INTRODUCTION

to the MATH *of*

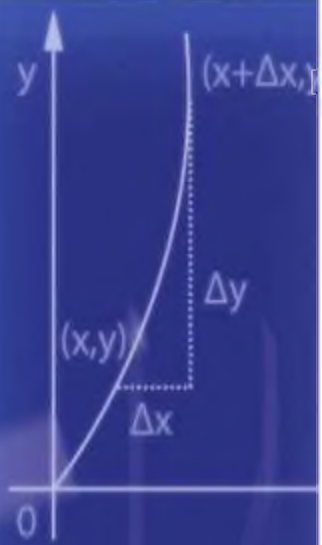
NEURAL NETWORKS

JEFF HEATON

$y=x^2$

$$8=\sum_{i=1}^{10} 2i$$

$$\frac{Bwt}{n}$$



$B=0.7h^{1/2}$

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

$= 0.8723+(1.0*0.5*(0.5-0.8723)) = 0.68615$

Heaton Research

Title	Introduction to the Math of Neural Networks
Author	Jeff Heaton
Published	May 01, 2012
Copyright	Copyright 2012 by Heaton Research, Inc., All Rights Reserved.
File Created	Thu May 17 13:06:16 CDT 2012
ISBN	978-1475190878
Price	9.99 USD

Publisher: Heaton Research, Inc
Introduction to the Math of Neural Networks
May, 2012
Author: Jeff Heaton
Editor: WordsRU.com
Cover Art: Carrie Spear
ISBN: 978-1475190878

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.

Introduction

- Math Needed for Neural Networks
- Prerequisites
- Other Resources
- Structure of this Book

If you have read other books I have written, you will know that I try to shield the reader from the mathematics behind AI. Often, you do not need to know the exact math that is used to train a neural network or perform a cluster operation. You simply want the result.

This results-based approach is very much the focus of the Encog project. Encog is an advanced machine learning framework that allows you to perform many advanced operations, such as neural networks, genetic algorithms, support vector machines, simulated annealing and other machine learning methods. Encog allows you to use these advanced techniques without needing to know what is happening behind the scenes.

However, sometimes you really do want to know what is going on behind the scenes. You do want to know the math that is involved. In this book, you will learn what happens, behind the scenes, with a neural network. You will also be exposed to the math.

There are already many neural network books that at first glance appear as a math text. This is not what I seek to produce here. There are already several very good books that achieve a pure mathematical introduction to neural networks. My goal is to produce a mathematically-based neural network book that targets someone who has perhaps only college-level algebra and computer programming background. These are the only two prerequisites for understanding this book, aside from one more that I will mention later in this introduction.

Neural networks overlap several bodies of mathematics. Neural network goals, such as classification, regression and clustering, come from statistics. The gradient descent that goes into backpropagation, along with other training methods, requires knowledge of Calculus. Advanced training, such as Levenberg Marquardt, require both Calculus and Matrix Mathematics.

To read nearly any academic-level neural network or machine learning targeted book, you will need some knowledge of Algebra, Calculus, Statistics and Matrix Mathematics. However, the reality is that you need only a relatively small amount of knowledge from each of these areas. The goal of this book is to teach you enough math to understand neural networks and their training. You

will learn exactly how a neural network functions, and when you are finished this book, you should be able to implement your own in any computer language you are familiar with.

Since knowledge of some areas of mathematics is needed, I will provide an introductory-level tutorial on the math. I only assume that you know basic algebra to start out with. This book will discuss such mathematical concepts as derivatives, partial derivatives, matrix transformation, gradient descent and more.

If you have not done this sort of math in a while, I plan for this book to be a good refresher. If you have never done this sort of math, then this book could serve as a good introduction. If you are very familiar with math, you can still learn neural networks from this book. However, you may want to skip some of the sections that cover basic material.

This book is not about Encog, nor is it about how to program in any particular programming language. I assume that you will likely apply these principles to programming languages. If you want examples of how I apply the principles in this book, you can learn more about Encog. This book is really more about the algorithms and mathematics behind neural networks.

I did say there was one other prerequisite to understanding this book, other than basic algebra and programming knowledge in any language. That final prerequisite is knowledge of what a neural network is and how it is used. If you do not yet know how to use a neural network, you may want to start with my article, 'A Non-Mathematical Introduction to Using Neural Networks', which you can find at

http://www.heatonresearch.com/content/non-mathematical-introduction-using-neural-networks.

The above article provides a brief crash course on what neural networks are. You may also want to look at some of the Encog examples. You can find more information about Encog at the following URL:

http://www.heatonresearch.com/encog/

If neural networks are cars, then this book is a mechanics guide. If I am going to teach you to repair and build cars, I make two basic assumptions, in order of importance. The first is that you've actually seen a car, and know what one is used for. The second assumption is that you know how to drive a car. If neither of these is true, then why do you care about learning the internals of how a car works? The same applies to neural networks.

Other Resources

There are many other resources on the internet that will be very useful as you read through this book. This section will provide you with an overview of some of these resources.

The first is the Khan Academy. This is a collection of YouTube videos that demonstrate many areas of mathematics. If you need additional review on any mathematical concept in this book, there is most likely a video on the Khan Academy that covers it.

http://www.khanacademy.org/

Second is the Neural Network FAQ. This text-only resource has a great deal of information on neural networks.

http://www.faqs.org/faqs/ai-faq/neural-nets/

The Encog wiki has a fair amount of general information on machine learning. This information is not necessarily tied to Encog. There are articles in the Encog wiki that will be helpful as you complete this book.

http://www.heatonresearch.com/wiki/Main_Page

Finally, the Encog forums are a place where AI and neural networks can be discussed. These forums are fairly active and you will likely receive an answer from myself or from one of the community members at the forum.

http://www.heatonresearch.com/forum

These resources should be helpful to you as you progress through this book.

Structure of this Book

The first chapter, "Neural Network Activation", shows how the output from a neural network is calculated. Before you can find out how to train and evaluate a neural network, you must understand how a neural network produces its output.

Chapter 2, "Error Calculation", demonstrates how to evaluate the output from a neural network. Neural networks begin with random weights. Training adjusts these weights to produce meaningful output.

Chapter 3, "Understanding Derivatives", focuses on a very important Calculus topic. Derivatives, and partial derivatives, are used by several neural network training methods. This chapter will introduce you to those aspects of derivatives that are needed for this book.

Chapter 4, "Training with Backpropagation", shows you how to apply knowledge from Chapter 3 towards training a neural network. Backpropagation is one of the oldest training techniques for neural networks. There are newer – and much superior – training methods available. However, understanding backpropagation provides a very important foundation for resilient propagation (RPROP), quick propagation (QPROP) and the Levenberg Marquardt Algorithm (LMA).

Chapter 5, "Faster Training with RPROP", introduces resilient propagation, which builds upon backpropagation to provide much quicker training times.

Chapter 6, "Weight Initialization", shows how neural networks are given their initial random weights. Some sets of random weights perform better than others. This chapter looks at several, less than random, weight initialization methods.

Chapter 7, "LMA Training", introduces the Levenberg Marquardt Algorithm. LMA is the most mathematically intense training method in this book. LMA can sometimes offer very rapid training for a neural network.

Chapter 8, "Self Organizing Maps", shows how to create a clustering neural network. The Self Organizing Map (SOM) can be used to group data. The structure of the SOM is similar to the feedforward neural networks seen in this book.

Chapter 9, "Normalization", shows how numbers are normalized for neural networks. Neural networks typically require that input and output numbers be in the range of 0 to 1, or -1 to 1. This chapter shows how to transform numbers into that range.

Chapter 1: Neural Network Activation

- Summation
- Calculating Activation
- Activation Functions
- Bias Neurons

In this chapter, you will find out how to calculate the output for a feedforward neural network. Most neural networks are in some way based on the feedforward neural network. Learning how this simple neural network is calculated will form the foundation for understanding training, as well as other more complex features of neural networks.

Several mathematical terms will be introduced in this chapter. You will be shown summation notation and simple mathematical formula notation. We will begin with a review of the summation operator.

Understanding the Summation Operator

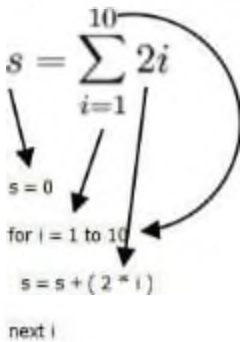
In this section, we will take a quick look at the summation operator. The summation operator, represented by the capital Greek letter sigma, can be seen in Equation 1.1.

Equation 1.1: The Summation Operator

$$s = \sum_{i=1}^{10} 2i$$

The above equation is a summation. If you are unfamiliar with sigma notation, it is essentially the same thing as a programming **for** loop. Figure 1.1 shows Equation 1.1 reduced to pseudocode.

Figure 1.1: Summation Operator to Code

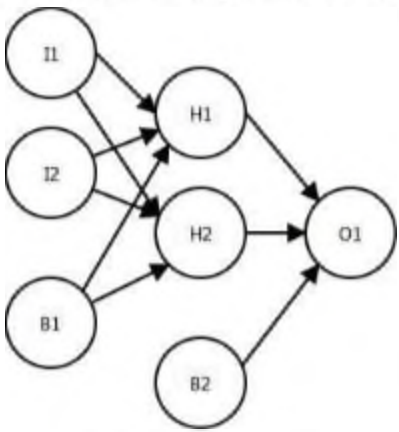


As you can see, the summation operator is very similar to a **for** loop. The information just below the sigma symbol specifies the stating value and the indexing variable. The information above the sigma specifies the limit of the loop. The information to the right of sigma specifies the value that is being summed.

