significant neuron has larger weights and thresholds.

The preferred method is usually just to use the **changeNeuronCount** method, which was discussed earlier, and simply specify how many neurons you would like to have. If this number decreases the neuron count, then the least significant neurons will be removed to change to the new neuron count.

You can also prune an entire layer. To prune a layer, set its neuron count to zero. The following line removes the **hidden1** layer.

```
prune.changeNeuronCount(hidden1, 0);
```

Pruning an entire layer is a major change to the neural network. All connections between this layer and the rest of the network are dropped. New connections are made between the remaining layers to fill in the gap left by the layer. At this point the network should be randomized and retrained. It is unlikely that any of the remaining weights will be of much use in producing meaningful output from the neural network.

Using Incremental Pruning

Selective pruning is usually used after you have trained the neural network and wish to remove ineffective neurons. Incremental pruning works from the opposite direction. Incremental pruning starts by creating new neural networks according to some specifications provided up front. The pruning algorithm then exhaustively tries every possible hidden layer configuration. Of course, it would take forever to try every combination, that's why certain specifications are given up front. Of course, this is still process intensive, and can take hours or days to complete.

Configuring Incremental Pruning

The **PruneIncremental** class is used to perform incremental pruning. To use incremental pruning you must first decide how many hidden layers you are willing to have, and how many hidden neurons you would like on each layer. We will look at the code necessary to make use of incremental pruning. The first step is to somehow obtain a training set.

```
NeuralDataSet training =
(NeuralDataSet)encog.find( ...load training set... );
```

The above code loads a training set from an Encog EG file; however, it could be obtained by any valid method used for obtaining a training set in Encog. We also create a pattern. Here we create a pattern for a feedforward neural network. No hidden layers should be specified, as the incremental training algorithm will cycle through these.

```
FeedForwardPattern pattern = new FeedForwardPattern();
pattern.setInputNeurons(training.getInputSize());
pattern.setOutputNeurons(training.getIdealSize());
pattern.setActivationFunction(new ActivationTANH());
```

Next, we actually create the **PruneIncremental** object.

```
PruneIncremental prune = new PruneIncremental(
  training,
  pattern,
  100,
  new ConsoleStatusReportable());
```

The 100 parameter specifies that we will train each hidden layer combination for up 100 iterations. The **ConsoleStatusReportable** object specifies that all status reports should be sent to the console while the incremental pruning algorithm runs.

Next, we must specify how many hidden layers we wish and the valid range for neurons on each level. The following two lines do this:

```
prune.addHiddenLayer(5, 50);
prune.addHiddenLayer(0, 50);
```

The above lines specify that this neural network can have up to two hidden layers. The first hidden layer can have a minimum of five neurons and a maximum of 50 neurons. Because the minimum is not zero, the first hidden layer is required. The second hidden layer can have a minimum of zero neurons, or a maximum of 50 neurons. Because the minimum number of neurons is zero, the second hidden layer is not required.

We can now begin processing. Calling the **process** method will begin cycling through all of the possible combinations of hidden layers and neurons.

The progress will be reported to the report object specified when the **PruneIncremental** object was constructed.

```
prune.process();
```

Once the processing is done, the **process** method returns.

```
BasicNetwork network = prune.getBestNetwork();
```

Once the processing is done, the best network can be obtained by calling the **getBestNetwork** method.

Incremental Pruning Example

In Chapter 10, "Using Temporal Data" we saw a program that attempted to predict the price of a particular stock. There were several command-line arguments that could be used to cause the neural network to be generated, trained or evaluated. One additional parameter that is available is **prune**. To use incremental pruning with this example, specify the **prune** argument. You may wish to review the Chapter 10 example before proceeding with this example. When this argument is specified, the **MarketPrune** class is used. This class is shown in Listing 13.1.

Listing 13.1: Pruning the Stock Market Neural Network

```
package org.encog.examples.neural.predict.market;

import java.io.File;

import org.encog.ConsoleStatusReportable;
import org.encog.neural.activation.ActivationTANH;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.pattern.FeedForwardPattern;
import org.encog.neural.prune.PruneIncremental;
import org.encog.persist.EncogPersistedCollection;

public class MarketPrune {

  public static void incremental() {
  File file = new File(Config.FILENAME);

  if (!file.exists()) {
    System.out.println("Can't read file: "
      + file.getAbsolutePath());
```

```
    return;
  }

  EncogPersistedCollection encog =
    new EncogPersistedCollection(file);

    NeuralDataSet training = (NeuralDataSet) encog
    .find(Config.MARKET_TRAIN);

    FeedForwardPattern pattern = new FeedForwardPattern();
    pattern.setInputNeurons(training.getInputSize());
    pattern.setOutputNeurons(training.getIdealSize());
    pattern.setActivationFunction(new ActivationTANH());

    PruneIncremental prune =
      new PruneIncremental(training, pattern, 100,
      new ConsoleStatusReportable());

    prune.addHiddenLayer(5, 50);
    prune.addHiddenLayer(0, 50);

    prune.process();

    encog.add(Config.MARKET_NETWORK, prune.getBestNetwork());

  }
}
```

Pruning is accomplished by using the **incremental** method. This method is shown here.

```
public static void incremental() {
```

We begin by reading in the same EG file from which this example stores the neural network and training data.

```
File file = new File(Config.FILENAME);

if( !file.exists() ) {
  System.out.println(
    "Can't read file: " + file.getAbsolutePath() );
  return;
}
```

We use an **EncogPersistedCollection** to read the file.

```
EncogPersistedCollection encog =
  new EncogPersistedCollection(file);
```

Incremental training needs a training set. The training set is loaded from the EG file. For more information on how this data was created, refer to Chapter 10.

```
NeuralDataSet training =
  (NeuralDataSet)encog.find(Config.MARKET_TRAIN);
```

A feedforward pattern will be used. We will use the same input and output sizes as specified by the training data.

```
FeedForwardPattern pattern = new FeedForwardPattern();
pattern.setInputNeurons(training.getInputSize());
pattern.setOutputNeurons(training.getIdealSize());
pattern.setActivationFunction(new ActivationTANH());
```

We can now create a **prune** object. We will only use 100 iterations, which will cause this example to run reasonably fast. However, for better results, more iterations should be used.

```
PruneIncremental prune = new PruneIncremental(
  training, pattern, 100, new ConsoleStatusReportable());
```

Allow up to two hidden layers. The first hidden layer is between 5 and 50. The second is between 0 and 50.

```
prune.addHiddenLayer(5, 50);
prune.addHiddenLayer(0, 50);
```

We now begin processing.

```
prune.process();

encog.add(Config.MARKET_NETWORK, prune.getBestNetwork());
}
```

Once processing is done, the best neural network is saved back to the EG file. The output from the pruning process is shown here.

```
1/2346 : Current: H1=5; Best: H1=5
2/2346 : Current: H1=6; Best: H1=5
3/2346 : Current: H1=7; Best: H1=5
4/2346 : Current: H1=8; Best: H1=8
5/2346 : Current: H1=9; Best: H1=8
```

```
6/2346 : Current: H1=10; Best: H1=8
7/2346 : Current: H1=11; Best: H1=8
8/2346 : Current: H1=12; Best: H1=8
…
2338/2346 : Current: H1=42,H2=50; Best: H1=47,H2=6
2339/2346 : Current: H1=43,H2=50; Best: H1=47,H2=6
2340/2346 : Current: H1=44,H2=50; Best: H1=47,H2=6
2341/2346 : Current: H1=45,H2=50; Best: H1=47,H2=6
2342/2346 : Current: H1=46,H2=50; Best: H1=47,H2=6
2343/2346 : Current: H1=47,H2=50; Best: H1=47,H2=6
2344/2346 : Current: H1=48,H2=50; Best: H1=47,H2=6
2345/2346 : Current: H1=49,H2=50; Best: H1=47,H2=6
2346/2346 : Current: H1=50,H2=50; Best: H1=47,H2=6
Best network: H1=47,H2=6
```

As you can see, the best neural network structure found was 47 hidden layer 1 neurons and 6 hidden layer 2 neurons. A larger first layer and smaller second hidden layer are quite common for a potentially optimum hidden layer architecture. The program saved this network to the EG file, and the network is now ready for training.

Summary

This chapter introduced pruning. Pruning is used to refer to any process where the hidden layer structure is automatically determined. This chapter showed two methods for pruning. Incremental starts with an empty neural network and works upwards, selective pruning starts with a trained neural network and works downwards.

Selective pruning removes neurons from a previously trained neural network. Encog provides options for the programmer to either specify the exact neuron to remove, or to allow Encog to pick the neuron to remove. When Encog picks the neuron to remove, the least significant neuron is removed. This can often produce a more efficient neural network.

Incremental pruning starts with an empty, untrained neural network. Hidden neurons are added in a variety of configurations. At each step, the neural network is trained, and the final error is recorded. Once all combinations have been tried, the best-trained neural network is returned. This neural network is then returned.

This book has focused primarily on feedforward neural networks. Additional neural network types, such as Self-Organizing Maps, Hopfield Neural Networks, Boltzmann Machines and Elman/Jordan neural networks were also covered. Encog also contains patterns that create a number of other, less commonly used neural network types. The book will conclude with Chapter 14 where some of these other neural network architectures will be explored.

Terms

Incremental Pruning

Pruning

Selective Pruning

Questions for Review

1. Would one of the pruning techniques described in this chapter work for making the hidden layers of a SOM more efficient? Why or why not?

2. When Encog must choose to remove a neuron from a hidden layer, which neuron is chosen?

3. You have a feedforward neural network that has already been trained. Which pruning technique should you use for this situation?

4. How is the least significant neuron determined?

5. What method is called to allow Encog to select a neuron to remove? What method is called when you want to remove a specific neuron?

Chapter 14: Other Network Patterns

- Radial Basis Function Networks (RBF)
- Adaptive Resonance Theory (ART1)
- Counter-Propagation Neural Networks (CPN)

In this chapter we will examine some of the other neural network types supported by Encog that were not covered in earlier parts of the book. Most of the interest in Encog, at least gauged by forum questions, seems to be in the area of feedforward, recurrent and self-organizing maps. These neural network types were the focus of the first part of the book. However, they are not the only network types supported by Encog. This chapter will look at some of the less frequently used network types supported by Encog. Other examples and articles, on these network types, will be added as they are available.

We will look at three such neural network types in this chapter. Each of these neural network types could easily fill one or more chapters. We will only present the highlights of each type here. If there is a particular neural network type that you would like to see added to Encog, the Encog forum is the best place to make this known. The Encog forum can be found at this URL:

http://www.heatonresearch.com/forum/

If you would like to see additional coverage for one particular type of neural network, the forum is also useful. Forum posts and questions play a considerable role in the selection of future features of Encog. The forum is also useful for notifying us of any bugs you discover in Encog.

This chapter will look at three additional neural network types. The Radial Basis Function network works similar to a regular feedforward network, except its hidden layer is partially governed by a radial basis function. The Adaptive Resonance Theory (ART1) can be taught to recognize a number of bipolar input patterns and exhibits plasticity. The Counter-Propagation neural network is a hybrid neural network that is trained in both a supervised and unsupervised fashion. We will begin with Radial Basis Function networks.

Radial Basis Function Networks

Radial Basis Function networks (RBF) are a special type of feedforward network. They make use of a radial basis function. We saw radial basis functions in Chapter 9, "Unsupervised Training Methods". In this chapter, we will see how a special layer, based on a radial basis function, can be used to create a radial basis function neural network.

A RBF network contains a compound activation function that is based on several radial basis functions, usually Gaussian functions. This makes a radial basis function very useful for function approximation and predictive neural networks. In this section, we will see how to construct and use a RBF network in Encog.

Constructing a RBF Neural Network

A RBF network has a very specific structure as shown in Figure 14.1 below.

Figure 14.1: A RBF Network in Encog Workbench

As you can see, the RBF network has three layers. The input and output layers are both linear. The middle layer is a special RBF-based layer, provided by Encog. This layer class type is known as the **RadialBasisFunctionLayer**.

Encog provides a special pattern to create this sort of neural network. The **RadialBasisPattern** can be used to create a RBF. The following code shows how to use the radial basis function pattern.

```
RadialBasisPattern pattern = new RadialBasisPattern();
pattern.setInputNeurons(   [Input Neurons]   );
pattern.addHiddenLayer(    [Hidden Neurons]  );
pattern.setOutputNeurons(  [Output Neurons]  );
BasicNetwork network = pattern.generate();
```

This is very similar to the types of feedforward neural networks we have seen so far. The input and output layers are used just as they are in any feedforward network. It is the hidden layer that is handled differently. We will explore this in the next section.

How the RBF is Used

The hidden layer in a RBF network makes use of one or more Gaussian functions. However, this is not the first time we've seen the Gaussian function. Equation 14.1 shows the Gaussian function.

Equation 14.1: The Gaussian Equation

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

In this equation, the constant **a** represents the peak of the curve, **b** is the position of the curve, and **c** is the width of the curve. If we set the peak and width to one, and the position to zero, we are left with Equation 14.2, which is a very simple Gaussian function.

Equation 14.2: Simple Gaussian Equation

$$f(x) = e^{-\frac{x^2}{4}}$$

Equation 14.2 can be graphed. The graph of this equation is shown in Figure 14.2.

Figure 14.2: Graph of the Simple Gaussian Function

As you can see, this creates a bell curve. This curve was used in Chapter 9, "Unsupervised Training Methods" to define a neighborhood of neurons. RBF neural networks make use of several Gaussian functions, added together. Adding Gaussian functions together can create more complex shapes. This creates a compound Gaussian function Equation 14.3 shows two Gaussian functions added together to form a compound equation.

Equation 14.3: A Compound Gaussian Function

$$f(x) = \left(1e^{-\frac{(x+2)^2}{(2-1)^2}}\right) + \left(2e^{-\frac{(x-2)^2}{(2-1)^2}}\right)$$

Before we look at the graph of what this compound Gaussian function looks like we should examine the graphics of the two clauses that make up the compound function. Figure 14.3 shows the first clause in the compound Gaussian function.