Calculating a Neural Network

We will begin by looking at how a neural network calculates its output. You should already know the structure of a neural network from the resources included in this book's introduction. Consider a neural network such as the one in Figure 1.2.

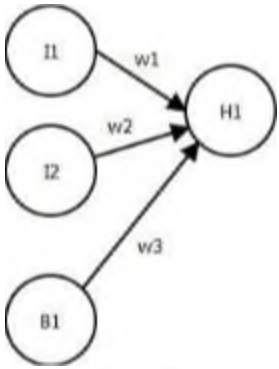
Figure 1.2: A Simple Neural Network



This neural network has one output neuron. As a result, it will have one output value. To calculate the value of this output neuron (**O1**), we must calculate the activation for each of the inputs into **O1**. The inputs that feed into **O1** are **H1**, **H2** and **B2**. The activation for **B2** is simply 1.0, because it is a bias neuron. However, **H1** and **H2** must be calculated independently. To calculate **H1** and **H2**, the activations of **I1**, **I2** and **B1** must be considered. Though **H1** and **H2** share the same inputs, they will not calculate to the same activation. This is because they have different weights. In the above diagram, the weights are represented by lines.

First, we must find out how one activation calculation is done. This same activation calculation can then be applied to the other activation calculations. We will examine how **H1** is calculated. Figure 1.3 shows only the inputs to **H1**.

Figure 1.3: Calculating H1's Activation



We will now examine how to calculate H1. This relatively simple equation is shown in Equation 1.2.

Equation 1.2: Calculate H1

$$h_1 = A(\sum_{c=1}^{n}(i_c * w_c))$$

To understand Equation 1.2, we can first look at the variables that go into it. For the above equation we have three input values, described by the variable **i**. The three input values are input values of **I1**, **I2** and **B1**. **I1** and **I2** are simply the input values with which the neural network was provided to compute the output. **B1** is always 1, because it is the bias neuron.

There are also three weight values considered: **w1**, **w2** and **w3**. These are the weighted connections between **H1** and the previous layer. Therefore, the variables to this equation are:

```
i[1] = first input value to the neural network
i[2] = second input value to neural network
i[3] = 1
w[1] = weight from I1 to H1
w[2] = weight from I2 to H1
w[3] = weight from B1 to H1
n = 3, the number of connections
```

Though the bias neuron is not really part of the input array, a value of one is always placed into the input array for the bias neuron. Treating the bias as a forward-only neuron makes the calculation much easier.

To understand Equation 1.2, we will consider it as pseudocode.

```
double w[3] // the weights
double i[3] // the input values
double sum = 0; // the sum
// perform the summation (sigma)
```

```
for c = 0 to 2
  sum = sum + ( w[c] * i[c] )
next
// apply the activation function
sum = A(sum)
```

Here, we sum up each of the inputs times its respective weight. Finally, this sum is passed to an activation function. Activation functions are a very important concept in neural network programming. In the next section, we will examine activation functions.

Activation Functions

Activation functions are very commonly used in neural networks. They serve several important functions for a neural network. The primary reason to use an activation function is to introduce non-linearity to the neural network. Without this non-linearity, a neural network could do little to learn non-linear functions. The output that we expect neural networks to learn is rarely linear.

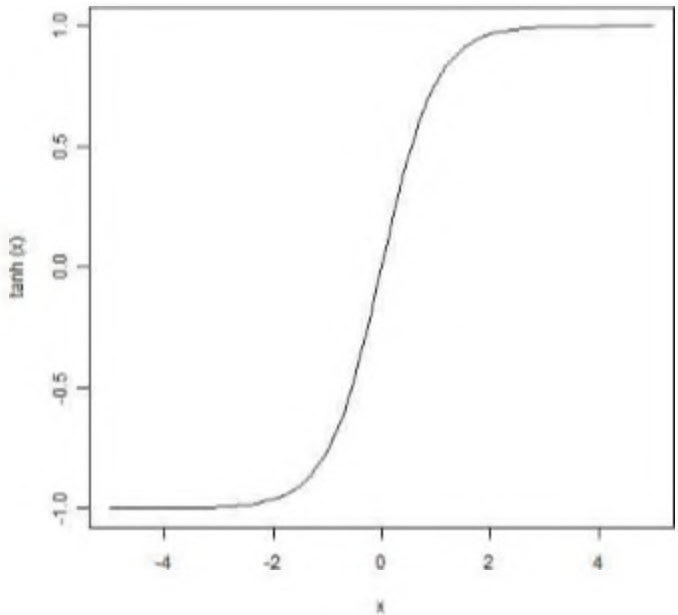
The two most common activation functions are the sigmoid and hyperbolic tangent activation function. The hyperbolic tangent activation function is the more common of these two, as it has a number range from -1 to 1, compared to the sigmoid function which ranges only from 0 to 1.

Equation 1.3: The Hyperbolic Tangent Function

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

The hyperbolic tangent function is actually a trigonometric function. However, our use for it has nothing to do with trigonometry. This function was chosen for the shape of its graph. You can see a graph of the hyperbolic tangent function in Figure 1.4.

Figure 1.4: The Hyperbolic Tangent Function



Notice that the range is from -1 to 1. This allows it to accept a much wider range of numbers. Also notice how values beyond -1 to 1 are quickly scaled. This provides a consistent range of numbers for the network.

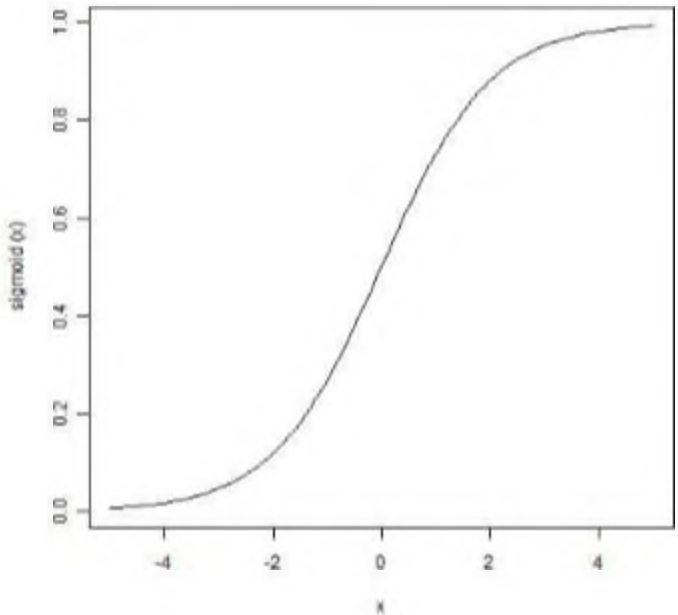
Now we will look at the sigmoid function. You can see this in Equation 1.4.

Equation 1.4: The Sigmoid Function

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

The sigmoid function is also called the logistic function. Typically it does not perform as well as the hyperbolic tangent function. However, if the values in the training data are all positive, it can perform well. The graph for the sigmoid function is shown in Figure 1.5.

Figure 1.5: The Sigmoid Function

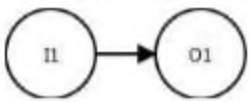


As you can see, it scales numbers to 1.0. It also has a range that only includes positive numbers. It is less general purpose than hyperbolic tangent, but it can be useful. The sigmoid function outperforms the hyperbolic tangent function.

Bias Neurons

You may be wondering why bias values are even needed. The answer is that bias values allow a neural network to output a value of zero even when the input is near one. Adding a bias allows the output of the activation function to be shifted to the left or right on the x-axis. To understand this, consider a simple neural network where a single input neuron **I1** is directly connected to an output neuron **O1**. The network shown in Figure 1.6 has no bias.

Figure 1.6: A Bias-less Connection



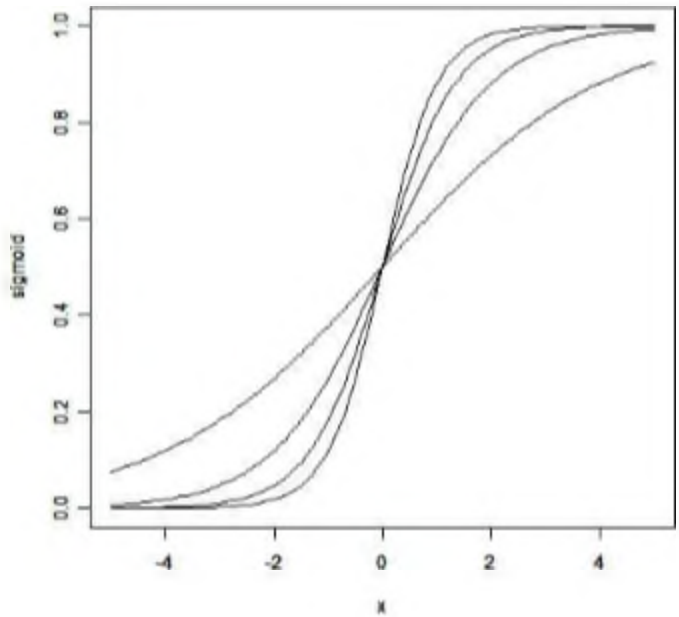
This network's output is computed by multiplying the input (**x**) by the weight (**w**). The result is then passed through an activation function. In this case, we are using the sigmoid activation function.