Figure 14.3: Graph of the First Gaussian Clause

Notice that it is a typical Gaussian curve, with a shifted center. Figure 14.4 shows a graph of the second clause.

Figure 14.4: Graph of the Second Gaussian Clause

This Gaussian equation has a higher peak and is shifted as well. When the two clauses are added together, we see Figure 14.5, which is the graph of Equation 14.3.

Figure 14.5: Graph of Compound Gaussian Equation

Notice how the shape of the curve takes on characteristics from both of the clauses of the Gaussian compound function. This is how Gaussian functions can be used to approximate other functions. By stringing many of these clauses together, complex graphics can be created.

The hidden layer of a RBF is made up of a number of RBF functions that are added together. The number of hidden neurons is the number of RBF functions that are used. This complex activation function allows the RBF neural network to recognize certain patterns that a regular feedforward neural network might not be able to. In the next section we will look at a simple RBF neural network.

A Simple RBF Network

The RBF example that we will see in this section simply learns the XOR pattern. There are many other things that the RBF network is capable of doing; however, a simple XOR network will demonstrate how to use this sort of network in Encog. This example can be seen in Listing 14.1.

Listing 14.1: An XOR Radial Basis Function

```
package org.encog.examples.neural.xorradial;

import org.encog.neural.data.NeuralDataSet;
```

```
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.RadialBasisFunctionLayer;
import org.encog.neural.pattern.RadialBasisPattern;
import org.encog.util.logging.Logging;
import org.encog.util.simple.EncogUtility;

public class XorRadial {

  public static double XOR_INPUT[][] = {
    { 0.0, 0.0 },
    { 1.0, 0.0 },
    { 0.0, 1.0 },
    { 1.0, 1.0 } };

  public static double XOR_IDEAL[][] = {
    { 0.0 },
    { 1.0 },
    { 1.0 },
    { 0.0 } };

  public static void main(final String args[]) {

    Logging.stopConsoleLogging();

    RadialBasisPattern pattern = new RadialBasisPattern();

    pattern.setInputNeurons(2);
    pattern.addHiddenLayer(4);
    pattern.setOutputNeurons(1);

    BasicNetwork network = pattern.generate();
    RadialBasisFunctionLayer rbfLayer =
      (RadialBasisFunctionLayer)network.getLayer(
      RadialBasisPattern.RBF_LAYER);

      rbfLayer.randomizeGaussianCentersAndWidths(0, 1);

      final NeuralDataSet trainingSet = new BasicNeuralDataSet(
        XorRadial.XOR_INPUT, XorRadial.XOR_IDEAL);

      // train the neural network
      EncogUtility.trainToError(network, trainingSet, 0.01);

      // test the neural network
      System.out.println("Neural Network Results:");
```

```
      EncogUtility.evaluate(network, trainingSet);
    }
}
```

This program is very similar to any of the other XOR examples we have looked at, except it starts with a RBF pattern. The following lines of code create an RBF network for the XOR operator.

```
RadialBasisPattern pattern = new RadialBasisPattern();
pattern.setInputNeurons(2);
pattern.addHiddenLayer(4);
pattern.setOutputNeurons(1);
BasicNetwork network = pattern.generate();
```

The above code creates the typical feedforward network for an XOR operator with two input neurons and a single output neuron. Additionally, a hidden layer of four RBF neurons is used.

For an RBF network, you will need to specify the position, peak and widths for each of the hidden layer RBFs. To do this, you will need access to the hidden RBF layer. The following code obtains this layer.

```
RadialBasisFunctionLayer rbfLayer =
  (RadialBasisFunctionLayer)network.getLayer(
    RadialBasisPattern.RBF_LAYER);
```

There are two ways that Encog can set these parameters. You can either specify them yourself, or set them to random values. If you wanted to set them yourself, you would use the following code.

```
rbfLayer.setRadialBasisFunction(0, new
GaussianFunction(0.0,1,0.5));

rbfLayer.setRadialBasisFunction(1, new
GaussianFunction(0.25,1,0.5));

rbfLayer.setRadialBasisFunction(2, new
GaussianFunction(0.5,1,0.5));

rbfLayer.setRadialBasisFunction(3,
  new GaussianFunction(1.0,1,0.5));
```

The above code defines each of the four RBF functions with the specified values. The first parameter is the position of the RBF function. The second parameter is the peak, and the third is the width. This allows you to define the RBFs. More advanced RBF algorithms can actually automatically optimize the positions of the RBFs. Encog's support of RBF neural networks does not yet support this, but this feature will likely be added in the future.

To set the RBFs to random values, the following code is used. This is how the example is actually implemented.

```
rbfLayer.randomizeGaussianCentersAndWidths(0, 1);
```

The above line randomizes the widths, centers and peaks between the min and max values specified above.

Next, we create a training set for the XOR operator.

```
final NeuralDataSet trainingSet = new BasicNeuralDataSet(
  XorRadial.XOR_INPUT, XorRadial.XOR_IDEAL);
```

The **EncogUtility** class is used to train the neural network. The **trainToError** method trains the neural network until the specified error is reached. This will train the network until the error is below 1%.

```
EncogUtility.trainToError(network, trainingSet, 0.01);
```

Once the network has been trained it is evaluated. The **evaluate** method is used to do this. This method displays every training element with the neural network's output. This lets you see how well the network has been trained to act as an XOR operator.

```
System.out.println("Neural Network Results:");
EncogUtility.evaluate(network, trainingSet);
```

Entire chapters could be written about RBF neural networks. This example shows the basics of how to use a RBF neural network.

Adaptive Resonance Theory

Adaptive Resonance Theory (ART) is a form of neural network developed by Stephen Grossberg and Gail Carpenter. There are several versions of the ART neural network, which are numbered ART-1, ART-2 and ART-3. The ART

neural network is trained using either a supervised or unsupervised learning algorithm, depending on the version of ART being used. ART neural networks are used for pattern recognition and prediction.

This section will focus on ART1 using unsupervised training. The ART1 example we will look at will function similar to the self-organizing map (SOM) seen in Chapter 9. The ART neural network will accept bipolar or binary patterns and then group them.

The ART network is a simple two-layer neural network. This network is shown in Figure 14.6.

Figure 14.6: The Adaptive Resonance Theory 1 (ART1) Network

As you can see, there is an input and an output layer. A pattern is presented to the input layer, and the winning output neuron determines the group to which the pattern has been assigned. This example can be seen in Listing 14.2.

Listing 14.2: Adaptive Resonance Theory

```
package org.encog.examples.neural.art.art1;

import org.encog.neural.data.bipolar.BiPolarNeuralData;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.logic.ART1Logic;
import org.encog.neural.pattern.ART1Pattern;

public class NeuralART1 {

  public static final int INPUT_NEURONS = 5;
  public static final int OUTPUT_NEURONS = 10;
```

```
public static final String[] PATTERN = {
  "   O ",
  "  O O",
  "    O",
  "  O O",
  "    O",
  "  O O",
  "    O",
  " OO O",
  " OO  ",
  " OO O",
  " OO  ",
  "OOO  ",
  "OO   ",
  "O    ",
  "OO   ",
  "OOO  ",
  "OOOO ",
  "OOOOO",
  "O    ",
  " O   ",
  "  O  ",
  "   O ",
  "    O",
  "  O O",
  " OO O",
  " OO  ",
  "OOO  ",
  "OO   ",
  "OOOO ",
  "OOOOO"  };

private boolean[][] input;

public void setupInput() {
  this.input = new boolean[PATTERN.length][INPUT_NEURONS];

  for (int n = 0; n < PATTERN.length; n++) {
    for (int i = 0; i < INPUT_NEURONS; i++) {
      this.input[n][i] = (PATTERN[n].charAt(i) == 'O');
    }
  }
}

public void run() {
```

```
    this.setupInput();
    ART1Pattern pattern = new ART1Pattern();
    pattern.setInputNeurons(INPUT_NEURONS);
    pattern.setOutputNeurons(OUTPUT_NEURONS);
    BasicNetwork network = pattern.generate();
    ART1Logic logic = (ART1Logic) network.getLogic();

    for (int i = 0; i < PATTERN.length; i++) {
      BiPolarNeuralData in =
        new BiPolarNeuralData(this.input[i]);
      BiPolarNeuralData out =
        new BiPolarNeuralData(OUTPUT_NEURONS);

      logic.compute(in, out);
      if (logic.hasWinner()) {
        System.out.println(PATTERN[i] + " - "
          + logic.getWinner());
      } else {
        System.out.println(PATTERN[i]
          + " - new Input and all Classes exhausted");
      }
    }
  }

  public static void main(String[] args) {
    NeuralART1 art = new NeuralART1();
    art.run();
  }
}
```

The program is concentrated primarily in the **run** method. The **run** method is shown here.

```
public void run(){
```

The **run** method begins by setting up the input. This is just a simple routine that loops over the input strings and converts them to bipolar numbers. Bipolar numbers were introduced in Chapter 12, "Recurrent Neural Networks". Basically a bipolar number encodes **true** as 1 and **false** as -1.

```
this.setupInput();
```

Now that the input patterns have been loaded, the ART network is created. The **ART1Pattern** is provided to create an ART1 network.

```
ART1Pattern pattern = new ART1Pattern();
pattern.setInputNeurons(INPUT_NEURONS);
pattern.setOutputNeurons(OUTPUT_NEURONS);
BasicNetwork network = pattern.generate();
```

Once the network has been created, we access the **ART1Logic** class. We will deal directly with this class while using the ART1 network. The pattern hides the complexity of creating the network. If you would like to see how the network was actually constructed, refer to Appendix C.

```
ART1Logic logic = (ART1Logic)network.getLogic();
```

We will loop over all of the sample patterns and present them to the network. As they are presented to the network, it will learn to group them. To do this we begin by looping over every provided pattern.

```
for(int i=0;i<PATTERN.length;i++) {
```

We obtain holders for both the input and output from the neural network. The input initializes from the pattern array.

```
BiPolarNeuralData in = new BiPolarNeuralData(this.input[i]);
BiPolarNeuralData out = new BiPolarNeuralData(OUTPUT_NEURONS);
```

Calling the **compute** method of the logic class determines the group the pattern is placed.

```
logic.compute(in, out);
```

If there was a winning neuron, then the pattern was successfully grouped.

```
if( logic.hasWinner() ) {
  System.out.println(PATTERN[i] + " - " + logic.getWinner() );
} else {
  System.out.println(PATTERN[i] +
    " - new Input and all Classes exhausted" );
}
```

If there was no winner, the input pattern was not close enough to an existing group, and there are no additional groups to form. Creating more output neurons would increase the group count and would help this situation.

Here you can see the output from this program.

```
   O  - 0
  O O - 1
    O - 1
  O O - 2
    O - 1
  O O - 2
    O - 1
 OO O - 3
 OO   - 3
 OO O - 4
 OO   - 3
OOO   - 5
OO    - 5
O     - 5
OO    - 6
OOO   - 7
OOOO  - 8
OOOOO - 9
O     - 5
 O    - 3
  O   - 2
   O  - 0
    O - 1
  O O - 4
 OO O - 9
 OO   - 7
OOO   - 8
OO    - 6
OOOO  - new Input and all Classes exhausted
OOOOO - new Input and all Classes exhausted
```

The above illustrates the new patterns that are being added to groups. The pattern is shown on the left and the group number on the right. Some patterns are repeated, others are close enough that a group number is repeated. Finally, once all of the groups are used, no new patterns can be learned.

This is very similar to a self-organizing map, in that patterns are learned with no actual supervision given. The network is told what the patterns are, but are left to group them itself.

However, there are some important differences between a SOM and an ART network. The ART network has no distinct training phase. It exhibits

plasticity. It is shaped and modeled as it is used. The network began learning as soon as the first pattern was provided and grouped. As additional patterns are provided, it continues to learn.

This example provided an introduction to the ART1 network. Encog currently supports only the ART1 structure of adaptive resonance theory. Future versions of Encog may expand this. More advanced variants of ART can take inputs other than bipolar and use more complex learning algorithms.

Counter-Propagation Neural Networks

So far, this book has classified neural network training into one of two camps. Either the training is supervised or unsupervised. Not all network architectures are either supervised or unsupervised. Sometimes a neural network can be both supervised and unsupervised. Such is the case concerning the counter-propagation neural network (CPN).

Figure 14.7: The Counter-Propagation Neural Network

The network above looks very similar to some of the feedforward neural networks we have seen before. However, you will notice there are some additional labels. There is an input layer just like other neural network types. However, the hidden layer is called an instar layer and the output is called an outstar layer.

However, there really is no hidden layer. It is best to think of this network as the combination of two neural networks. The first network extends from the input layer to the instar layer. This is a competitive network that is trained in an unsupervised fashion, similar to a SOM. There is also a 2-layer feedforward neural network that extends from the instar to the outstar layer. This second neural network is trained in a method similar to backpropagation.

We will look at an example that takes images of small rockets and attempts to determine the angle at which the rocket is flying.