Consider the output of the sigmoid function for the following four weights.

```
sigmoid(0.5*x)
sigmoid(1.0*x)
sigmoid(1.5*x)
sigmoid(2.0*x)
```

Given the above weights, the output of the sigmoid will be as seen in Figure 1.7.

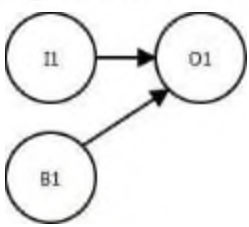
Figure 1.7: Adjusting Weights



Changing the weight w alters the "steepness" of the sigmoid function. This allows the neural network to learn patterns. However, what if you wanted the network to output 0 when x is a value other than 0, such as 3? Simply changing the steepness of the sigmoid will not accomplish this. You must be able to shift the entire curve to the right.

That is the purpose of bias. Adding a bias neuron causes the neural network to appear as in Figure 1.8.

Figure 1.8: A Biased Connection



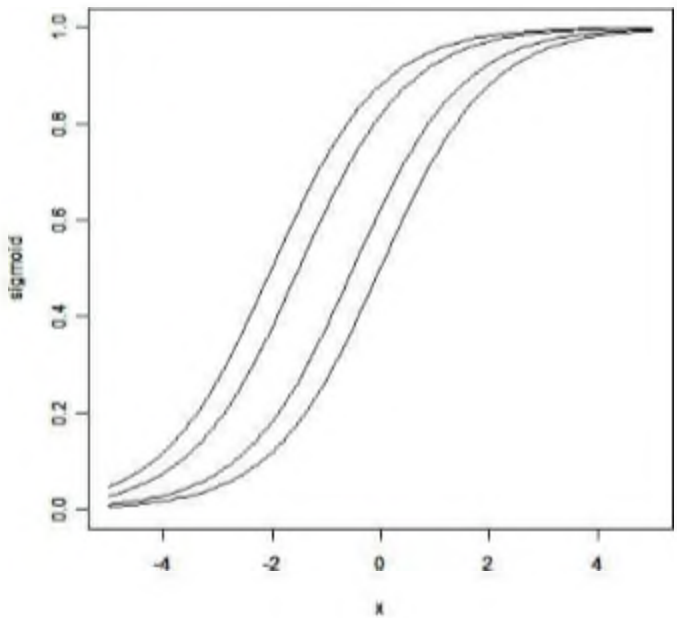
Now we can calculate with the bias neuron present. We will calculate for several bias weights.

```
sigmoid(1*x + 1*1)
sigmoid(1*x + 0.5*1)
```

```
sigmoid(1*x + 1.5*1)
sigmoid(1*x + 2*1)
```

This produces the following plot, seen in Figure 1.9.

Figure 1.9: Adjusting Bias



As you can see, the entire curve now shifts.

Chapter Summary

This chapter demonstrated how a feedforward neural network calculates output. The output of a neural network is determined by calculating each successive layer after the input layer. The final output of the neural network eventually reaches the output layer.

Neural networks make use of activation functions. An activation function provides non-linearity to the neural network. Because most of the data that a neural network seeks to learn is non-linear, the activation functions must be non-linear. An activation function is applied after the weights and activations have been multiplied.

Most neural networks have bias neurons. Bias is an important concept for neural networks. Bias neurons are added to every non-output layer of the neural network. Bias neurons are different than ordinary neurons in two very important ways. Firstly, the output from a bias neuron is always one. Secondly, a bias neuron has no inbound connections. The constant value of one allows the layer to respond with non-zero values even when the input to the layer is zero. This can be very important for certain data sets.

The neural networks will output values determined by the weights of the connections. These weights are usually set to random initial values. Training is the process in which these random weights are adjusted to produce meaningful results. We need a way for the neural network to measure the effectiveness of the neural network. This measure is called error calculation. Error calculation is discussed in the next chapter.

Chapter 2: Error Calculation Methods

- Understanding Error Calculation
- The Error Function
- Error Calculation Methods
- How the Error is Used

In this chapter, we will find out how to calculate errors for a neural network. When performing supervised training, a neural network's actual output must be compared against the ideal output specified in the training data. The difference between actual and ideal output is the error of the neural network.

Error calculation occurs at two levels. First, there is the local error. This is the difference between the actual output of one individual neuron and the ideal output that was expected. The local error is calculated using an error function.

The local errors are aggregated together to form a global error. The global error is the measurement of how well a neural network performs to the entire training set. There are several different means by which a global error can be calculated. The global error calculation methods discussed in this chapter are listed below.

- Sum of Squares Error (ESS)
- Mean Square Error (MSE)
- Root Mean Square (RMS)

Usually, you will use MSE. MSE is the most common means of calculating errors for a neural network. Later in the book, we will look at when to use ESS. The Levenberg Marquardt Algorithm (LMA), which will be covered in Chapter 8, requires ESS. Lastly, RMS can be useful in certain situations. RMS can be useful in electronics and signal processing.

The Error Function

We will start by looking at the local error. The local error comes from the error function. The error function is fed the actual and ideal outputs for a single output neuron. The error function then produces a number that represents the error of that output neuron. Training methods will seek to minimize this error.

This book will cover two error functions. The first is the standard linear error function, which is the most commonly used function. The second is the arctangent error function that is introduced by the Quick Propagation training method. Arctangent error functions and Quick Propagation will be discussed in Chapter 4, "Back Propagation". This chapter will focus on the standard linear error function. The formula for the linear error function can be seen in Equation 2.1.

Equation 2.1: The Linear Error Function

$$E = (i - a)$$

The linear error function is very simple. The error is the difference between the ideal (**i**) and actual (**a**) outputs from the neural network. The only requirement of the error function is that it produce an error that you would like to minimize.

For an example of this, consider a neural network output neuron that produced 0.9 when it should have produced 0.8. The error for this neural network would be the difference between 0.8 and 0.9, which is -0.1.

In some cases, you may not provide an ideal output to the neural network and still use supervised training. In this case, you would write an error function that somehow evaluates the output of the neural network for the given input. This evaluation error function would need to assign some sort of a score to the neural network. A higher number would indicate less desirable output, while a lower number would indicate more desirable output. The training process would attempt to minimize this score.

Calculating Global Error

Now that we have found out how to calculate the local error, we will move on to global error. MSE error calculation is the most common, so we will begin with that. You can see the equation that is used to calculate MSE in Equation 2.2.

Equation 2.2: MSE Error Calculation

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} E^2$$

As you can see, the above equation makes use of the local error (E) that we defined in the last section. Each local error is squared and summed. The resulting sum is then divided by the total number of cases. In this way, the MSE error is similar to a traditional average, except that each local error is squared. The squaring negates the effect of some errors being positive and others being negative. This is because a positive number squared is a positive number, just as a negative number squared is also a positive number. If you are unfamiliar with the summation operator, shown as a capital Greek letter sigma, refer to Chapter 1.

The MSE error is typically written as a percentage. The goal is to decrease this error percentage as training progresses. To see how this is used, consider the following program output.

```
Beginning training...
Iteration #1 Error:51.023786% Target Error: 1.000000%
Iteration #2 Error:49.659291% Target Error: 1.000000%
Iteration #3 Error:43.140471% Target Error: 1.000000%
Iteration #4 Error:29.820891% Target Error: 1.000000%
Iteration #5 Error:29.457086% Target Error: 1.000000%
Iteration #6 Error:19.421585% Target Error: 1.000000%
Iteration #7 Error:2.160925% Target Error: 1.000000%
Iteration #8 Error:0.432104% Target Error: 1.000000%
Input=0.0000,0.0000, Actual=0.0091, Ideal=0.0000
Input=1.0000,0.0000, Actual=0.9793, Ideal=1.0000
Input=0.0000,1.0000, Actual=0.9472, Ideal=1.0000
Input=1.0000,1.0000, Actual=0.0731, Ideal=0.0000
Machine Learning Type: feedforward
Machine Learning Architecture: ?:B->SIGMOID->4:B->SIGMOID->?
Training Method: lma
Training Args:
```

The above shows a program learning the XOR operator. Notice how the MSE error drops in each iteration? Finally, by iteration eight the error is below one percent, and training stops.

Other Error Calculation Methods

Though MSE is the most common method of calculating global error, it is not the only method. In this section, we will look at two other global error calculation methods.

Sum of Squares Error

The sum of squares method (ESS) uses a similar formula to the MSE error method. However, ESS does not divide by the number of elements. As a result, the ESS is not a percent. It is simply a number that is larger depending on how severe the error is. Equation 2.3 shows the MSE error formula.

Equation 2.3: Sum of Squares Error

$$\text{ESS} = \frac{1}{2}\sum_{p} E^2$$

As you can see above, the sum is not divided by the number of elements. Rather, the sum is simply divided in half. This results in an error that is not a percent, but instead a total of the errors. Squaring the errors eliminates the effect of positive and negative errors.

Some training methods require that you use ESS. The Levenberg Marquardt Algorithm (LMA) requires that the error calculation method be ESS. LMA will be covered in Chapter 7, "LMA Training".

Root Mean Square Error

The Root Mean Square (RMS) error method is very similar to the MSE method previously discussed. The primary difference is that the square root of the sum is taken. You can see the RMS formula in Equation 2.4.