Listing 14.3: Counter Propagation Neural Network

```
package org.encog.examples.neural.cpn;

import org.encog.neural.data.NeuralData;
import org.encog.neural.data.NeuralDataSet;
import org.encog.neural.data.basic.BasicNeuralData;
import org.encog.neural.data.basic.BasicNeuralDataSet;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.training.Train;
import org.encog.neural.networks.training.cpn.TrainInstar;
import org.encog.neural.networks.training.cpn.TrainOutstar;
import org.encog.neural.pattern.CPNPattern;

public class RocketCPN {

  public static final int WIDTH = 11;
  public static final int HEIGHT = 11;

  public static final String[][] PATTERN1 =  { {
    "           ",
    "           ",
    "     O     ",
    "     O     ",
    "    OOO    ",
    "    OOO    ",
    "    OOO    ",
    "   OOOOO   ",
```

```
    "   OOOOO   ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "         O ",
    "        O  ",
    "      OOO  ",
    "     OOO   ",
    "    OOO    ",
    " OOOOO     ",
    "OOOOO      ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "           ",
    "  OO       ",
    "  OOOOO    ",
    "  OOOOOOO  ",
    "  OOOOO    ",
    "  OO       ",
    "           ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "OOOOO      ",
    " OOOOO     ",
    "    OOO    ",
    "     OOO   ",
    "      OOO  ",
    "        O  ",
    "         O ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "   OOOOO   ",
    "   OOOOO   ",
    "    OOO    ",
    "    OOO    ",
```

```
    "    OOO    ",
    "     O     ",
    "     O     ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "      OOOOO",
    "     OOOOO ",
    "     OOO   ",
    "    OOO    ",
    "   OOO     ",
    "   O       ",
    "  O        ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "           ",
    "       OO  ",
    "    OOOOO  ",
    "  OOOOOOO  ",
    "    OOOOO  ",
    "       OO  ",
    "           ",
    "           ",
    "           "  },

  { "           ",
    "           ",
    "  O        ",
    "   O       ",
    "   OOO     ",
    "    OOO    ",
    "     OOO   ",
    "     OOOOO ",
    "      OOOOO",
    "           ",
    "           "  } };

    String[][] PATTERN2 = { {
      "           ",
      "           ",
      "     O     ",
```

```
      "      O      ",
      "      O      ",
      "     OOO     ",
      "     OOO     ",
      "     OOO     ",
      "    OOOOO    ",
      "             ",
      "             "   },

    { "             ",
      "             ",
      "      O      ",
      "      O      ",
      "     O O     ",
      "     O O     ",
      "     O O     ",
      "    O   O    ",
      "    O   O    ",
      "             ",
      "             "   },

    { "             ",
      "             ",
      "             ",
      "      O      ",
      "     OOO     ",
      "     OOO     ",
      "     OOO     ",
      "    OOOOO    ",
      "             ",
      "             ",
      "             "   },

    { "             ",
      "             ",
      "             ",
      "             ",
      "      O      ",
      "      O      ",
      "      O      ",
      "     OOO     ",
      "             ",
      "             ",
      "             "   },

    { "             ",
```

```
      "  O         ",
      "     O      ",
      "     O      ",
      "    OOO     ",
      "    OO      ",
      "    OOO   O",
      "    OOOO    ",
      "   OOOOO    ",
      "            ",
      "        O   "  },

    { "            ",
      "            ",
      "     O      ",
      "     O      ",
      "    OOO     ",
      "    OOO     ",
      "    OOO     ",
      "   OOOOO    ",
      "   OOOOO    ",
      "            ",
      "            "  },

    { "            ",
      "            ",
      "        O   ",
      "       O    ",
      "    OOO     ",
      "    OOO     ",
      "   OOO      ",
      "  OOOOO     ",
      " OOOOO      ",
      "            ",
      "            "  },

    { "            ",
      "            ",
      "         O  ",
      "        O   ",
      "      OOO   ",
      "    OOO     ",
      "   OOO      ",
      " OOOOO      ",
      "OOOOO       ",
      "            ",
      "            "  } };
```

```
  public static final double HI = 1;
  public static final double LO = 0;


  private double[][] input1;
  private double[][] input2;
  private double[][] ideal1;

  private int inputNeurons;
  private int instarNeurons;
  private int outstarNeurons;

  public void prepareInput() {
    int n,i,j;

    this.inputNeurons = WIDTH * HEIGHT;
    this.instarNeurons = PATTERN1.length;
    this.outstarNeurons = 2;

    this.input1 =
      new double[PATTERN1.length][this.inputNeurons];
    this.input2 =
      new double[PATTERN2.length][this.inputNeurons];
    this.ideal1 =
      new double[PATTERN1.length][this.instarNeurons];

    for (n=0; n<PATTERN1.length; n++) {
      for (i=0; i<HEIGHT; i++) {
        for (j=0; j<WIDTH; j++) {

          input1[n][i*WIDTH+j] =
           (PATTERN1[n][i].charAt(j) == 'O') ? HI : LO;
          input2[n][i*WIDTH+j] =
           (PATTERN2[n][i].charAt(j) == 'O') ? HI : LO;
         }
       }
     normalizeInput();

   for (n=0; n<PATTERN1.length; n++) {
     this.ideal1[n][0] = Math.sin(n * 0.25 * Math.PI);
     this.ideal1[n][1] = Math.cos(n * 0.25 * Math.PI);
   }
}

public double sqr(double x) {
```

```
  return x*x;
}


void normalizeInput() {
  int  n,i;
  double length1, length2;

  for (n=0; n<PATTERN1.length; n++) {
    length1 = 0;
    length2 = 0;
    for (i=0; i<this.inputNeurons; i++) {
      length1  += sqr(this.input1[n][i]);
      length2 += sqr(this.input2[n][i]);
    }

    length1  = Math.sqrt(length1);
    length2 = Math.sqrt(length2);

    for (i=0; i<this.inputNeurons; i++) {
      input1[n][i] /= length1;
      input2[n][i] /= length2;
    }
  }
}

public BasicNetwork createNetwork() {
  CPNPattern pattern = new CPNPattern();
  pattern.setInputNeurons( this.inputNeurons );
  pattern.setInstarCount( this.instarNeurons );
  pattern.setOutstarCount( this.outstarNeurons );

  BasicNetwork network = pattern.generate();
  network.reset();

  return network;
}

public void trainInstar(BasicNetwork network,
  NeuralDataSet training) {

  int epoch = 1;

  Train train = new TrainInstar(network,training,0.1);

  for(int i=0;i<20;i++) {
```

```
    train.iteration();
    System.out.println("Training instar, Epoch #"
      + epoch + ", Error: " + train.getError() );
    epoch++;
    }
}

public void trainOutstar(BasicNetwork network,
  NeuralDataSet training) {
  int epoch = 1;

  Train train = new TrainOutstar(network,training,0.1);
  for(int i=0;i<20;i++) {
    train.iteration();
    System.out.println("Training outstar, Epoch #"
      + epoch + ", error=" + train.getError() );
     epoch++;
  }
}

public NeuralDataSet generateTraining(
  double[][] input,double[][] ideal) {
  NeuralDataSet result = new BasicNeuralDataSet(input,ideal);
  return result;
}

public double determineAngle(NeuralData angle) {

  double result;

  result = ( Math.atan2(angle.getData(0),
    angle.getData(1)) / Math.PI) * 180;
  if (result < 0)
    result += 360;

  return result;
}

public void test(
  BasicNetwork network,String[][] pattern,double[][] input) {

  for(int i=0;i<pattern.length;i++) {
    NeuralData inputData = new BasicNeuralData(input[i]);
    NeuralData outputData = network.compute(inputData);

    double angle = determineAngle(outputData);
```

```
      // display image
      for(int j=0;j<HEIGHT;j++){

        if( j==HEIGHT-1 )
          System.out.println("["+pattern[i][j]+"] -> "
          + ((int)angle) + " deg");
        else
          System.out.println("["+pattern[i][j]+"]");
      }

      System.out.println();
    }
  }

  public void run() {
    prepareInput();
    normalizeInput();
    BasicNetwork network = createNetwork();
    NeuralDataSet training =
      generateTraining(this.input1,this.ideal1);
    trainInstar(network,training);
    trainOutstar(network,training);
    test(network,PATTERN1,this.input1);
  }

  public static void main(String[] args) {
    RocketCPN cpn = new RocketCPN();
    cpn.run();
  }

}
```

We will look at the various parts of this network and how it is trained. We will begin with creating the CPN network.

Creating the CPN Network

The example creates the CPN network in the **createNetwork** method. This method is shown here.

```
public BasicNetwork createNetwork() {
```

A pattern is used to actually construct the CPN network. The **CPNPattern** can be used to do this. Here we communicate the size of the input, instar and outstar layers.

```
  CPNPattern pattern = new CPNPattern();
  pattern.setInputNeurons( this.inputNeurons );
  pattern.setInstarCount( this.instarNeurons );
        pattern.setOutstarCount( this.outstarNeurons );
```

The network is then generated and reset to random values.

```
  BasicNetwork network = pattern.generate();
  network.reset();
  return network;
}
```

Once the network has been created it is returned. The pattern hides the complexity of creating the network. If you would like to see how the network was actually constructed, refer to Appendix C.

We will now see how this neural network is trained. Because this is a hybrid neural network, training must occur in two steps. First, the instar layer is trained. This is unsupervised training. Once the competitive part of the network has been trained the outstar portion of the neural network is trained in a supervised fashion. We will look at both phases of training, beginning with the instar training.

Instar Training

The instar training is essentially a simplified version of the training used for the self-organizing map (SOM). However, it is simplified in that there is no concept of a neighborhood. This training occurs inside of the **trainInstar** method. This method is shown here.

```
public void trainInstar(BasicNetwork network,
  NeuralDataSet training) {
```

To perform the instar training we make use of the **TrainInstar** class.

```
  int epoch = 1;

  Train train = new TrainInstar(network,training,0.1);
  for(int i=0;i<20;i++) {
    train.iteration();
    System.out.println(
      "Training instar, Epoch #" + epoch + ", Error: " +
      train.getError() );
    epoch++;
  }
}
```

The training causes winning neurons to consolidate their wins. This encourages diffraction in the amount the output neurons creates and helps the network to better group the input data. We limit the training to 20 iterations. There is no reported error, so it is necessary to simply choose how long you would like to train the network.

Outstar Training

The outstar training is essentially a simplified version of backpropagation. However, it is much simpler because there are no hidden layers with which to contend. The outstar training occurs in the **trainOutstar** method. This method is shown here.

```
public void trainOutstar(BasicNetwork network,
  NeuralDataSet training) {
```

The outstar training occurs inside of the **TrainOutstar** class. Here we specify a learning rate of 1% and provide training data.

```
  int epoch = 1;

  Train train = new TrainOutstar(network,training,0.1);
```

Just as was done with instar training, we loop for 20 iterations. We display an error, which is based on how far the neural network's actual output is from the ideal output.

```
  for(int i=0;i<20;i++) {
```

```
    train.iteration();
    System.out.println(
      "Training outstar, Epoch #" + epoch + ", error="
      + train.getError() );
    epoch++;
  }
}
```

Once the outstar training is complete, the network is ready to use.

Using the CPN Network

The input to this neural network will be a small rocket image. The output from the neural network will be the angle that the rocket is pointed. One important part of this program is translating the output of the neural network to an angle. This is done by the **determineAngle** method. This method accepts neural network output and returns an angle. This method is shown here.

```
public double determineAngle(NeuralData angle) {
```

The output from the neural network determines the angle at which the rocket is believed to be traveling. There are two output neurons. They are fed to the "two argument" arc tangent function (ATAN2). The **atan2** function essentially translates a point, specified by the two parameters, into an angle. This angle, which is in radians, is translated into degrees before being returned.

```
  double result;

  result = ( Math.atan2(angle.getData(0),
    angle.getData(1)) / Math.PI) * 180;
  if (result < 0)
    result += 360;
  return result;
}
```

This method is made use of by the **test** method, which actually tests the neural network. This method is shown here.

```
public void test(BasicNetwork network,
  String[][] pattern,double[][] input) {
```

We will now loop over all of the patterns.

```
  for(int i=0;i<pattern.length;i++) {
```

We prepare the input for the neural network and then calculate the output.

```
    NeuralData inputData = new BasicNeuralData(input[i]);
    NeuralData outputData = network.compute(inputData);
```

The angle is determined from the output.

```
    double angle = determineAngle(outputData);
```

We now loop over the rows and columns of the input pattern to display it. We also display the degree output from the neural network on the last line of the pattern.

```
    for(int j=0;j<HEIGHT;j++) {
      if( j==HEIGHT-1 )
        System.out.println("["+pattern[i][j]+"] -> "
          + ((int)angle) + " deg");
      else
        System.out.println("["+pattern[i][j]+"]");
    }
  System.out.println();
}
```

Once the program completes all of the rockets and the network's guess at their degree, rotation is displayed.

Running the CPN Example

When you execute this program, you will see the instar training occur first, followed by the outstar training.

```
Training instar, Epoch #1, Error: 0.0
Training instar, Epoch #2, Error: 0.0
Training instar, Epoch #3, Error: 0.0
...
Training instar, Epoch #19, Error: 0.0
Training instar, Epoch #20, Error: 0.0
Training outstar, Epoch #1, error=0.6603311211122305
...
Training outstar, Epoch #19, error=0.6603311211122305
```

```
Training outstar, Epoch #20, error=0.6603311211122305
[            ]
[            ]
[     O      ]
[     O      ]
[    OOO     ]
[    OOO     ]
[    OOO     ]
[   OOOOO    ]
[   OOOOO    ]
[            ]
[            ] -> 0 deg

[            ]
[            ]
[         O  ]
[        O   ]
[      OOO   ]
[    OOO     ]
[   OOO      ]
[ OOOOO      ]
[OOOOO       ]
[            ]
[            ] -> 45 deg

[            ]
[            ]
[            ]
[  OO        ]
[  OOOOO     ]
[  OOOOOOO   ]
[  OOOOO     ]
[  OO        ]
[            ]
[            ]
[            ] -> 90 deg

...

[            ]
[            ]
[            ]
[        OO  ]
[    OOOOO   ]
[  OOOOOOO   ]
[    OOOOO   ]
```

```
[        OO  ]
[            ]
[            ]
[            ] -> 270 deg

[            ]
[            ]
[  O         ]
[   O        ]
[   OOO      ]
[    OOO     ]
[     OOO    ]
[     OOOOO  ]
[      OOOOO]
[            ]
[            ] -> 315 deg
```

Once the training is complete, the patterns are presented.

Where to Go from Here

After reading this book, you should have a good idea of how to make use of Encog. This book was kept very general, as it attempted to cover a broad range of topics applicable to Encog. We are also creating more specialized material for Encog. These can be found at the following URL:

http://www.heatonresearch.com/

We hope to produce several focused e-books on specific areas of Encog programming, such as more advanced market prediction and pattern recognition.

Encog is an open source project that has received contributions from several programmers. If you are interested in contributing code to the Encog project, please contact us at the above URL.

Summary

Most of this book has been about feedforward, recurrent, and self-organizing map neural networks. In this chapter you learned about some of the other network types supported by Encog. Though these network types are used much less frequently than the other types in this book, they do have

their uses. In this chapter we saw the Radial Basis Function network, the Adaptive Resonance Theory network and the Counter-Propagation network.

The Radial Basis Function (RBF) network is a network with a hidden layer based on radial basis functions. Radial basis functions can be used to approximate other complex functions. This gives the RBF hidden layer a much more advanced activation function than traditional, feedforward neural networks.

The Adaptive Resonance Theory (ART) network learns to group patters as they are presented to it. This network trains as it accepts patterns to classify. The output neurons represent the groups that the input neurons are classified into.

The Counter-Propagation Network (CPN) is a hybrid network that makes use of both supervised and unsupervised training. This network contains instar and outstar layers. The instar layer is trained in an unsupervised way; the outstar uses supervised training.

This concludes this book on Encog programming. Encog is an ever-evolving project. For more information on current Encog projects and additional articles about Encog, visit the following URL:

http://www.encog.org/

Encog is very much shaped by input from its users. We would love to hear about how you are using Encog and what new features may be beneficial as well. No software product, or book, is perfect. Bug reports are also very helpful as well. There is a forum at the above URL that can be used for the discussion of this book and Encog.

Terms

Adaptive Resonance Theory

Counter-Propagation Neural Network

Instar Training

Outstar Training

Plasticity

Radial Basis Function

Radial Basis Function Network

Questions for Review

1. How is an ART network similar to a SOM network? How do they differ?
2. Describe what happens during instar training.
3. What must be true of the input provided to an ART1 network?
4. Describe what happens during outstar training.
5. When is the ART network actually trained?
6. How does an ART network exhibit plasticity?
7. How is a RBF network different than traditional feedforward neural networks?
8. How do you train a RBF network in Encog?
9. What determines the number of groups an ART network can recognize?
10. Is Encog open source? Can anyone contribute to Encog?

Appendix A: Installing and Using Encog

- Downloading Encog
- Running Examples
- Running the Workbench

This appendix shows how to install and use Encog. This consists of downloading Encog from the Encog Web site, installing and then running the examples. You will also be shown how to run the Encog Workbench. Encog makes use of Java. This appendix assumes that you have already downloaded and installed Java JSE version 1.6 or later on your computer. The latest version of Java can be downloaded from the following Web site:

http://java.sun.com/

Java is a cross-platform programming language, so Encog can run on a variety of platforms. Encog has been used on Macintosh and Linux operating systems. However, this appendix assumes that you are using the Windows operating system. The screen shots illustrate procedures on the Windows 7 operating. However, Encog should run just fine on Windows XP or later.

It is also possible to use Encog with an IDE. Encog was developed primary using the Eclipse IDE. However, there is no reason why it should not work with other Java IDE's such as Netbeans or IntelliJ.

Installing Encog

You can always download the latest version of Encog from the following URL:

http://www.encog.org

On this page, you will find a link to download the latest version of Encog and find the following files at the Encog download site:

- The Encog Core
- The Encog Examples
- The Encog Workbench
- The Encog Workbench Source Code

For this book, you will need to download the first three files (Encog Core, and Encog Examples and Encog Workbench). There will be several versions of the workbench available. You can download the workbench as a Windows executable, a universal script file, or a Macintosh application. Choose the flavor of the workbench that is most suited to your computer. You do not need the workbench source code to use this book.

You should extract the Encog Core and Examples files for this first example. All of the Encog projects are built using ANT scripts. You can obtain a copy of ANT from the following URL.

http://ant.apache.org/

Encog contains an API reference in the core download. This documentation is contained in the standard Javadoc format. Instructions for installing Ant can be found at the above Web site. If you are going to use Encog with an IDE, it is not necessary to install Ant. Once you have correctly installed Ant, you should be able to issue the **ant** command from a command prompt. Figure A.1 shows the expected output of the **ant** command.

Figure A.1: Ant Successfully Installed

You should also extract the Encog Core, Encog Examples and Encog Workbench files into local directories. This appendix will assume that they have been extracted into the following directories:

- c:\encog-java-core-2.3.0\

- c:\encog-java-examples-2.3.0\
- c:\encog-workbench-win-2.3.0\

Now that you have installed Encog and Ant on your computer, you are ready to compile the core and examples. If you only want to use an IDE, you can skip to that section in this Appendix.

Compiling the Encog Core

Unless you would like to modify Encog itself, it is unlikely that you would need to compile the Encog core. Compiling the Encog core will recompile and rebuild the Encog core JAR file. It is very easy to recompile the Encog core using Ant. Open a command prompt and move to the following directory.

```
c:\encog-java-core-2.3.0\
```

From here, issue the following Ant command.

```
ant
```

This will rebuild the Encog core. If this command is successful, you should see output similar to the following:

```
C:\encog-java-core-2.3.0>ant
Buildfile: build.xml

init:

compile:

doc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source files for package org.encog...
  [javadoc] Loading source files for package org.encog.bot...
  [javadoc] Loading source files for package
org.encog.bot.browse...
  [javadoc] Loading source files for package
org.encog.bot.browse.extract...
  [javadoc] Loading source files for package
org.encog.bot.browse.range...
  [javadoc] Loading source files for package
org.encog.bot.dataunit...
  [javadoc] Loading source files for package org.encog.bot.rss...
```

```
  [javadoc] Loading source files for package org.encog.matrix...
  [javadoc] Loading source files for package org.encog.neural...
  [javadoc] Loading source files for package
org.encog.neural.activation...
...
  [javadoc] Loading source files for package
org.encog.util.math...
  [javadoc] Loading source files for package
org.encog.util.math.rbf...
  [javadoc] Loading source files for package
org.encog.util.randomize...
  [javadoc] Loading source files for package
org.encog.util.time...
  [javadoc] Constructing Javadoc information...
  [javadoc] Standard Doclet version 1.6.0_16
  [javadoc] Building tree for all the packages and classes...
  [javadoc] Building index for all the packages and classes...
  [javadoc] Building index for all classes...

dist:

BUILD SUCCESSFUL
Total time: 4 seconds
C:\encog-java-core-2.3.0>
```

This will result in a new Encog core JAR file being placed inside of the **lib** directory.

Compiling and Executing Encog Examples

The Encog examples are placed in a hierarchy of directories. The root example directory is located here.

```
c:\encog-java-examples-2.3.0\
```

The actual example JAR file is placed in a **lib** subdirectory off of the above directory. The examples archive that you downloaded already contains such a JAR file. It is not necessary to recompile the examples JAR file unless you make changes to one of the examples. To compile the examples, move to the root examples directory, given above.

Third-Party Libraries Used by Encog

Encog makes use of several third-party libraries. These third party libraries provide Encog with needed functionality. Rather than "reinvent the wheel", Encog makes use of these third-party libraries for needed functionality. That being said, Encog does try to limit the number of third-party libraries used so that installation is not terribly complex. The third-party libraries are contained in the following directory:

```
c:\encog-java-core-2.3.0\jar\
```

You will see the following JAR files there.

- hsqldb.jar
- junit-4.6.jar
- sl4j-api-1.5.6.jar
- sl4j-jdk14-1.5.6.jar

You may see different version numbers of these JARs as later versions will be released and included with Encog. These names of these JARs are listed here.

- The Hypersonic SQL Database
- JUnit
- Simple Logging Facade for Java (SLF4J)
- SLFJ Interface for JDK Logging

The Hypersonic SQL database is used internally by the Encog Unit Tests. As a result, the HSQL JAR does not need to be used when Encog is actually run. The same is true for JUnit, which is only used for unit tests.

The two SLF4J JARs are required. Encog will log certain events to make debugging and monitoring easier. Encog uses SLF4J to accomplish this. SLF4J is not an actual logging system, but rather a facade for many of the popular logging systems. This allows frameworks, such as Encog, to not need to dictate which logging API's an application using their framework should use. The SLF4J JAR must always be used with a second SLF4J JAR file which defines what actual logging API to use. Here we are using **sl4j-jdk14-1.5.6.jar**, which states that we are using the JDK logging features that are built into Java. However, by using a different interface JAR we could easily switch to other Java logging systems, such as LOG4J.

Running an Example from the Command Line

When you execute a Java application that makes use of Encog, the appropriate third- party JARs must be present in the Java **classpath**. The following command shows how you might want to execute the **XORResilient** example:

```
java -server -classpath .\jar\encog-core-2.3.0.jar;.\jar\slf4j-
api-1.5.6.jar;.\jar\slf4j-jdk14-1.5.6.jar;.\lib\encog-examples-
2.3.0.jar org.encog.examples.neural.xorresilient.XORResilient
```

If the command does not work, make sure that the JAR files located in the **lib** and **jar** directories are present and named correctly. There may be new versions of these JAR files since this document was written. If this is the case, you will need to update the above command to match the correct names of the JAR files.

The examples download for Encog contains many examples. The Encog examples are each designed to be relatively short, and are usually console applications. This makes them great starting points for creating your own application to use a similar neural network technology as the example you are using. To run a different example, specify the package name and class name as was done above for **XORResilient**.

You will also notice from the above example that the **-server** option was specified. This runs the application in "Java Server Mode". Java Server mode is very similar to the regular client mode. Programs run the same way, except in server mode it takes longer to start the program. But for this longer load time, you are rewarded with greater processing performance. Neural network applications are usually "processing intense". As a result, it always pays to run them in "Server Mode".

Using Encog with the Eclipse IDE

The examples can also be run from an IDE. I will show you how to use the examples with the Eclipse IDE. The examples download comes with the necessary Eclipse IDE files. As a result, you can simply import the Encog examples into Eclipse. Eclipse is not the only IDE with which Encog can be used. Encog should work fine with any Java IDE. Eclipse can be downloaded from the following URL:

http://www.eclipse.org/

Once Eclipse has been started you should choose to import a project. To do this choose "Import" from the "File" menu. Once you have done this, you should see the Eclipse Import dialog box, as seen in Figure A.2.

Figure A.2: Import into Eclipse

From the "Import into Eclipse" dialog box, choose "Existing Projects into Workspace". This will allow you to choose the folder to import, as seen in Figure A.3.

Figure A.3: Choose Source Folder

You should choose whichever directory you installed the Encog examples into, such as the following directory:

```
c:\encog-java-examples-2.3.0\
```

Once you have chosen your directory, you will be given a list of the projects available in that directory. There should only be one project, as shown in Figure A.4.

Figure A.4: Choose the Project to Import

Once the project has been imported, it will appear as a folder inside of the IDE. This folder will have a "Red X" over it if any sort of error occurs. You can see a project with an import error in Figure A.5.

Figure A.5: A Project with Errors

If you had any errors importing the project, the next section describes how to address them. If there were no errors, you can safely skip the next section and continue with "Executing an Example".

Resolving Path Errors

It is not unusual to have errors when importing the Encog examples project. This is usually because Eclipse failed to figure out the correct paths of the JAR files used by the examples. To fix this, it is necessary to remove, and then re-add the JAR files used by the examples. To do this, right click the project folder in Eclipse and choose properties. Select "Java Build Path", and you will see Figure A.6.

Figure A.6: The Java Build Path

Select the four JAR files used by the examples and choose "Remove". Once the JARs are removed, you must re-add them so the examples will compile. To add the JAR files select the "Add JARs" button. This will present you with a file selection dialog box that allows you to navigate to the four required JAR files. They will be located in the following directory:

```
c:\encog-java-examples-2.3.0\jar\
```

Figure A.7 shows the four JAR files being selected.

Figure A.7: JAR Files

Now that the JAR files have been selected, there should be no errors remaining. We are now ready to run an example.

Running an Example

To run an example, simply navigate to the class that you wish to import in the IDE. Right click the class and choose "Run As" and then select "Run as Java Application". This will run the example, and show the output in the IDE. Figure A.8 shows the ADALINE example run from the IDE.

Figure A.8: The ADALINE Example

Appendix B: Example Locations

Encog comes with quite a few examples. This book does not maintain its own set of examples, rather it draws upon the examples already included in the Encog download. You can download all of the Encog examples from the following URL:

http://www.encog.org

This appendix provides a quick reference to packages that contain examples from each of the chapters.

Chapter 1 Examples

Chapter 1 contains only one example: training with RPROP.

Training with RPROP:
```
org.encog.examples.neural.xorresilient
```

Chapter 2 Examples

Chapter 2 provides two examples. The first example expands on the example in Chapter 1.

Simple Feedforward Neural Network
```
org.encog.examples.neural.xorresilient
```

Hopfield Neural Network
```
org.encog.examples.neural.hopfield
```

Chapter 3 Examples

Chapter 3 focuses on activation functions and did not have a complete program as an example.

Chapter 4 Examples

Chapter 4 focuses on the Encog Workbench and had no code example.

Chapter 5 Examples

Chapter 5 introduces propagation training and presents four examples.

Using Backpropagation
```
org.encog.examples.neural.xorbackprop
```

Using the Manhattan Update Rule
```
org.encog.examples.neural.xormanhattan
```

Using Resilient Propagation
```
org.encog.examples.neural.xorresilient
```

Using Multipropagation
```
org.encog.examples.neural.benchmark
```

Chapter 6 Examples

Chapter 6 has one large example on forest cover prediction.

Forest Cover Prediction
```
org.encog.examples.neural.forest.feedforward
```

Chapter 7 Examples

Chapter 7 has two examples that deal with Encog persistence and serialization.

Encog Persistence Example
```
org.encog.examples.neural.persist.EncogPersistence
```

Encog Serialization Example
```
org.encog.examples.neural.persist.Serial
```

Chapter 8 Examples

Chapter 8 includes one large example that simulates a lunar lander.