Equation 2.4: Root Mean Square Error

$$\text{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} E^2}$$

Root mean square error will always be higher than MSE. The following output shows the calculated error for all three error calculation methods. All three cases use the same actual and ideal values.

```
Trying from -1.00 to 1.00
Actual:[-0.36,0.07,0.55,0.05,-0.37,0.34,-0.72,-0.10,-0.41,-
0.32]
```

```
Ideal:[-0.37,0.06,0.51,0.06,-0.36,0.35,-0.67,-0.09,-0.43,-0.33]
Error (ESS): 0.00312453
Error (MSE): 0.062491%
Error (RMS): 2.499810%
```

RMS is not used very often for neural network error calculation. RMS was originally created in the field of electrical engineering. I myself have not used RMS a great deal. Many research papers involving RMS show it being used for waveform analysis.

Chapter Summary

Neural networks start with random values for weights. These networks are then trained until a set of weights is found that provides output from the neural network that closely matches the ideal values from the training data. For training to progress, a means is needed to evaluate the degree to which the actual output from the neural network matches the ideal output expected of the neural network.

This chapter began by introducing the concepts of local and global error. Local error is the error used to measure the difference between the actual and ideal output of an individual output neuron. This error is calculated using an error function. Error functions are only used to calculate local error.

Global error is the total error of the neural network across all output neurons and training set elements. Three different techniques were presented in this chapter for the calculation of global error. Mean Square Error (MSE) is the most commonly used technique. Sum of Squares Error (ESS) is used by some training methods to calculate error. Root Mean Square (RMS) can be used to calculate the error for certain applications. RMS was created for the field of electrical engineering for waveform analysis.

The next chapter will introduce a mathematical concept known as derivatives. Derivatives come from Calculus and will be used to analyze the error functions and adjust the weights to minimize this error. In this book, we will learn about several propagation training techniques. All propagation training techniques use derivatives to calculate update values for the weights of the neural network.

Chapter 3: Derivatives

- Slope of a Line
- What is a Derivative
- Partial Derivatives
- Chain Rule

In this chapter, we will look at the mathematical concept of a derivative. Derivatives are used in many aspects of neural network training. The next few chapters will focus on training a neural network, and a basic understanding of derivatives will be useful to help you properly understand them.

The concept of a derivative is central to an understanding of Calculus. The topic of derivatives is very large and could easily consume several chapters. I am only going to explain those aspects of differentiation that are important to the understanding of neural network training. If you are already familiar with differentiation you can safely skim, or even skip, this chapter.

Calculating the Slope of a Line

The slope of a line is a numerical quality of a line that tells you the direction and steepness of a line. In this section, we will see how to calculate the slope of a straight line. In the next section, we will find out how to calculate the slope of a curved line at a single point.

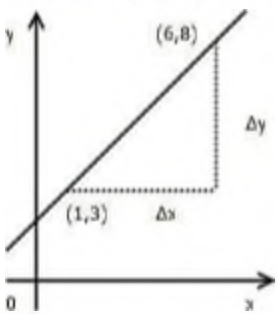
The slope of a line is defined as the "rise" over the "run", or the change in **y** over the change in **x**. The slope of a line can be written in the form of Equation 3.1.

Equation 3.1: The Slope of a Straight Line

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

This can be visualized graphically as in Figure 3.1.

Figure 3.1: Slope of a Line

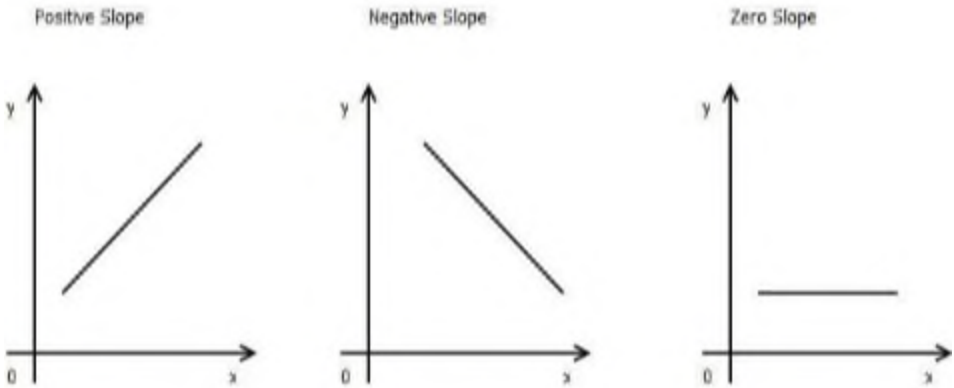


We could easily calculate the slope of the above line using Equation 3.1. Filling in the numbers for the two points we have on the line produces the following:

```
(8-3) / (6-1) = 1
```

The slope of this line is one. This is a positive slope. When a line has a positive slope, it goes up left to right. When a line has a negative slope, it goes down left to right. When a line is horizontal, the slope is 0, and when the line is vertical, the slope is undefined. Figure 3.2 shows several slopes for comparison.

Figure 3.2: Several Slopes



A straight line, such as the ones seen above, can be written in slope-intercept form. Equation 3.2 shows the slope intercept form of an equation.

Equation 3.2: Slope Intercept Form

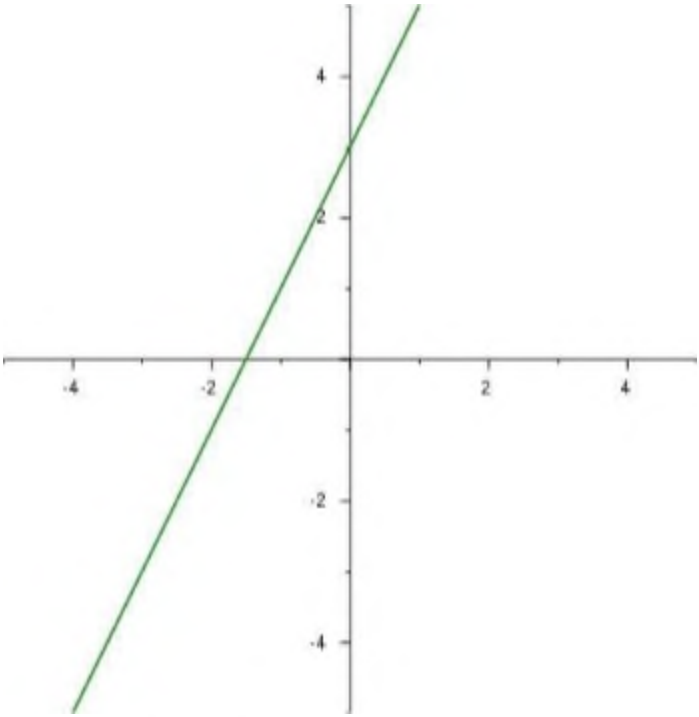
$$y = mx + b$$

Where **m** is the slope of the line and **b** is the y-intercept, which is the y-coordinate of the point where the line crosses the y axis. To see this in action, consider the chart of the following equation:

```
f(x) = 2x + 3
```

This equation can be seen graphically in Figure 3.3.

Figure 3.3: The Graph of 2x+3



As you can see from the above diagram, the line intercepts the y-axis at 3.

What is a Derivative?

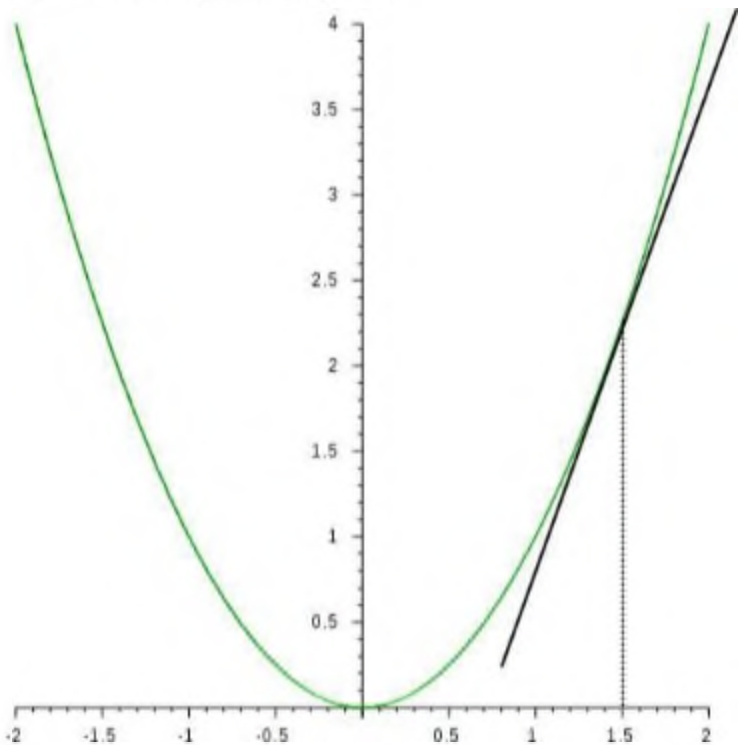
In the last section, we saw that we can easily calculate the slope of any straight line. But we will very rarely work with a simple straight line. Usually, we are faced with the sort of curves that we saw in the last chapter when we examined the activation functions of neural networks. A derivative allows us to take the derivative of a line at one point. Consider this simple equation:

Equation 3.3: X Squared

$$y = x^2$$

You can see equation 3.3 graphed in Figure 3.4.

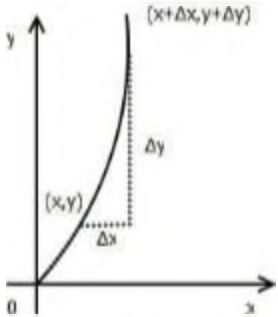
Figure 3.4: Graph of x Squared



In the above chart, we would like to obtain the derivative at 1.5. The chart of **x** squared is given by the u-shaped line. The slope at 1.5 is given by the straight line that just barely touches the u-shaped line at 1.5. This straight line is called a tangent line. If we take the derivative of Equation 3.2, we are left with an equation that will provide us with the slope of Equation 3.2 at any point **x**. It is relatively easy to derive such an equation. To see how this is done, consider

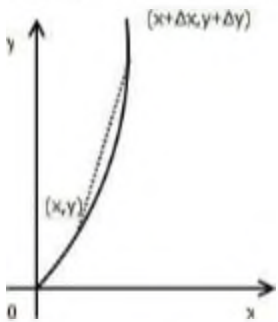
Figure 3.5.

Figure 3.5: Calculate Slope at X



Here we are given a point at (x,y). This is the point for which we would like to find the derivative. However, we need two points to calculate a slope. So, we create a second point that is equal to **x** and **y** plus **delta-x** and **delta-y**. You can see this imaginary line in Figure 3.6.

Figure 3.6: Slope of a Secant Line



The imaginary line above is called a secant line. The slope of this secant line is close to the slope at (x,y), but it is not the exact number. As **delta-x** and **delta-y** become closer to zero, the slope of the secant line becomes closer to the instantaneous slope at (x,y). We can use this fact to write an equation that is the derivative of Equation 3.2.

Before we look specifically at Equation 3.2, we will look at the general case of how to find a derivative for any function f(x). This formula uses a constant **h** that defines a second point by adding **h** to **x**. The smaller that **h** becomes, the more accurate a value we are given for the slope of the line at **x**. Equation 3.4 shows the slope of the secant line between **x** and **h**.

Equation 3.4: The Slope of the Secant Line

$$m = \frac{f(x+h) - f(x)}{(x+h) - x}$$

The derivative is equal to the above equation as **h** approaches zero. This is shown in Equation 3.5.

Equation 3.5: The Derivative of f(x) as a Slope

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{(x+h) - x}$$

You should take note of two things about the above formula. Notice the apostrophe above **f**? This designates the function as a derivative, and is pronounced "f-prime". The second is the word lim. This designates a limit. The arrow at the bottom specifies "as h approaches zero". When taking a limit, sometimes the limit is undefined at the value it approaches. Therefore, the limit is the value either at, or close to, the value the limit is approaching. In many cases, the limit can be determined simply by solving the formula with the approached value substituted for **x**.

The above formula can be simplified by removing redundant **x** terms in the denominator. This results in Equation 3.6, which is the definition of a derivative as a slope.

Equation 3.6: Derivative Formula

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Now we have a formula for a derivative, and we will see a simple application of the formula in the next section. We can use this equation to finally determine the derivative for Equation 3.2. Equation 3.2 is simply the function of **x** squared.

Using the formula from the last section, it is easy to take the derivative of a formula such as **x** squared. Equation 3.7 shows Equation 3.5 modified to use **x** squared in place of **f(x)**.

Equation 3.7: Derivative of X Squared (step 1)

$$f'(x) = \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h}$$

Equation 3.7 can be expanded using a simple algebraic rule that allows us to expand the term **(x+h)** squared. This results in Equation 3.8.

Equation 3.8: Derivative of X Squared (step 2)

$$f'(x) = \lim_{h \to 0} \frac{x^2 + 2xh + h^2 - x^2}{h}$$

This allows us to remove redundant **x** terms in the numerator, giving us Equation 3.9.

Equation 3.9: Derivative of X Squared (step 3)

$$f'(x) = \lim_{h \to 0} \frac{2xh + h^2}{h}$$

We can also cancel out the **h** terms in the numerator with the same term in the denominator. This leaves us with Equation 3.10.

Equation 3.10: Derivative of X Squared (step 4)

$$f'(x) = \lim_{h \to 0} 2x + h$$

We can now evaluate the limit at zero. This produces the final general formula for the derivative shown in Equation 3.11.

Equation 3.11: Final Derivative of X Squared

$$f'(x) = 2x$$

The above equation could be found in the front cover of many Calculus textbooks. Simple derivative formulas like this are useful for converting common equations into derivative form. Calculus textbooks usually have these derivative formulas listed in a table. Using this table, more complex derivatives can be obtained. I will not review how to obtain the derivative of any arbitrary function. Generally, when I want to take the derivative of an arbitrary function I use a program called R. R can be obtained from the R Project for Statistical Computing at this URL:

http://www.r-project.org/

The following R command could be used to find the derivative of **x** squared.

```
D( expression(x^2),"x")
```

For a brief tutorial on R, visit the following URL:

http://www.heatonresearch.com/wiki/Brief_R_Tutorial

Using Partial Derivatives

So far, we have only seen "total derivatives". A partial derivative of a function of several variables is the derivative of the function with respect to one of those variables. All other variables will be held constant. This differs from a total derivative, in which all variables are allowed to vary.

The partial derivative of a function **f** with respect to the variable **z** is variously denoted by these forms:

$$f'_z,\ f_z,\ \partial_z f,\ \text{or}\ \frac{\partial f}{\partial z}$$

The form that will be used in this book is shown here.

$$\frac{\partial f}{\partial z} = \ldots$$

For an example of partial derivatives, consider the function **f** that has more than one variable:

$$z = f(x, y) = x^2 + xy + y^2.$$

The derivative of **z**, with respect to **x** is given as follows. The variable **y** is treated as a constant.

$$\frac{\partial z}{\partial x} = 2x + y$$

Partial derivatives are an important concept for neural networks. We will typically take the partial derivative of the error of a neural network with respect to each of the weights. This will be covered in greater detail in the next chapter.

Using the Chain Rule

There are many different rules in Calculus to allow you to take derivatives manually. We just saw an example of the power rule. This rule states that given the equation:

$$f(x) = x^n$$

the derivative of f(x) will be as follows:

$$f'(x) = nx^{n-1}$$

This allows you to quickly take the derivative of any power. There are many other derivative rules, and they are very useful to know. However, if you do not wish to learn manual differentiation, you can generally get by without it by using a program such as R.

However, there is one more rule that is very useful to know. This rule is called the chain rule. The chain rule deals with composite functions. A composite function is nothing more than when one function takes the results of a second function as input. This may sound complex, but programmers make use of composite functions all the time. Here is an example of a composite function call in Java.

```
System.out.println( Math.pow(3,2) );
```

This is a composite function because we take the result of the function **pow** and feed it to **println**.

Mathematically, we write this as follows. Imagine we had functions **f** and **g**. If we wished to pass the value of 5 to **f**, and then pass the result of **f** onto **g**, we would use the expression:

$$(f \circ g)(5)$$

The chain rule of calculus gives us a relatively easy way to calculate the composite of two functions. The chain rule is given in Equation 3.12.

Equation 3.12: The Chain Rule

$$(f \circ g)'(t) = f'(g(t))g'(t)$$

The chain rule will be very valuable in calculating the derivative across an entire neural network. A neural network is very much a composite function. In a typical three layer neural network, the output of the input layer flows to the hidden layer and finally to the output layer.

Chapter Summary

In this chapter we took a look at derivatives. Derivatives are a core concept in Calculus. A derivative is defined as the slope of a curved line for one individual value of **x**. The derivative can also be thought of as the instantaneous rate of change at the point **x**. Derivatives can be calculated manually or by using a software package such as R.

Derivatives are very important for neural network training. The derivatives of activation functions are used to calculate the error gradient with respect to individual weights. Various training algorithms make use of these gradients to determine how best to update the neural network weights.

In the next chapter, we will look at backpropagation. Backpropagation is a training algorithm that adjusts the weights of neural networks to produce more desirable output from the neural network. Backpropagation works by calculating the partial derivative of the error function with respect to each of the weights.

Chapter 4: Backpropagation

- Understanding Gradients
- Calculating Gradients
- Understanding Backpropagation
- Momentum and Learning Rate

So far, we have only looked at how to calculate the output from a neural network. The output from the neural network is a result of applying the input to the neural network across the weights of several layers. In this chapter, we will find out how these weights are adjusted to produce outputs that are closer to the desired output.

This process is called training. Training is an iterative process. To make use of training, you perform multiple training iterations. These training iterations are intended to lower the global error of the neural network. Global and local error were discussed in Chapter 2.

Understanding Gradients

The first step is to calculate the gradients of the neural network. The gradients are used to calculate the slope, or gradient, of the error function for a particular weight. A weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to know that it should either increase or decrease the weight. There are a number of different training methods that make use of gradients. These training methods are called propagation training. This book will discuss the following propagation training methods:

- Backpropagation
- Resilient Propagation
- Quick Propagation

This chapter will focus on using the gradients to train the neural network using backpropagation. The next few chapters will cover the other propagation methods.