Lunar Lander
```
org.encog.examples.neural.lunar
```

Chapter 9 Examples

Chapter 9 includes one example that demonstrates how to use a self-organizing map (SOM).

Self-Organizing Maps
```
org.encog.examples.neural.gui.som
```

Chapter 10 Examples

Chapter 10 includes two predictive neural networks. One predicts sunspots, and the other attempts to predict the prices of various stocks.

Sun Spot Prediction
```
org.encog.examples.neural.predict.sunspot
```

Market Prediction
```
org.encog.examples.neural.predict.market
```

Chapter 11 Examples

Chapter 11 introduces one large example on how to perform image recognition.

Image Recognition
```
org.encog.examples.neural.image
```

Chapter 12 Examples

Chapter 12 introduces an example of the Elman neural network.

Elman Neural Network
```
org.encog.examples.neural.recurrant.elman
```

Chapter 13 Examples

Chapter 13 introduces an example that expands the stock market example, in Chapter 10, to use incremental pruning.

Market Prediction Pruning
```
org.encog.examples.neural.predict.market
```

Chapter 14 Examples

Chapter 14 introduces three examples that include an RBF network, an ART1 network and a Counter-Propagation (CPN) network.

Radial Basis Function Network
```
org.encog.examples.neural.xorradial
```

Adaptive Resonance Theory (ART1) Network
```
org.encog.examples.neural.art.art1
```

Counter-Propagation Network
```
org.encog.examples.neural.cpn
```

Appendix C: Encog Patterns

Encog provides pattern classes for many common neural network types. In this appendix we will look at the code necessary to construct these types of neural networks. We will also see the code that you would use to create any of these neural networks "from scratch", without using a pattern.

Encog contains patterns for many different neural network types. Not all of these types were covered in this book. For completeness, this appendix covers all of the patterns included in Encog. For those patterns covered by this book, chapter references are included.

Adaline Neural Network

Professor Bernard Widrow and his graduate student Ted Hoff developed the ADALINE neural network. It is a very simple neural network usually used for pattern recognition. ADALINE is short for Adaptive Linear Neuron or Adaptive Linear Element. It is considered a single layer neural network. Though an activation function forms a sort of primitive output layer. Weighted connections are made to this activation function. A threshold, or bias, is also provided for. The output from an ADALINE neural network is usually bipolar. The Adaline neural network was not covered in this book.

Network Diagram

Creating from a Pattern

```
ADALINEPattern pattern = new ADALINEPattern();
pattern.setInputNeurons(inputNeurons);
```

```
pattern.setOutputNeurons(outputNeurons);
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
BasicNetwork network = new BasicNetwork();

Layer inputLayer = new BasicLayer(new ActivationLinear(), false,
                        inputNeurons);
Layer outputLayer = new BasicLayer(new ActivationLinear(), true,
                        outputNeurons);

network.addLayer(inputLayer);
network.addLayer(outputLayer);
network.getStructure().finalizeStructure();
```

ART1 Neural Network

Adaptive Resonance Theory (ART) is a form of neural network developed by Stephen Grossberg and Gail Carpenter. There are several versions of the ART neural network, which are numbered ART-1, ART-2 and ART-3. The ART neural network is trained using either a supervised or unsupervised learning algorithm, depending on the version of ART being used. ART neural networks are used for pattern recognition and prediction. The ART1 neural network was covered in Chapter 14.

Network Diagram

Creating from a Pattern

```
ART1Pattern pattern = new ART1Pattern();
pattern.setInputNeurons(inputNeurons);
pattern.setOutputNeurons(outputNeurons);
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
BasicNetwork network = new BasicNetwork(new ART1Logic());

Layer layerF1 = new BasicLayer(new ActivationLinear(), false,
inputNeurons);
Layer layerF2 = new BasicLayer(new ActivationLinear(), false,
outputNeurons);

Synapse synapseF1toF2 = new WeightedSynapse(layerF1, layerF2);
Synapse synapseF2toF1 = new WeightedSynapse(layerF2, layerF1);
layerF1.getNext().add(synapseF1toF2);
layerF2.getNext().add(synapseF2toF1);

network.tagLayer(BasicNetwork.TAG_INPUT, layerF1);
network.tagLayer(BasicNetwork.TAG_OUTPUT, layerF2);
network.tagLayer(ART1Pattern.TAG_F1, layerF1);
network.tagLayer(ART1Pattern.TAG_F2, layerF2);

network.setProperty(ARTLogic.PROPERTY_A1, a1);
network.setProperty(ARTLogic.PROPERTY_B1, b1);
network.setProperty(ARTLogic.PROPERTY_C1, c1);
network.setProperty(ARTLogic.PROPERTY_D1, d1);
network.setProperty(ARTLogic.PROPERTY_L, l);
network.setProperty(ARTLogic.PROPERTY_VIGILANCE, v);

network.getStructure().finalizeStructure();
```

Bidirectional Associative Memory (BAM)

Bidirectional Associative Memory (BAM) is a type of neural network developed by Bart Kosko in 1988. The BAM is a recurrent neural network that works similarly that allows patterns of different lengths to be mapped bidirectionally to other patterns. This allows it to act as almost a two-way hash map. During training the BAM is fed pattern pairs. The two halves of each pattern do not have to be the same length. However, all patterns must be of the same overall structure. The BAM can be fed a distorted pattern on

either side and will attempt to map to the correct value. The BAM neural network was not covered in this book.

Network Diagram

Creating from a Pattern

```
BAMPattern pattern = new BAMPattern();
pattern.setF1Neurons(INPUT_NEURONS);
pattern.setF2Neurons(OUTPUT_NEURONS);
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
BasicNetwork network = new BasicNetwork(new BAMLogic());

Layer f1Layer = new BasicLayer(new ActivationBiPolar(), false,
  f1Neurons);
Layer f2Layer = new BasicLayer(new ActivationBiPolar(), false,
  f2Neurons);
Synapse synapseInputToOutput = new WeightedSynapse(f1Layer,
  f2Layer);
Synapse synapseOutputToInput = new WeightedSynapse(f2Layer,
  f1Layer);
f1Layer.addSynapse(synapseInputToOutput);
f2Layer.addSynapse(synapseOutputToInput);

network.tagLayer(BAMPattern.TAG_F1, f1Layer);
network.tagLayer(BAMPattern.TAG_F2, f2Layer);

network.getStructure().finalizeStructure();
```

Boltzmann Machine

A Boltzmann machine is a type of neural network developed by Geoffrey Hinton and Terry Sejnowski. It appears identical to a Hopfield neural network except it contains a random nature to its output. A temperature value is present that influences the output from the neural network. As this temperature decreases, so does the randomness. This is called simulated

annealing. Boltzmann networks are usually trained in an unsupervised mode. However, supervised training can be used to refine what the Boltzmann machine is recognizing. The Boltzmann machine was covered in Chapter 12.

Network Diagram

Creating from a Pattern

```
BoltzmannPattern pattern = new BoltzmannPattern();
pattern.setInputNeurons(NEURON_COUNT);
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
final BasicNetwork result = new BasicNetwork(new
BoltzmannLogic());
result.setProperty(BoltzmannLogic.PROPERTY_ANNEAL_CYCLES,
  annealCycles);
result.setProperty(BoltzmannLogic.PROPERTY_RUN_CYCLES,
runCycles);
result.setProperty(BoltzmannLogic.PROPERTY_TEMPERATURE,
  temperature);
result.addLayer(layer);
layer.addNext(layer);
result.getStructure().finalizeStructure();
```

Counter-Propagation Neural Network

Counter-propagation Neural Networks (CPN) were developed by Professor Robert Hecht-Nielsen in 1987. CPN neural networks are hybrid neural networks, employing characteristics of both a feedforward neural network and a self-organizing map (SOM). The CPN is composed of three layers, the input, the instar and the outstar. The connection from the input to the instar layer is competitive, with only one neuron being allowed to win. The connection between the instar and outstar is feedforward. The layers are trained separately, using instar training and outstar training. The CPN

network is good at classification. This network type was covered in Chapter 14.

Network Diagram

Creating from a Pattern

```
CPNPattern pattern = new CPNPattern();
pattern.setInputNeurons( inputNeurons );
pattern.setInstarCount( instarNeurons );
pattern.setOutstarCount( outstarNeurons );
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
Layer input, instar, outstar;

BasicNetwork network = new BasicNetwork();
network.addLayer(input = new BasicLayer(new ActivationLinear(),
false,
  inputCount));
network.addLayer(instar = new BasicLayer(new
ActivationCompetitive(),
  false, instarCount));
network.addLayer(outstar = new BasicLayer(new ActivationLinear(),
  false, outstarCount));
```

```
network.getStructure().finalizeStructure();
network.reset();

network.tagLayer(BasicNetwork.TAG_INPUT, input);
network.tagLayer(BasicNetwork.TAG_OUTPUT, outstar);
network.tagLayer(CPNPattern.TAG_INSTAR, instar);
network.tagLayer(CPNPattern.TAG_OUTSTAR, outstar);
```

Elman Neural Network

The Elman neural network is a simple recurrent neural network (SRN) developed by Jeffrey L. Elman in 1990. This network type consists of an input layer, a hidden layer, and an output layer. In this way it resembles a three-layer, feedforward neural network. However, it also has a context layer. This context layer is fed, without weighting, the output from the hidden layer. The Elman network then remembers these values and outputs them on the next run of the neural network. These values are then sent, using a trainable weighted connection, back into the hidden layer. Elman neural networks are very useful for predicting sequences, since they have a limited short-term memory. The Elman neural network was covered in Chapter 12.

Network Diagram

Creating from a Pattern

```
ElmanPattern pattern = new ElmanPattern();
pattern.setActivationFunction(new ActivationTANH());
pattern.setInputNeurons(1);
pattern.addHiddenLayer(2);
pattern.setOutputNeurons(1);
return pattern.generate();
```

Creating from Scratch

```
Layer context = new ContextLayer(hiddenNeurons);
BasicNetwork network = new BasicNetwork();

Layer input = new BasicLayer(activation, true,
  inputNeurons);
network.addLayer(input);

Layer hidden = new BasicLayer(activation, true,
  hiddenNeurons);
network.addLayer(hidden);
hidden.addNext(context, SynapseType.OneToOne);
context.addNext(hidden);
Layer output = new BasicLayer(activation, true,
  outputNeurons);
network.addLayer(output);
network.getStructure().finalizeStructure();
```

Feedforward Neural Network

The feedforward neural network, or perceptron, is a type of neural network first described by Warren McCulloch and Walter Pitts in the 1940s. The feedforward neural networks, and its variants, are the most widely used form of neural network. The feedforward neural network is often trained with the backpropagation training technique, though there are other more advanced training techniques, such as resilient propagation. The feedforward neural network uses weighted connections from an input layer to zero or more hidden layers, and finally to an output layer. It is suitable for many types of problems. The feedforward neural network was introduced in Chapter 1, and covered in many other chapters.

Network Diagram

Creating from a Pattern

```
FeedForwardPattern pattern = new FeedForwardPattern();
pattern.setInputNeurons(input);
pattern.setOutputNeurons(output);
pattern.setActivationFunction(new ActivationSigmoid());
pattern.addHiddenLayer(hidden1);
pattern.addHiddenLayer(hidden2);
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(2));
network.addLayer(new BasicLayer(3));
network.addLayer(new BasicLayer(1));
network.getStructure().finalizeStructure();
```

Hopfield Neural Network

Dr. John Hopfield developed the Hopfield neural network in 1979. The Hopfield network is a single layer recurrent neural network. The Hopfield network always maintains a "current state" which is the current output of the neural network. The Hopfield neural network also has an energy property, which is calculated exactly the same as the temperature property of the Boltzmann machine. The Hopfield network is trained for several patterns. The state of the Hopfield network will move towards the closest patter, thus "recognizing" that pattern. As the Hopfield network moves towards one of these patterns, the energy lowers. The Hopfield neural network was covered in Chapter 12.

Network Diagram

Creating from a Pattern

```
HopfieldPattern pattern = new HopfieldPattern();
pattern.setInputNeurons(WIDTH*HEIGHT);
BasicNetwork hopfield = pattern.generate();
```

Creating from Scratch

```
final Layer layer = new BasicLayer(new ActivationBiPolar(),
  false, neuronCount);

final BasicNetwork result = new BasicNetwork(
  new HopfieldLogic());
result.addLayer(layer);
layer.addNext(layer);
result.getStructure().finalizeStructure();
```

Jordan Neural Network

The Jordan Neural Network is a simple recurrent network (SRN) developed by Michael I. Jordan in 1986. The context layer holds the previous output from the output layer and then echoes that value back to the hidden layer's input. The hidden layer then always receives input from the previous iteration's output layer. Jordan neural networks are generally trained using genetic, simulated annealing, or one of the propagation techniques. Jordan neural networks are typically used for prediction. The Elman neural network was covered in Chapter 12.

Network Diagram

Creating from a Pattern

```
ElmanPattern pattern = new ElmanPattern();
pattern.setActivationFunction(new ActivationTANH());
pattern.setInputNeurons(1);
pattern.addHiddenLayer(2);
pattern.setOutputNeurons(1);
return pattern.generate();
```

Creating from Scratch

```
Layer input = new BasicLayer(activation, true,
  inputNeurons);
Layer hidden = new BasicLayer(activation, true,
  hiddenNeurons);
Layer output = new BasicLayer(activation, true,
  outputNeurons);
Layer context = new ContextLayer(outputNeurons);
BasicNetwork network = new BasicNetwork();
network.addLayer(input);
network.addLayer(hidden);
network.addLayer(output);

output.addNext(context, SynapseType.OneToOne);
context.addNext(hidden);
```

```
network.getStructure().finalizeStructure();
network.reset();
```

Radial Basis Function Neural Network

The Radial Basis Function neural network contains a hidden layer based on radial basis functions (RBF). A radial basis function is a function that peaks in the center and rapidly falls off in each direction along an axis. One of the most common examples of a RBF is the Gaussian function. The hidden layer consists of one or more RBF's. This allows for a complex function to be modeled inside of the hidden layers. RBF neural networks are used for a variety of purposes, such as function approximation and prediction. The Radial Basis Function network was covered in Chapter 14.