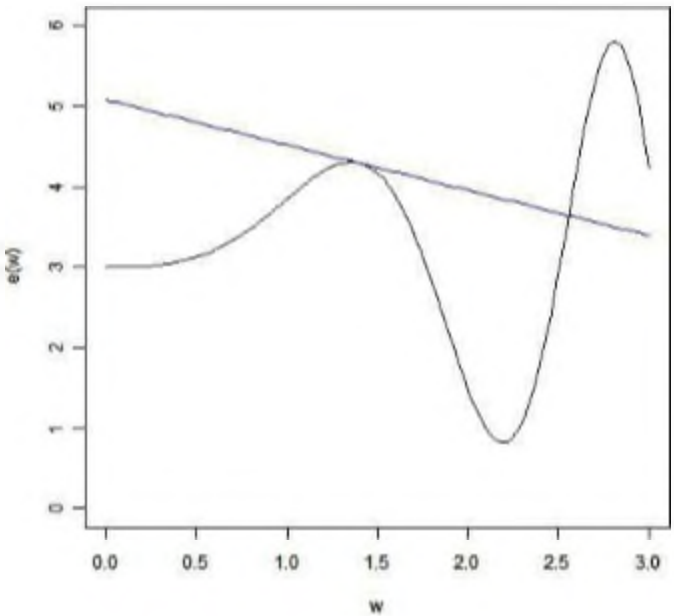
What is a Gradient

First of all, let's look at what a gradient is. Basically, training is a search. You are searching for the set of weights that will cause the neural network to have the lowest global error for a training set. If we had an infinite amount of computation resources, we would simply try every possible combination of weights and see which one provided the absolute best global error.

Because we do not have unlimited computing resources, we have to use some sort of shortcut. Essentially, all neural network training methods are really a kind of shortcut. Each training method is a clever way of finding an optimal set of weights without doing an impossibly exhaustive search.

Consider a chart that shows the global error of a neural network for each possible weight. This graph might look something like Figure 4.1.

Figure 4.1: Gradient



Looking at this chart, you can easily see the optimal weight. The optimal weight is the location where the line has the lowest y value. The problem is that we do not get to see the entire graph – that would require the exhaustive search mentioned above. We only see the error for the current value of the weight. But we can determine the slope of the error curve at a particular weight. In the above chart, we see the slope of the error curve at 1.5. The slope is given by the straight line that just barely touches the error curve at 1.5. This slope is the gradient. In this case, the slope is -0.5622.

The gradient is the instantaneous slope of the error function at the specified weight. This uses the same definition for the derivative that we learned in Chapter 3. The gradient is given by the derivative of the error curve at that point. This line tells us something about how steep the error function is at the given weight.

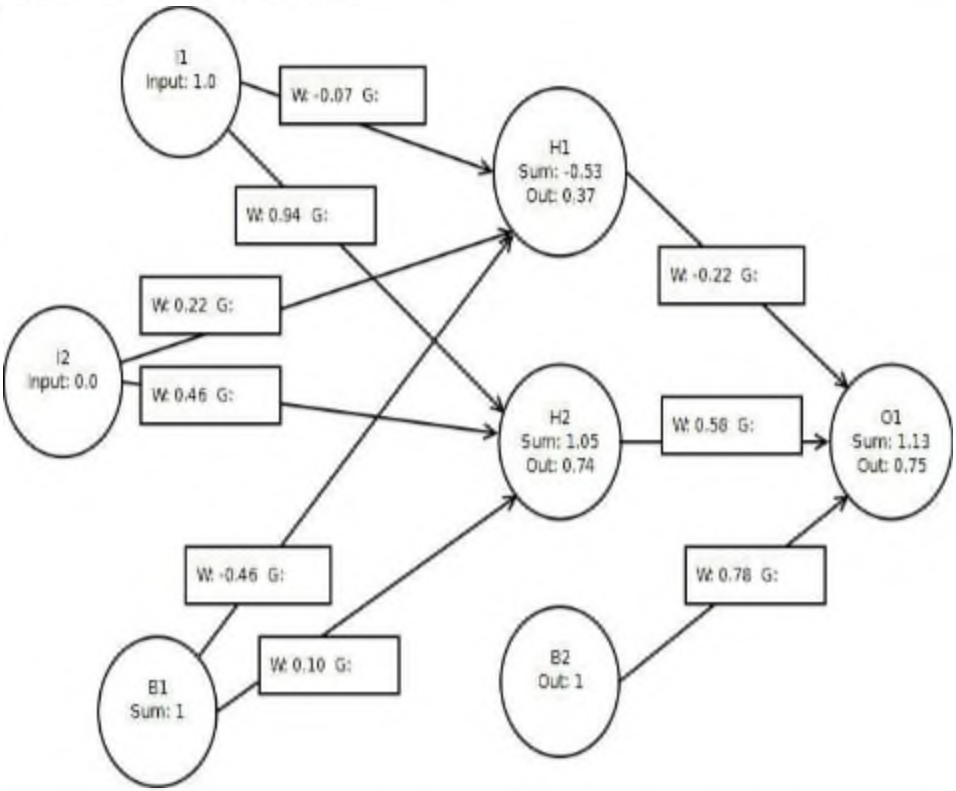
Used with a training technique, this can provide some insight into how the weight should be adjusted for a lower error. Now that we have seen what a gradient is, in the next section, we will find out how to calculate a gradient.

Calculating Gradients

We will now look at how to calculate the gradient. We will calculate an

individual gradient for each weight. I will show you the equations as well as how to apply them to an actual neural network with real numbers. The neural network that we will use is shown in Figure 4.2.

Figure 4.2: An XOR Network



The neural network above is a typical three layer feedforward network like the ones we have seen before. The circles indicate neurons. The lines connecting the circles are the weights. The rectangles in the middle of the connections give the weight for each connection.

The problem that we now face is how to calculate the partial derivative for the output of each neuron. This can be accomplished using a method based on the chain rule of Calculus. The chain rule was discussed in Chapter 3.

We will begin with one training set element. For Figure 4.2 we are providing an input of **[1,0]** and expecting an output of **[1]**. You can see that the input is applied on the above figure, as the first input neuron has an input value of 1.0 and the second input neuron has an input value of 0.0.

This input feeds through the network and eventually produces an output. The exact process by which the output and sums are calculated was covered in

Chapter 1. Backpropagation has both a forward and backward pass. The forward pass occurred when the output of the neural network was calculated. We will calculate the gradients only for this item in the training set. Other items in the training set will have different gradients. We will discuss how the gradients for each individual training set element are combined later in this chapter, when we talk about "Batch and Online Training".

We are now ready to calculate the gradients. There are several steps involved in calculating the gradients for each weight. These steps are summarized here.

- Calculate the error, based on the ideal of the training set
- Calculate the node delta for the output neurons
- Calculate the node delta for the interior neurons
- Calculate individual gradients

These steps will be covered in the following sections.

Calculating the Node Deltas

The first step is to calculate a constant value for every node, or neuron, in the neural network. We will start with the output nodes and work our way backwards through the neural network – this is where the term, backpropagation, comes from. We initially calculate the errors for the output neurons and propagate these errors backwards through the neural network.

The value that we will calculate for each node is called the node delta. The term, layer delta, is also sometimes used to describe this value. Layer delta describes the fact that these deltas are calculated one layer at a time. The method for calculating the node deltas differs depending on whether you are calculating for an output or interior node. The output neurons are, obviously, all output nodes. The hidden and input neurons are the interior nodes. The equation to calculate the node delta is provided in Equation 4.1.

Equation 4.1: Calculating the Node Deltas

$$\delta_i = \begin{cases} -Ef_i' & \text{, output nodes} \\ f_i' \sum_k w_{ki}\delta_k & \text{, interior nodes} \end{cases}$$

We will calculate the node delta for all hidden and non-bias neurons. There is no need to calculate the node delta for the input and bias neurons. Even though the node delta can easily be calculated for input and bias neurons using the above equation, these values are not needed for the gradient calculation. As you will soon see, gradient calculation for a weight only looks at the neuron that

the weight is connected to. Bias and input neurons are only the beginning point for a connection. They are never the end point.

We will begin by using the formula for the output neurons. You will notice that the formula uses a value **E**. This is the error for this output neuron. You can see how to calculate **E** from Equation 4.2.

Equation 4.2: The Error Function

$$E = (a - i)$$

You may recall a similar equation to Equation 2.1 from Chapter 2. This is the error function. Here, we subtract the ideal from the actual. For the neural network provided in Figure 4.2, this can be written like this:

```
E = 0.75 - 1.00 = -0.25
```

Now that we have **E**, we can calculate the node delta for the first (and only) output node. Filling in Equation 4.1, we get the following:

```
-( -0.25) * dA(1.1254) = 0.185 * 0.25 = 0.05
```

The value of 0.05 will be used for the node delta of the output neuron. In the above equation, **dA** represents the derivative of the activation function. For this example, we are using a sigmoid activation function. The sigmoid activation function is shown in Equation 4.3.

Equation 4.3: The Sigmoid Function

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

The derivative of the sigmoid function is shown in Equation 4.4.

Equation 4.4: The Derivative of the Sigmoid Function

$$f'(x) = s(x) * (1.0 - s(x))$$

Now that the node delta has been calculated for the output neuron, we should calculate it for the interior neurons as well. The equation to calculate the node delta for interior neurons was provided in Equation 4.1. Below, we apply this for the first hidden neuron:

```
dA(sum of H1) * (O1 Node Delta * Weight of H1 -> O1)
```

You will notice that Equation 4.1 called for the summing of a number of items based on the number of inbound connections to output neuron one. Because there is only one inbound connection to output neuron one, there is only

one value to sum. This value is the product of the output neuron one node delta and the weight between hidden neuron one and output neuron one.