Network Diagram

Creating from a Pattern

```
RadialBasisPattern pattern = new RadialBasisPattern();
pattern.setInputNeurons(2);
pattern.addHiddenLayer(4);
pattern.setOutputNeurons(1);
BasicNetwork network = pattern.generate();
```

Creating from Scratch

```
Layer input = new BasicLayer(new ActivationLinear(), false,
  inputNeurons);
Layer output = new BasicLayer(new ActivationLinear(), true,
outputNeurons);
BasicNetwork network = new BasicNetwork();
RadialBasisFunctionLayer rbfLayer = new RadialBasisFunctionLayer(
  hiddenNeurons);
network.addLayer(input);
network.addLayer(rbfLayer, SynapseType.Direct);
network.addLayer(output);
network.getStructure().finalizeStructure();
network.reset();
```

```
network.tagLayer(RBF_LAYER, rbfLayer);
rbfLayer.randomizeGaussianCentersAndWidths(0, 1);
```

Recurrent Self-Organizing Map

The recurrent self-organizing map (RSOM) is a self-organizing map that has a recurrent layer. This combines elements of the Jordan Neural Network with the Kohonen Self Organizing Map. RSOM's are good at classifying temporal sequences. For example, it is good at analyzing historical index information and determining whether we are in a period of economic growth or decline. The Recurrent Self Organizing Map was not covered in this book.

Network Diagram

Creating from a Pattern

```
RSOMPattern pattern = new RSOMPattern();
pattern.setInputNeurons(inputNeurons);
pattern.setOutputNeurons(outputNeurons);
BasicNetwork result = pattern.generate();
```

Creating from Scratch

```
Layer output = new BasicLayer(new ActivationLinear(), false,
  outputNeurons);
Layer input = new BasicLayer(new ActivationLinear(), false,
  inputNeurons);

BasicNetwork network = new BasicNetwork();
Layer context = new ContextLayer(outputNeurons);
network.addLayer(input);
network.addLayer(output);
```

```
output.addNext(context, SynapseType.OneToOne);
context.addNext(input);

network.getStructure().finalizeStructure();
```

Self-Organizing Map

The Self-Organizing Map (SOM) is a neural network type introduced by Teuvo Kohonen. SOM's are used to classify data into groups. The Self Organizing Map was covered in Chapter 9.

Network Diagram

Creating from a Pattern

```
SOMPattern pattern = new SOMPattern();
pattern.setInputNeurons(3);
pattern.setOutputNeurons(MapPanel.WIDTH*MapPanel.HEIGHT);
result = pattern.generate();
```

Creating from Scratch

```
Layer input = new BasicLayer(new ActivationLinear(), false,
  inputNeurons);
Layer output = new BasicLayer(new ActivationLinear(), false,
  outputNeurons);

BasicNetwork network = new BasicNetwork();
network.addLayer(input);
```

```
network.addLayer(output);
network.getStructure().finalizeStructure();
```

Glossary

Activation Function: A function used to scale the output of a neural network layer. If this activation function has a derivative, then propagation training can be used on the neural network.

Adaptive Resonance Theory (ART1): A neural network architecture that learns to classify patterns as they are presented. (2, 14)

Annealing Cycles: The number of cycles that the simulated annealing training method will use per iteration. (8)

Artificial Intelligence (AI): A branch of computer science that seeks to give machines human-like intelligence. Neural networks are one tool used in AI. (1)

Artificial Neural Network (ANN): See neural network.

Autoassociation: A means of pattern recognition where the output of the neural network is the entire pattern it recognized. The network returns the same data with which it was trained. (2, 12)

Backpropagation: A propagation algorithm where the error gradients are applied directly to the weight matrix, scaled only by a learning rate. (5)

Backward Pass: One of two passes in propagation training where the error gradients are calculated and used to determine changes that should be made to the weight matrix of a neural network. (5)

Basic Layer: A very versatile Encog neural network layer type that is used in many different neural networks. It has a number of neurons, an activation function and optional threshold values. (2)

Batch Training: The accumulation of the weight matrix deltas from a number of training set elements before these deltas are actually applied to the weight matrix. (5)

Best Matching Unit (BMU): The neuron, in a Self Organizing Map (SOM), that had the shortest Euclidean distance to the training data element. (9) (BMU)

Bidirectional Associative Memory: A neural network type that forms bidirectional associations between two layers. (2)

Biological Neural Network: The actual neural network contained in humans and other animals. This is what an artificial neural network attempts to simulate to some degree. (1)

BiPolar: Activation Function: An activation function to support bipolar numbers. This maps a true value to 1 and a false value to -1. (3)

Black Box: A computer system where the inputs and outputs are well understood; however, the means to produce the output is not known. (1)

Boltzmann Machine: A simple recurrent neural network that adds a temperature element that randomizes the output of the neural network. (2, 12)

Bounding Box: A box that is drawn around the relevant part of an image. (11)

Competitive Activation Function: An activation function where only a certain number of neurons are allowed to fire. These winning neurons were the ones with the highest activation. (3)

Competitive Training: A training method, typically used by a Self Organizing Map, which chooses a best matching unit (BMU) and further strengthens that neuron's activation for the current training element. (9)

Context Layer: An Encog layer type that remembers the input values from the last iteration and uses those values as the output for the current iteration. This layer type is used for simple, recurrent neural network types such as the Elman and Jordan neural networks. (1, 12)

Counter-Propagation Neural Network: A hybrid neural network that combines elements of a regular feedforward neural network and a Self Organizing Map. Counter-Propagation Neural Networks use both supervised and unsupervised training, which are called outstar and instar training respectively. (14) (CPN)

Crop: The process where irrelevant portions of an image are removed. (11)

Crossover: A simulation of the biological mating process in a Genetic Algorithm where elements from two "parent" solutions are combined to

produce "offspring solutions" that share characteristics of both "parents". (8)

CSV File: A comma separated value file. These are typically used as training input for an Encog neural network. (4, 6)

Derivative: In calculus, a measure of how a function changes as its input changes. Propagation training uses the derivative of the activation function to calculate an error gradient. (3)

Direct Synapse: An Encog synapse that directly connects two layers of neurons. This layer type is typically used in a Radial Basis Function neural network. (2)

Downsample: The process where the resolution and color depth of an image are reduced. This can make the image easier to recognize for a neural network. (11)

EG File: An XML based file that Encog uses to store neural networks, training data and other objects. (6)

Elman Neural Network: A simple recurrent neural network where the output of the hidden layer is fed to a context layer and then fed back into the hidden layer. The Elman Neural Network can be useful for temporal data. (2, 12)

Encog: An Artificial Intelligence Framework for Java, .Net and Silverlight that specializes in neural network applications. (1)

Encog Benchmark: A means of calculating the performance of Encog on a particular machine. The benchmark is expressed as a number; a lower number indicates a faster machine. This benchmark uses multithreaded training and will score multicore machines higher. (4)

Encog File: See EG file.

Encog Workbench: A GUI application that allows Encog EG files to be edited. (1)

Ending Temperature: The temperature at which a simulated annealing iteration should end. The temperature defines the degree to which the weights are randomly perturbed in a simulated annealing cycle. (8)

Epoch: See iteration.

Equilateral Normalization: A means by which nominal data is normalized for a neural network. Often provides better results than the competing one-of-n normalization. (6)

Equilibrium: The point at which further iterations to a thermal neural network produce no further meaningful change. (12)

Error Rate: The degree to which the output of neural network differs from the expected output. (1, 5)

Euclidian Distance: The square root of the squares of the individual differences in set of data. Euclidian Distance is often used to determine which vector is most similar to a comparison vector. (6)

Evaluation: The process in which a trained neural network is evaluated against data that was not in the original training set. (6)

Feedforward Neural Network: A multilayer neural network where connections only flow forward. (1)

Field Group: A group of normalization output fields that depend on each other to calculate the output value. (6)

Forward Pass: One of two passes in propagation training where the output from the neural network is calculated for a training element. (5)

Future Window: The data that a temporal neural network is attempting to predict. (10)

Gaussian Activation Function: An activation based on the Gaussian function. (3)

Gaussian Neighborhood Function: A neighborhood function, used for a Self Organizing Map, based on the Gaussian function. (9)

Genetic Algorithms: An Artificial Intelligence algorithm that attempts to derive a solution by simulating the biological process of natural selection. (8)

Gradient Error: A value that is calculated for individual connections in the neural network that can provide insight into how the weight should be changed to lower the error of the neural network. (5)

Greedy Training: A training strategy where iterations that do not lower the error rate of a neural network are discarded. (12)

Hidden Layer: Layers in a neural network that exists between the input and output layers. They are used to assist the neural network in pattern recognition. (1)

Hopfield Neural Network: A thermal neural network that contains a single self-connected layer. (2, 12)

Hybrid Training: Training a neural network with more than one training algorithm. (12)

Hyperbolic Tangent Activation Function: An activation function that makes use of the hyperbolic tangent function. This activation function can return both positive and negative numbers. (2)

Ideal Output: The expected output of a neural network. (5)

Incremental Pruning: A means to automatically determine an efficient number of hidden neurons by increasing the hidden layer and testing each potential configuration. (13)

Input Field: An Encog normalization field that accepts raw, un-normalized data. Input fields are provided that accept input from a number of different sources. (6)

Input Layer: The layer in a neural network that accepts input. (1)

Instar Training: An unsupervised training technique used for the counter-propagation neural network. (14)

Intensity Downsample: A downsample technique where color information is discarded, and only the intensity, or brightness, of color is used. (11)

Iteration: The basic unit of training where each iteration attempts to improve the neural network in some way. (1, 5)

Jordan Neural Network: A simple recurrent neural network where the output of the output layer is fed to a context layer and then fed back into the hidden layer. The Jordan Neural Network can be useful for temporal data. (2, 12)

Kohonen Neural Network: Another name for the Self Organizing Map (SOM). (9)

Layer: A group of similar neurons in a neural network. (1)

Layer Tag: The means by which Encog names layers. (2)

Learning rate: The percentage of a change to the weight matrix that is allowed to occur. This allows changes that would overwhelm the neural network to be scaled to less dramatic values. (5)

Lesser GNU Public License (LGPL): The license under which Encog is licensed. (1)

Linear Activation Function: An activation function based on a simple linear function. (3)

LOG Activation Function: An activation function based on logarithms. (3)

Long Term Memory: The weights and threshold values of a neural network. (1)

Lunar Lander Game: A classic computer game where the user fires thrusters to produce as soft a landing as possible, without running out of fuel. (8)

Manhattan Update Rule: A propagation training technique where only the sign of the error gradient is used to determine the direction to change the connection weights. The magnitude of the error gradients is discarded. (5)

Memory Collection: Encog persistence where the entire EG file is loaded into memory. (7)

Momentum: The degree to which weight deltas from the pervious iteration are applied to the current iteration. Used in backpropagation to help avoid local minima. (5)

Multicore: A computer capable of concurrently executing multiple threads. Software must be written to be multithreaded to use these machines to their full potential. (5)

Multiplicative Normalization: A normalization technique to adjust a vector to sum to one. Multiplicative normalization has the effect of only using the magnitude of the input vector. To use sign and magnitude, z-axis normalization should be considered. (6)

Multithreaded: A programming technique where the programming task is divided among multiple threads. This allows a multicore machine to greatly reduce the amount of time a program can take to execute. (5)

Mutation: A technique used in Genetic Algorithms where the offspring are randomly changed in some way. (8)

Neighborhood Function: A function that scales training in a Self Organizing Map to neurons near the best matching unit. (9)

Network Pattern: A common neural network type, such as a Self Organizing Map, Elman network or Jordan network. Encog provides classes that assist in the creation of these neural network types. (2)

Neural Logic: Encog classes that show Encog how to calculate the output for various types of neural networks. (2)

Neural Network: A computerized simulation of an actual biological neural network. Sometimes referred to as an Artificial Neural Network (ANN); however, typically referred to as simply a "neural network". (1)

Neural Network Properties: Operating parameters that certain neural network types require Encog to associate with the neural network. (2)

Nominal Value: A value that is a member of a set, for example, male or female. (6)

Normalization: The process where numbers are scaled in order to be acceptable input to a neural network. (6)

Normalization Target: Where the Encog normalization classes are to store the results from the normalization value. (6)

Numeric Value: A number value that is to be normalized that has meaning as a number. For example, altitude would be a numeric value, but a postal code would not. (6)

One-to-One Synapse: An Encog synapse that directly connects each neuron in a layer to the corresponding neuron in another layer. A One-to-One Synapse is typically used to connect a basic layer to a context layer. (2, 12)

One-of-N Normalization: A means by which nominal data is normalized for a neural network. Often provides inferior results than the competing equilateral normalization. (6)

Online Training: Training where the weight deltas are applied as soon as they are calculated. (5)

Output Field: A normalization field that specified how an input field, or group of input fields, should be normalized. (6)

Output Layer: The layer of a neural network that produces output. (1)

Outstar Training: A supervised training technique used for the counter-propagation neural network. (14)

Past Window: The values on which a temporal neural network bases future predictions. (10)

Pattern: Data that is fed into a neural network. (2)

Persistence: The ability to store data in a permanent form. Encog uses Encog EG files for persistence. (7)

Plasticity: The ability of a neural network to change as data is fed to it. (14)

Propagation Training: A group of training techniques that use error gradients to provide insight into how to update the weights of a neural network to achieve lower error rates. Forms of propagation training include backpropagation, resilient propagation, the Manhattan Update Rule, and others. (5)

Pruning: Attempts to optimize the number of hidden neurons in a neural network. (13)