Filling in actual values for the above expression, we are left with the following:

```
dA(-0.53) * (0.05 * -0.22) = -0.0025
```

The value -0.0025 is the node delta for the first hidden neuron. Calculating the second hidden neuron follows exactly the same form as above. The second neuron would be computed like this:

```
dA(sum of H2) * (O1 Node Delta * Weight of H2 -> O1)
```

Plugging in actual numbers, we find this:

```
dA(1.05) * (0.05 * 0.58) = 0.0055
```

The value of 0.0055 is the node delta for the second hidden neuron.

As previously explained, there is no reason to calculate the node delta for either the bias neurons or the input neurons. We now have every node delta needed to calculate a gradient for each weight in the neural network. Calculation of the individual gradients will be discussed in the next section.

Calculating the Individual Gradients

We can now calculate the individual gradients. Unlike the node deltas, only one equation is used to calculate the actual gradient. A gradient is calculated using Equation 4.5.

Equation 4.5: Individual Gradient

$$\frac{\partial E}{\partial w_{(ik)}} = \delta_k \cdot o_i$$

The above equation calculates the partial derivative of the error (**E**) with respect to each individual weight. The partial derivatives are the gradients. Partial derivatives were discussed in Chapter 3. To determine an individual gradient, multiply the node delta for the target neuron by the weight from the source neuron. In the above equation, **k** represents the target neuron and **i** represents the source neuron.

To calculate the gradient for the weight from **H1** to **O1**, the following values would be used:

```
output(h1) * nodeDelta(o1)
(0.37 * 0.05) = 0.01677
```

It is important to note that in the above equation, we are multiplying by the output of hidden 1, not the sum. When dealing directly with a derivative you should supply the sum. Otherwise, you would be indirectly applying the activation function twice. In the above equation, we are not dealing directly with the derivative, so we use the regular node output. The node output has already had the activation function applied.

Once the gradients are calculated, the individual positions of the weights no longer matter. We can simply think of the weights and gradients as single dimensional arrays. The individual training methods that we will look at will treat all weights and gradients equally. It does not matter if a weight is from an input neuron or an output neuron. It is only important that the correct weight is used with the correct gradient. The ordering of these weight and gradient arrays is arbitrary. However, Encog uses the following order for the above neural network:

```
Weight/Gradient 0: Hidden 1 -> Output 1
Weight/Gradient 1: Hidden 2 -> Output 1
Weight/Gradient 2: Bias 2 -> Output 1
Weight/Gradient 3: Input 1 -> Hidden 1
Weight/Gradient 4: Input 2 -> Hidden 1
Weight/Gradient 5: Bias 1 -> Hidden 1
Weight/Gradient 6: Input 1 -> Hidden 2
Weight/Gradient 7: Input 2 -> Hidden 2
Weight/Gradient 8: Bias 1 -> Hidden 2
```

The various learning algorithms will make use of the weight and gradient arrays. It is also important to note that these are two separate arrays. There is a weight array, and there is a gradient array. Both arrays will be exactly the same length. Training a neural network is nothing more than adjusting the weights to provide desirable output. In this chapter, we will see how backpropagation uses the gradient array to modify the weight array.

Applying Back Propagation

Backpropagation is a simple training method that uses the calculated gradients of a neural network to adjust the weights of the neural network. This is a form of gradient descent, as we are descending down the gradients to lower values. As these weights are adjusted, the neural network should produce more desirable output. The global error of the neural network should fall as it is trained. Before we can examine the backpropagation weight update process, we must look at two different ways that the gradients can be calculated.

Batch and Online Training

We have already covered how to calculate the gradients for an individual training set element. Earlier in this chapter, we saw how we could calculate the gradients for a case where the neural network was given an input of [1,0] and an output of [1] was expected. This works fine for a single training set element. However, most training sets have many elements. There are two different ways to handle multiple training set elements. These two approaches are called online and batch training.

Online training implies that you modify the weights after every training set element. Using the gradients that you obtained for the first training set element, you calculate and apply a change to the weights. Training progresses to the next training set element and also calculates an update to the neural network. This training continues until every training set element has been used. At this point, one iteration, or epoch, of training has completed.

Batch training also makes use of every training set element. However, the weights are not updated for every training set element. Rather, the gradients for each training set element are summed. Once every training set element has been used, the neural network weights can be updated. At this point, the iteration is considered complete.

Sometimes, a batch size will be set. For example, you might have a training set size of 10,000 elements. You might choose to update the weights of the neural network every 1,000 elements. This would cause the neural network weights to be updated 10 times during the training iteration.

Online training was the original method used for backpropagation. However, online training is inefficient, because the neural network must be constantly updated. Additionally, online training is very difficult to implement in a multi-threaded manner that can take advantage of multi-core processors. For these reasons, batch training is generally preferable to online training.

Backpropagation Weight Update

We are now ready to update the weights. As previously mentioned, we will treat the weights and gradients as a single dimensional array. Given these two arrays, we are ready to calculate the weight update for an iteration of backpropagation training. The formula to update the weights for backpropagation is shown in Equation 4.6.

Equation 4.6: Backpropagation Weight Update

$$\Delta w_{(t)} = \epsilon \frac{\partial E}{\partial w_{(t)}} + \alpha \Delta w_{(t-1)}$$

The above equation calculates the change in weight for each element in the weight array. You will also notice that the above equation calls for the weight change from the previous iteration. These values must be kept in another array.

The above equation calculates the weight delta to be the product of the gradient and the learning rate (represented by epsilon). Additionally, the product of the previous weight change and the momentum value (represented by alpha) is added. The learning rate and momentum are two parameters that must be provided to the backpropagation algorithm. Choosing values for learning rate and momentum is very important to the performance of the training. Unfortunately, the process for determining learning rate and momentum is mostly trial and error.

The learning rate scales the gradient and can slow down or speed up learning. A learning rate below zero will slow down learning. For example, a learning rate of 0.5 would decrease every gradient by 50%. A learning rate above 1.0 would speed up training. In reality, the learning rate is almost always below zero.

Choosing a learning rate that is too high will cause your neural network to fail to converge. A neural network that is failing to converge will generally have a high global error that simply bounces around, rather than converging to a low value. Choosing a learning rate that is too low will cause the neural network to take a great deal of time to converge.

Like the learning rate, the momentum is also a scaling factor. Momentum determines the percent of the previous iteration's weight change that should be applied to this iteration. Momentum is optional. If you do not want to use momentum, just specify a value of zero.

Momentum is a technique that was added to backpropagation to help the training find its way out of local minima. Local minima are low points on the error graph that are not the true global minimum. Backpropagation has a tendency to find its way into a local minimum and not find its way back out

again. This causes the training to converge to a higher undesirable error. Momentum gives the neural network some force in its current direction and may allow it to force through a local minimum.

We are now ready to plug values into Equation 4.6 and calculate a weight delta. We will calculate a weight change for the first iteration using the neural network that we previously used in this chapter. So far, we have only calculated gradients for one training set element. There are still four other training set elements to calculate for. We will sum all four gradients together to apply batch training. The batch gradient is calculated by summing the individual gradients, which are listed here:

```
0.01677795762852397
-0.05554301180824532
0.021940533165555356
-0.05861906411780882
```

Earlier in the chapter we calculated the first gradient, listed above. If you would like to see the calculation for the others, this information can be found at the following URL:

http://www.heatonresearch.com/wiki/Back_Propagation

Summing all of these gradients produces this batch update gradient:

```
-0.07544358513197481
```

For this neural network, we will use a learning rate of 0.7 and a momentum of 0.3. These are just arbitrary values. However, they do work well for training an XOR neural network. Plugging these values into Equation 4.6 results in this equation:

```
delta = (0.7 * -0.0754) + (0.3 * 0.0) = -0.052810509592382364
```

This is the first training iteration, so the previous delta value is 0.0. The momentum has no effect on the first iteration. This delta value will be added to the weight to alter the neural network for the first training iteration. All of the other weights in this neural network will be updated in the same way, according to their calculated gradient.

This first training iteration will lower the neural network's global error slightly. Additional training iterations will lower the error further. The following program output shows the convergence of this neural network.

```
Epoch #1 Error:0.3100155809627523
Epoch #2 Error:0.2909988918032235
Epoch #3 Error:0.2712902750837602
Epoch #4 Error:0.2583119003843881
Epoch #5 Error:0.2523050561276289
```

```
Epoch #6 Error:0.2502986971902545
Epoch #7 Error:0.2498182295192154
Epoch #8 Error:0.24974245650541688
Epoch #9 Error:0.24973458893806627
Epoch #10 Error:0.24972923906975902
...
Epoch #578 Error:0.010002702374503777
Epoch #579 Error:0.009947830890527089
Neural Network Results:
1.0,0.0, actual=0.9040102333814147,ideal=1.0
0.0,0.0, actual=0.09892634022671229,ideal=0.0
0.0,1.0, actual=0.904020682439766,ideal=1.0
1.0,1.0, actual=0.10659032105865764,ideal=0.0
```

Each iteration, or epoch, decreases the error. Once the error drops below one percent, the training stops. You can also see the output from the neural network for the XOR data. The answers are not exactly correct, but it is very clear that that the two training cases that should be 1.0 are much closer to 1.0 than the others are.