Radial Basis Activation Function: An activation function based on a radial basis function. (2)

Radial Basis Function (RBF): A function with its maximum value at its peak that decreases rapidly. (2, 14)

Radial Basis Function Layer: The layer, in a radial basis function network, that uses a compound radial basis function as its activation function. (2)

Radial Basis Function Network: A special type of neural network that makes use of a radial basis function layer. (14)

Recurrent Neural Network: A neural network that has connections back to previous layers. (1)

Resilient Propagation (RPROP): A propagation training technique that uses independent delta values for every connection in the network. This is one of the most efficient training algorithms offered by Encog. (1, 5)

RGB: The red, green and blue values that make up an image. (11)

RGB Downsample: A means of downsampling that preserves the color values of an image. (11)

Scaling: See downsampling. (11)

Score: A numeric value used to rank solutions provided by Genetic Algorithms and Simulated Annealing. (8)

Segregator: An Encog normalization object that excludes certain elements, based on the criteria provided. (6)

Selective Pruning: A pruning method where the weakest neurons are selected and removed. (13)

Self-Organizing Map (SOM): A neural network structure that organizes similar input patterns. (9)

Self-Connected Layer: A layer in a neural network that is connected to itself. (2, 12)

Serializable: A class that can be serialized. (7)

Short Term Memory: A context layer provides neural network short-term memory. (1)

Sigmoid Activation Function: An activation function based on the Sigmoid function. This activation function only produces positive values. (2, 3)

Simple Recurrent Neural Network (SRN): A neural network that has a recurrent connection through a context layer. The most typical SRN types are the Elman and Jordan neural networks. (12)

Simulated Annealing: A training technique that simulates the metallurgical annealing process. (8)

Sine Activation Function: An activation function based on the trigonometric sine function. (3)

Single Threaded: An application that is not multithreaded. See multithreaded. (5)

SoftMax Activation Function: An activation function that scales the output so the sum is one. (3)

Staring Temperature: The temperature for the first simulated annealing cycle. (8)

Supervised Training: Training where the acceptability of the output of the neural network can be calculated. (4)

Synapse: An Encog connection between two layers. (1)

Temporal Data: Data that occurs over time. (10)

Temporal Neural Network: A neural network that is designed to accept temporal data, and generally, offer a prediction. (10)

Terminal Velocity: The maximum velocity that a falling object can obtain before friction brings acceleration to zero. (8)

Thermal Neural Network: A neural network that contains a temperature; examples include the Hopfield Neural Network and the Boltzmann machine. (12)

Threshold Value: Values kept on the layers of networks. Together with the weights, these are adjusted to train the network. (2)

Training: The process of adjusting the weights and thresholds of a neural network to lower the error rate. (1, 6)

Training Set: Data that is used to train a neural network. (1)

Traveling Salesman Problem: A computer problem where a traveling salesman must find the shortest route among a number of cities. (12) (TSP)

Unsupervised Training: Training where no direction is given to the neural network as far as expected output. (4)

Update Delta: The amount that training has determined a connection weight should be updated by. (5)

Vector Length: The square root of the sum of the squares of a vector. This is a means of taking the average of the numbers in a vector. (6)

Weight Matrix: The collection of connection weights between two layers. (2)

Weighted Synapse: An Encog synapse between two layers that contains weights. This is the most common form of Encog synapse. (2)

Weightless Synapse: A weight matrix that has no weights, only connections. (2)

Window: A group of temporal data values. (10)

XML File: A file that is encoded in XML; Encog saves objects to XML files. (4)

XOR Operator: A logical operator that is only true when its two inputs do not agree. (1)

Index

A
Accuracy vi, 281-2, 306-7
Activation 86, 88-9, 95, 221, 446, 449, 458
  functions xiii, xxii, 50, 80-1, 85-91, 93-6, 98-9, 113-4, 120, 126, 137, 148, 455-7, 459-60, 463-4
    linear 91, 99, 258, 460
    logarithmic 92-3, 99
ActivationBiPolar 87, 98, 442, 448
ActivationCompetitive 98, 444
ActivationGaussian 89, 98
ActivationLinear 54, 60, 91, 98, 440-1, 444, 450-2
ActivationLOG 92, 98
ActivationSigmoid 41, 50-1, 56, 66, 85, 93-4, 98, 115, 124, 126, 134, 192, 221, 447
ActivationSIN 94-5
ActivationSoftMax 95-6, 98
ActivationTANH 96, 98, 362, 364, 367, 374, 378, 380-1, 446, 449
ADALINE neural network xvi, 439
ADALINEPattern 439
Adaptive Resonance Theory xv-i, 61, 64, 81, 385, 393-4, 396-9, 415-6, 436, 440-1, 455
AI xix, 27, 455
Annealing 249-50, 358, 443
Ant 420-1
Artificial Neural Network 43, 455-6, 461
Autoassociation 79, 81, 455

B
Backpropagation algorithm xxii, 62, 120-2, 124, 127, 130, 132-3, 136-7, 143-4, 261, 287, 330, 400, 410, 446
BAM Neural Network xvi, 61, 441-2
Basic Layer 50, 81, 455, 462
BasicInputField 156, 242, 244-5
BasicLayer 36, 41, 50-2, 54, 56, 59, 60, 66-7, 85, 115, 124, 126, 209-10, 295-6, 440-2, 446-52
BasicNetwork 50-1, 56-7, 63-4, 85, 126-7, 139-41, 192-4, 196-8, 209-10, 212-3, 215-20, 285-6, 406-11, 440-4, 446-52
BasicNeuralData 31, 38, 41, 66, 124, 127, 131, 134, 209-12, 216-8, 252, 270, 272, 327, 407
Batch processing 153-6
Benchmarking Encog 102-3, 139, 457
Best error rate 362-4
Best Matching Unit (BMU) 260-5, 271, 273-5, 455-6, 461
Better neural networks 247, 253
Binary
  file 170-1, 194, 217-8, 225
  serialization file 218
Biological Neural Network 27, 43, 456, 461
Bipolar
  input patterns 385
  numbers 79, 87, 98, 338, 346, 359, 396
BiPolarNeuralData 73-4, 79, 80,

345, 349-50, 396-7
Boltzmann Machine xvi, 61-2, 81, 337-8, 346-8, 353-5, 357, 359, 368-9, 442-3, 447, 456, 464
BoltzmannLogic 61-2, 351-2, 355, 358, 443
BoltzmannPattern 352, 357, 443
Bounding box 313, 334, 456
BufferedNeuralDataSet 170-1, 193-4

C
CalculateScore 241, 252-3, 363, 365
Classification xxi, 444
Classpath, java 424
CompetitiveTraining 257, 269-71, 273
Compiling and Executing Encog Examples xvi, 422
Configuration information 213-4, 222, 225
Connect 47, 49, 54, 56, 58, 63, 343, 457, 462
Connections vii, 32, 36, 54, 58, 120, 222, 332, 337, 339, 359-60, 376-7, 443, 458-9, 463
  encog 464
  recurrent 126, 337, 464
  trainable weighted 445
Console-mode training 175
Context 51, 59, 360-1, 446, 449, 452
ContextLayer 37, 42-3, 51-2, 59, 63, 359-60, 364, 367-9, 445-6, 448-9, 451, 456-7, 460, 462, 464
Converting training file 175, 192-3
Copy 59, 183, 185, 187-8, 420
Cores 136-7, 142, 421
Counter-Propagation network xvi, 385, 399, 400, 415, 436, 443, 456, 459, 462
CPN Neural Networks 399, 408-9, 411, 415, 436, 443, 456
CPNPattern 406, 409, 444-5
CSV File 116-7, 153, 158-9, 170, 185-6, 189-90, 193-4, 198-9, 202-3, 245, 457
  normalized 244
  raw 244
Cycles 68, 74, 76-80, 230, 234, 248, 251, 318, 329-30, 338, 340-3, 345, 368, 378, 443

D
Data
  files xxii, 172, 174, 189
  sources 147, 153-4
  target 153-5
  temporal 279, 288, 309, 457, 460, 464
DataInputStream 320
DataNormalization 153-5, 157-9, 169-71, 174, 178, 180-2, 184, 186, 188-9, 192-3, 196-8, 242, 244, 284, 288
Dataset 148, 160, 166-7, 170, 213, 288-91, 293, 308, 316-7
  encog 279, 289, 292
    image xv, 315
  specialized 308, 311
Delta values 122, 133, 138, 143, 280, 300, 455
Dimensions 163, 261-2
DirectSynapse 60
Distance formula 354-5
Download 40, 101, 172, 294, 308, 419-20, 433

Downsampling xv, 311, 313, 315-8, 325, 327-8, 332-4, 457, 463

E

Eclipse 424-5, 427
Elapsed time 175
Elman 63, 337, 360-4, 367-9
 and Jordan Neural Networks xxiii, 34, 51, 63, 337, 359, 368, 456, 464
 network 445, 461
Elman Neural Network xv-i, 81, 359-62, 364-7, 369, 435, 445, 448, 457
ElmanPattern 362, 364-5, 446, 449
Encode 150, 164, 353
Encoding 149-50, 153, 165, 201
Encog
 core JAR file 421-2
 downloading 419
 examples project 427
 object 202, 207, 213, 219-20, 224-5
 persistence files 180, 193, 195, 198, 202, 207
 platforms 207-8
Encog, API xxii
Encog, History xiii, xix
Encog, Installing xv, 40, 419
Encog, Logo 313-4
Encog, Serialization 434
Encog Workbench xiii, xxii, 27-8, 35, 43, 53, 65, 70, 98, 101-3, 105, 107, 113-7, 223-4, 419-20
Encog XML files xiv, 174, 207-8, 217-8
EncogError 321-2
EncogMemoryCollection 214
EncogPersistedCollection xiv, 178, 180, 192-3, 196, 207-9, 211-5, 225, 295, 297, 299, 301, 304, 380-1, 434
EncogPersistedObject 48, 55, 86, 213
EncogPersistence 208, 210
EncogUtility 193-5, 249, 323-4, 328, 330, 391-3
Epochs 38-9, 41, 67-9, 124, 128-9, 131-2, 134-5, 230-1, 249, 252, 281, 298-9, 363-4, 406-7, 410-3
Error 38-41, 67-9, 120, 128-32, 134-5, 137, 140-2, 151-3, 175-6, 209-12, 216-8, 281, 363-4, 410-3, 427
 gradients 120, 455, 457, 460, 462
 rate 38-9, 43, 68-9, 119, 127-8, 132, 144, 176-7, 209-11, 215, 217, 272, 274, 330, 458-9
Euclidean distance 153, 260, 274
Evaluation 75-6, 81, 129, 154, 176, 179, 187, 195, 198, 200-3, 284, 286, 290, 303-5, 344-5

F

Factor 87, 141-2, 162-3, 250, 261
FanInRandomizer 248-9, 252-3
Feedforward Neural Networks xvi, 28, 34, 62-3, 65, 104-6, 116, 222-3, 359, 361, 363-4, 368-9, 383, 385-7, 445-6
FeedforwardLogic 62-3, 124, 126, 192
FeedForwardPattern 362, 374, 378, 380-1, 447
File
 image 127
 java 196, 298
 normalized.csv 174, 193

serialized 207
universal 207
FileInputStream 320
FinalizeStructure 36, 41, 50-2, 54, 57, 59, 60, 66, 85, 126, 209-10, 295-6, 440-3, 445-8, 450, 452-3
ForestCover 172, 179, 192, 195

G
Gamma 351, 357
Gaussian function 52-3, 89, 90, 262-4, 274-5, 386-8, 390, 450, 458
GaussianFunction 392
Genetic algorithms xxiii, 143, 225, 229, 235, 241, 247-51, 253-4, 456, 458, 461
Goals xix, xx, 162, 230
Gradient 120-2, 137, 144
descents 120-1, 137-8

H
Hidden
layer
activation function 95
architecture 382
combination 378
configuration 377
count 105
RBFs 392
structures 32, 374, 382
layers 30-3, 35-6, 42-4, 52, 106-7, 194, 296, 360, 367-8, 373, 377-8, 381-3, 445-6, 450, 459-60
neurons 35, 105, 178, 359, 374, 377, 382, 387, 390, 459, 462
minimum number of 105-6
Hopfield 61, 75-6, 344
Hopfield Neural Network xvi, 27, 62-3, 69, 70, 75-6, 79-81, 338-40, 343-6, 368-70, 383, 433, 442, 447, 459, 464
HopfieldLogic 74-6, 79, 344-5, 448
HopfieldPattern 74-5, 343, 448
Hyperbolic tangent activation function 50, 81, 85, 92, 94, 96-9, 114, 148, 328, 459
Hyperthreading 136, 141-2

I
IDE 419-21, 424, 427, 429
Ideal output 37, 119-20, 125, 128, 144, 151-2, 316, 327, 459
Identities, unique 326
Image
data xxiii, 311, 332
processing 333
recognition xv, 311, 317-8, 435
ImageNeuralData 311, 315-7, 322-5, 328, 331
Images xv, xxi, xxiii, 79, 138, 308, 311-8, 322-4, 326-8, 331-3, 400, 456-7, 463
distorted 76, 340, 343
inputting 326
physical 312
Included Record 167
Index xvi, 49, 73, 167, 186-7, 190, 267-8, 321-2, 356, 467
IndexRangeSegregator 166
IndexSampleSegregator 166-7, 181-2, 185-6
Input
columns 107
data 31, 52, 116, 120, 149, 177, 199, 201, 257, 327, 410
field types 155-6, 158
fields 153-6, 159-60, 162-4, 168, 186-7, 189, 198, 202-4, 244-5, 459,

462
final ImageNeuralData 324, 331
image 323, 327, 332
layer 30-2, 36, 42-4, 62, 120, 126, 139, 221, 258, 338, 373, 394, 400, 445-6, 459
neurons 30, 34, 75, 105, 112, 126, 151, 160, 244, 258, 260, 278, 313-5, 332-3, 361
pattern 260, 274, 397, 412, 463
property 128
InputField 181, 183, 189, 202, 242, 244
InputFieldArray1D 156-7, 284, 288
InputFieldArray2D 158
InputFieldCSV 158-9, 182, 184, 186, 188-90
InputFieldNeuralDataSet 160
Instar training 400, 405, 409-10, 412, 415-6, 443-5, 456, 459
IntegerBalanceSegregator 168, 181, 183, 188
Intensity 258, 314, 332-3, 459
Internet 147
IOException 215-8, 320-4, 326-7, 329, 331
Iteration
method 38, 127, 194, 273
number 260
Iterations, previous 121, 360, 448

J
JAR Files 422-4, 427-9
Java applications 101, 114, 424
Java IDE 419, 424
Java Image 327
Java ImageIO 316
Java Network Diagram 448
Java Serialization 207, 215, 217-8, 224
for Encog 207
Java Server Mode 424
JOONE xix
Jordan Neural Network xv-i, 81, 337, 360-1, 366-70, 383, 448, 451, 460-1
JordanPattern 367

K
Kohonen neural network 257, 274-5, 460

L
Layer 48, 50, 55-6, 64, 81, 112, 115-6, 219-20, 440, 460
Layers 29-32, 35-6, 42-4, 47-60, 63-4, 69, 70, 80, 85, 98-9, 219-23, 337-9, 368-9, 374-7, 442-3, 462-5
contextual 51
instar 400, 409, 415, 443
middle 387
neuron 50
outstar 400, 409, 415
recurrent 451
self-connected 55, 369
Learning 27, 32-3, 105, 107, 235, 264-5, 399
hopfield neural network 70
neural networks 89
rate 120-2, 127, 133, 137, 143-4, 261, 264, 271-3, 275, 330, 410, 455, 460
Libraries, third-party 423
Linear layers 92
Load 101, 182, 208, 210, 212-4, 218, 293, 295, 297, 299, 304, 326-7, 332

Lunar Lander 229-30, 435

M
Magnitude 121-2, 460-1
Manhattan Update Rule xxii, 121-2, 130, 132-3, 136-7, 143-4, 434, 460, 462
ManhattanPropagation 131-2
Market data 292-4, 296, 303-5, 311
MarketDataDescription 293-4, 296, 301, 303
Matrix 222, 259-60, 268, 355-6
Memory-Based Encog Collections 214
Mode 317-8, 324-5, 329
Model 34-5, 223-4, 307
Momentum 121, 127, 143-4, 330, 460
Multiplicative normalization 161-3, 203-4, 461
Multithreaded training 137, 139, 143, 457
Multithreading xiv, 136, 140

N
Native 220-3
  file format 208
Neighborhood function 261-2, 270-1, 274-5, 458, 461
NeighborhoodGaussianMulti 269, 271
Net xix, 101, 103, 207, 225, 457
Net Encog 103
Neural
  network
    stabilized 345
    stock market 309, 379
    supervised feedforward 110, 122
    System.out.println 42, 67, 124, 134
    technology 424
    temporal 277, 307
    thermal 337-8, 370, 458-9, 464
    top 248
    traditional feedforward 416
    train xxii-i, 170, 208, 225, 231, 298-300, 329, 332, 382, 399, 458
    two-layer 394
    types 27, 31, 47, 51, 63-4, 80, 104, 109, 223, 274, 337-8, 368-9, 383, 385, 461
    untrained 230, 382
    valid 353
    visualize 27
    window-based prediction 307
  pilot xiv, 230-1, 235, 244, 247-50
Neural network
  2-layer feedforward 400
  adaptive resonance theory 61
  API xix
  applications xxiii, 42, 424, 457
  applying 308
  approach xxi
  artificial 27
  basic xix, 210, 217
  best-trained 381-2
  classes 31
  classification 96
  competitive 98, 274
  efficient 382
  fittest 253
  hybrid 385, 409, 443, 456
  layer feedforward 210, 217
  logic 63-4, 81
  multilayer 458
  normalization techniques 164

offspring 248
operations 102
output 31, 150, 200, 411
persist 101
predictive 274, 277, 308, 360, 386, 435
radial basis function xvi, 386, 450
random 248
regular feedforward 390, 456
saving Encog 224
self-connected 69, 368, 414
short-term memory 464
simple recurrent 29, 94, 97, 368, 370, 445, 456-7, 460, 464
single layer 32, 439
recurrent 62, 447
Neural Network Solution Programs xx
NeuralData 30-1, 39, 42, 48-9, 55, 67-8, 128, 155, 196-7, 199, 200, 243, 246, 270, 272-3, 290
NeuralDataPair 37-9, 41-2, 66-8, 124, 127-8, 134, 140-1, 159, 170, 209-12, 216-8, 252-3, 298-9, 380-1, 406-10
NeuralGeneticAlgorithm 248, 251-3
NeuralLogic 47-8, 60
NeuralSimulatedAnnealing 250, 361, 363, 365
Neuron
count 374-7
layer number 222
number 356
Neurons 30, 35-6, 47, 57-9, 88-9, 151-3, 221-2, 259-61, 338-9, 352-3, 355-60, 373-8, 382-3, 394-7, 455-7
groups of 88, 460
important 376
ineffective 377
matching 274
neighboring 274
non-winning 88
self-connected 356
significant 377, 382-3
single 34, 36, 69, 338, 353, 360
source layer 54
target 355
weakest 376, 463
winning 274, 331, 397, 410, 456
Nominal values 148, 150, 164, 204, 461
Normalization xiv, 148-9, 151, 153-4, 159-60, 163, 166, 168, 170-2, 180, 187-8, 201-4, 244-5, 288, 461-2
input fields 155
object 156, 160, 180, 188-9, 191, 193, 198-200, 244-6
process 170
types 160, 201-2
value 461
Normalization, Targets 159, 169-70, 186, 204, 461
NormalizeData 201-2
NormalizedSunspots 284, 288
NormalizeInput 405, 408
void 406
Norm.buildForNetworkInput 155, 197, 199
Norm.getInputFields 197-8
Norm.getNetworkInputLayerSize 193-4
Norm.getNetworkOutputLayerSize 193-4
NullStatusReportable 157-9, 284, 288
NumberFormat 237, 285, 290

O
One-dimensional array 156-7, 169, 203, 268, 288
One-to-One Synapse 82, 369, 462
Output
 fields 153-6, 158, 160-2, 165, 169, 186-7, 203-4, 244-5, 462
 grid 258-9
 layer 29-32, 34, 36, 42-4, 52, 56-7, 62, 65, 258, 338-9, 375, 387, 445-6, 448, 459-60
  neuron 32
 neuron
  index number 326
  number 257
 neurons 32, 34, 37, 105, 151-3, 160, 257-8, 260-2, 267-8, 277-8, 315, 326-7, 333, 338-9, 410-1
  single 34, 126, 246, 360, 392
  winning 394
 value, ideal 151, 153
OutputEquilateral 165, 181, 197-8
OutputField 157-64, 182, 184-6, 188, 190-1, 242, 245, 284, 288
Outstar training 410-2, 415-6, 443-5, 456, 462
OutstarNeurons 405, 444

P
Past window size 278-9, 293, 295, 308, 369
Pattern recognition xxi, 61, 274, 346, 394, 414, 439-40, 455, 459
Patterns xxi, xxiii, 61, 63, 69-76, 79, 80, 338-40, 343-6, 378, 394-400, 405-9, 411-2, 439, 441-4, 446-52
Peak 52, 263-4, 275, 387, 389, 392-3, 450, 463
Perceptron, multilayer 126
Pixels 79, 314-5, 333
Platform 219-20, 224-5, 419
Population 248-9
Propagation training xiv, xxii, 87-90, 92-3, 97-8, 119-23, 127, 129-31, 135-9, 143, 145, 225, 247, 455, 462-3
Properties vii, 30, 47, 63-4, 80-1, 98, 104, 107, 116, 128, 222-3, 225, 265, 427
PruneSelective class 374
Pruning xxiii, 32, 106, 368, 374-83, 436, 459, 462

R
Radial Basis Function Networks xv, 27, 52, 82, 90, 385-6, 415-6, 436, 450, 457, 463
RadialBasisFunctionLayer 52-4, 60, 387, 391-2, 450
RadialBasisPattern 387, 391-2, 450
Radius 271-3, 275
Random values 36, 40, 126, 259, 329, 392-3, 409
Randomness 62, 250, 254, 442
RangeRandomizer 270, 272
Raw data 155, 171, 176, 202, 245-6, 280, 287
RBF 52, 386-7, 390, 393, 415, 450-1, 463
 functions 52, 390, 393
 hidden layer 415
 neurons 392
 pattern 392
RBF-based layer, special 387
RBF Layer in Encog Workbench 53

RBF layers 54
hidden 392
RBF Network in Encog Workbench 386
RBF Neural Network 54, 386-8, 390, 392-3, 416, 436, 450
Recurrent Neural Networks xv, xxiii, 28, 34, 44, 51, 61-3, 251, 277, 330, 332, 337, 345-7, 359, 367-9
Rerunning 363-4
Resilient propagation xxii, 38, 41, 44, 62, 66-8, 121-2, 133-5, 140, 143, 211, 216-7, 298-300, 365-6, 462-3
Resolution 313, 315, 332, 457
Resolving Path Errors 427
Resources 211, 213-4, 220, 297
RGB 315, 317-8, 322, 325, 333-4, 463
RGBDownsample 315, 322, 325

S
Script file 317, 325, 420
Segregators 154-5, 166-8, 182-3, 186, 188, 202-4, 463
Selective pruning xv, xxiii, 374, 377, 382-3, 463
Self-Connected Layer 55, 82, 370, 459, 463
Self Organizing Map xiv, 63, 257-9, 265-8, 274-5, 383, 394, 398, 409, 435, 443, 451-2, 455-6, 460-1, 463
Sequence number 280
SerializeObject 218
Short-term memory 37, 42-3, 47, 51
Simulated annealing xxiii, 62, 143, 225, 229, 235, 241, 247, 249-51, 253-5, 346, 365, 448, 463-4
SOMPattern 270, 452
Stabilized pattern 338
Stochastic 346
Strategies 235, 318, 329-30, 365-6, 369
Supervised training xiv, xxiii, 62, 116-7, 119, 143-4, 229-31, 233, 235, 237, 239, 241, 247, 253-5, 315-6
Synapse 48, 54-6, 58, 442
types 43, 47-8, 54, 57, 64, 67, 223
Synapses xiii, xxiii, 29, 32, 42-4, 47-9, 54-60, 64, 75, 80, 104, 106-7, 112, 116, 222
direct 59, 60, 81, 457
self-connected 55
weighted 57-8, 82, 465

T
Tags xiii, xxiii, 63-4, 75, 98, 104, 220, 223
Temperature 62, 250-1, 253-5, 338, 346, 352, 358, 369, 442-3, 457, 464
Temporal
networks 280, 307
neural networks 277-8, 308-9, 360, 368, 458, 464
TemporalDataDescription 279-80, 285, 289, 308-9
TemporalDataSet 296, 307-8
TemporalError 279
TemporalNeuralDataSet 279, 281, 285, 287, 289, 292
TemporalPoint 280, 285, 289, 292, 308
Terminal velocity 229, 232, 239, 255, 464
Text files vii, 207, 214, 225
Threads 136-8, 269
optimal number of 137, 143

Threshold values 36-7, 42-3, 47, 49-51, 82, 85, 98, 112, 119-22, 143, 221, 258, 275, 369, 376
Thresholds 42, 67, 109, 112, 120, 122, 126, 130, 132, 137-8, 221, 223, 329-30, 357, 377
Training
 algorithms 116, 119-20, 123, 126, 133, 137, 143, 257, 318, 459
 data 94, 109, 114, 116, 120, 127-9, 136-8, 167, 174, 218-20, 223-4, 278, 293-4, 296-8, 380-1
 set xiii, 38, 44, 107-9, 119-20, 127-8, 211-3, 229, 247-8, 251, 253-4, 271-2, 325-6, 377-8, 381
 sets, larger 137, 139
 strategies 329-30, 365, 459
  encog hybrid 369
 techniques xxiii, 38, 143, 231, 359, 365-6, 369, 462, 464
Training Set Score Class xiv, 251
Training XOR network 209-10, 215, 217
TSP 346-8, 370, 465
TSP Neural Network 353

U
Uniform distribution 166-7
Unsupervised training xxiii, 108, 116-7, 119, 254, 274, 315-6, 394, 409, 415, 456, 465
Unsupervised Training Methods xiv, xxiii, 63, 257, 259, 261, 263, 265, 267, 269, 271, 273, 275, 386, 388

V
Variables 85, 149, 153, 238, 260, 263
Vector 260, 458, 461, 465
 length 161-2, 204, 465
Visualize 354-5, 365

W
Weight
 changes 121
 deltas 460, 462
 matrix 32, 47, 54-5, 57, 82, 120-1, 174, 222, 259, 265-6, 268, 365, 455, 460, 465
 vector 260
Weighted synapse object 57
WeightedSynapse 49, 52, 56-8, 67, 222, 441-2
WeightlessSynapse 58, 82, 222, 465
Weights 36-8, 42, 58, 121-2, 126, 137-8, 221-3, 249-50, 260, 267, 329-30, 351, 355-8, 376-7, 465
 connection 460, 465
Window size 277-9, 289, 293, 295, 308, 360
Windows 104, 109-10, 112, 277-9, 282, 292-3, 295, 307-9, 419-20, 458, 465
Winners 88-9, 258, 265, 324, 331, 397

X
XML File 117, 207, 218, 221, 224, 457, 465
XOR xxii, 32-8, 41-2, 64-6, 105, 109, 122-7, 134, 208-12, 216-8, 220, 251-2, 360, 390, 392-3

Y
YahooFinanceLoader 293-5, 300, 303

Year-WINDOW 286, 290-1

Z

Z-axis 162-3, 202-4, 461
Zip codes 148