Chapter Summary

In this chapter you were introduced to backpropagation. Backpropagation is one of the oldest and most commonly used training algorithms available for neural networks. Backpropagation works by calculating a gradient value for each weight in the network. Many other training methods also make use of these gradient values.

There is one gradient for each weight in the neural network. Calculation of the gradients is a step-by-step process. The first step in calculating the gradients is to calculate the error for each of the outputs of the neural network. This error is for one training set element. The gradients for all training set elements may be batched together later in the process.

Once the error for the output layer has been established, you can go on to calculate values for each of the output neurons. These values are called the node deltas for each of the output neurons. We must calculate the node deltas for the output layer first. We calculate the node deltas for each layer of the neural network, working our way backwards to the input layer. This is why this technique is called backpropagation.

Once the node deltas have been calculated, it is very easy to calculate the gradients. At the end, you will have all of the gradients for one training set element. If you are using online training, you will now use these gradients to apply a change to the weights of the neural network. If you are using batch training, you will sum the gradients from each of the training set elements into a single set of gradients for the entire training set.

Backpropagation must be provided with a learning rate and momentum. Both of these are configuration items that will have an important effect on the training speed of your neural network. Learning rate specifies how fast the weights should be updated. Too high a learning rate will cause a network to become unstable. Too low a learning rate will cause the neural network to take too long to train. Momentum allows the neural network to escape local minima. Local minima are low points in the error graph that are not the true global minimum.

Choosing values for the learning rate and momentum can be tricky. Often it is just a matter of trial and error. The Resilient Propagation training method (RPROP) requires no parameters to be set. Further, RPROP often trains much faster than backpropagation. RPROP will be covered in the next chapter.

Chapter 5: Faster Training with RPROP

- Understanding Error Calculation
- The Error Function
- Error Calculation Methods
- How is the Error Used

In the last chapter, we looked at backpropagation. Backpropagation is one of the oldest and most popular methods for training a neural network. Unfortunately, backpropagation is also one of the slowest methods for training a neural network. In this chapter, we will take a look at a faster training technique called resilient propagation or RPROP.

RPROP works very much like backpropagation. Both backpropagation and RPROP must first calculate the gradients for the weights of the neural network. Where backpropagation and RPROP differ is in the way in which the gradients are used.

In a simple XOR example, it typically takes backpropagation over 500 iterations to converge to a solution with an error rate of below one percent. It will usually take RPROP around 30 to 100 iterations to accomplish the same thing. This large increase in performance is one reason why RPROP is a very popular training algorithm.

Another factor in the popularity of RPROP is that there are no necessary training parameters to the RPROP algorithm. When you make use of backpropagation, you must specify the learning rate and momentum. These two parameters can have a huge impact on the effectiveness of your training. RPROP does include a few training parameters, but they can almost always be left at their default settings.

There are several variants of the RPROP protocol. Here are some of the variants.

- RPROP+
- RPROP-
- iRPROP+
- iRPROP-

This book will focus on classic RPROP, as described in a 1994 paper by Martin Reidmiller entitled "Rprop – Description and Implementation Details".

The other four variants described above are relatively minor adaptations of classic RPROP. For more information about all variants of RPROP, you can visit the following URL:

http://www.heatonresearch.com/wiki/RPROP

In the next sections, we will see how the RPROP algorithm is implemented.

RPROP Arguments

As previously mentioned, one advantage RPROP has over backpropagation is that no training arguments need to be provided in order for RPROP to be used. That is not to say that there are no configuration settings for RPROP. The configuration settings for RPROP do not usually need to be changed from their defaults. However, if you really want to change them, there are several configuration settings that you can set for RPROP training. These configuration settings are:

- Initial Update Values
- Maximum Step

As you will see in the next section, RPROP keeps an array of update values for the weights. This determines how large of a change will be made to each weight. This is something like the learning rate in backpropagation, only much better. There is an update value for every weight in the neural network. This allows the update values to be fine tuned to each individual weight as training progresses. Some backpropagation algorithms will vary the learning rate and momentum as learning progresses. The RPROP approach is much better, because unlike backpropagation, it does not simply use a single learning rate for the entire neural network.

These update values must start from somewhere. The "initial update values" argument defines this. By default, this argument is set to a value of 0.1. As a general rule, this default should never be changed. One possible exception to this is in a neural network that has already been trained. If the neural network is already trained, then some of the initial update values are going to be too strong for the neural network. The neural network will regress for many iterations before it is able to improve. An already trained neural network may benefit from a much smaller initial update.

Another approach for an already trained neural network is to save the update values once training stops and use them for the new training. This will allow you to resume training without the initial spike in errors that you would normally see when resuming resilient propagation training. This method will only work if you are resuming resilient propagation. If you were previously training the neural network with a different training algorithm, then you will not have an array of update values to restore from.

As training progresses, the gradients will be used to adjust the updates up and down. The "maximum step" argument defines the maximum upward step size that can be taken over the update values. The default value for the maximum step argument is 50. It is unlikely that you will need to change the value of this

argument.

As well as arguments defined for RPROP, there are also constants. These constants are simply values that are kept by RPROP during processing. These values are never changed. The constants are:

- Delta Minimum (1e-6)
- Negative Eta (0.5)
- Positive Eta (1.2)
- Zero Tolerance (1e-16)

Delta minimum specifies the minimum value that any of the update values can go to. This is important, because if an update value were to go to zero it would never be able to increase beyond zero. Negative and positive eta will be described in the next sections. The zero tolerance defines how close a number should be to zero before that number is equal to zero. In computer programming, it is typically considered bad practice to directly compare a floating point number to zero. This is because the number would have to be exactly equal to zero.

Data Structures

There are several data structures that must be kept in memory while RPROP training is performed. These structures are all arrays of floating point numbers. They are summarized here:

- Current Update Values
- Last Weight Change Values
- Current Weight Change Values
- Current Gradient Values
- Previous Gradient Values

The current update values are kept to hold the current update values for the training. If you wish to be able to resume training at some point, this is the array that must be stored. There is one update value per weight. These update values cannot go below the minimum delta constant. Likewise, these update values cannot exceed the maximum step argument.

The last weight delta value must also be tracked. Backpropagation kept this value for momentum. RPROP uses this value in a different way than backpropagation. We will see how this array is used in the next section. The current weight change is kept long enough to change the weights and then is copied to the previous weight change.

The current and previous gradients are needed too. RPROP is particularly interested when the sign changes from the current gradient to the previous gradient. This indicates that an action must be taken with regard to the update values. This is covered in the next section.

Understanding RPROP

In the last few sections, the arguments, constants and data structures necessary for RPROP were covered. In this section, we will see exactly how to run through an iteration of RPROP. In the next section, we will apply real numbers to RPROP and see how training iterations progress for an XOR training. We will train exactly the same network that we used with backpropagation. This will give us a good idea of the difference in performance of backpropagation compared to RPROP.

When we talked about backpropagation, we mentioned two weight update methods: online and batch. RPROP does not support online training. All weight updates used with RPROP will be performed in batch mode. Because of this, each iteration of RPROP will receive gradients that are the sum of the individual gradients of each training set. This is consistent with using backpropagation in batch mode.

There are three distinct steps in an iteration of an RPROP iteration. They are covered in the next three sections.

Determine Sign Change of Gradient

At this point, we should have the gradients. These gradients are nearly exactly the same as the gradients calculated by the backpropagation algorithm. The only difference is that RPROP uses a gradient that is the inverse of the backpropagation gradient. This is easy enough to adjust. Simply place a negative operator in front of every backpropagation gradient. Because the same process is used to obtain gradients in both RPROP and backpropagation, we will not repeat it here. To learn how to calculate a gradient, refer to Chapter 4.

The first step is to compare the gradient of the current iteration to the gradient of the previous iteration. If there is no previous iteration, then we can assume that the previous gradient was zero.

To determine whether the gradient sign has changed, we will use the sign (sgn) function. The sgn function is defined in Equation 5.1.

Equation 5.1: The Sign Function (sgn)

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

The sgn function returns the sign of the number provided. If x is less than zero, the result is -1. If x is greater than zero, then the result is 1. If x is equal to

zero, then the result is zero. I usually implement the sign function to use a tolerance for zero, since it is nearly impossible for floating point operations to hit zero precisely on a computer.

To determine whether the gradient has changed sign, Equation 5.2 is used.

Equation 5.2: Determine Gradient Sign Change

$$c = \frac{\partial E^{(t)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t-1)}}{\partial w_{ij}}$$

Equation 5.2 will result in a constant **c**. This value is evaluated to be negative or positive or close to zero. A negative value for **c** indicates that the sign has changed. A positive value indicates no change in sign for the gradient. A value near zero indicates that there was either a very small change in sign or nearly a change in sign. To see all three of these outcomes, consider the following situations.

```
-1 * 1 = -1  (negative, changed from negative to positive)
1 * 1 = 1 (positive, no change in sign)
1.0 * 0.000001 = 0.000001 (near zero, almost changed signs, but
```

Now that we have calculated the constant **c**, which gives some indication of sign change, we can calculate the weight change. This is covered in the next section.

Calculate Weight Change

Now that we have the change in sign of the gradient, we can see what happens in each of the three cases mentioned in the previous section. These three cases are summarized in Equation 5.3.

Equation 5.3: Calculate RPROP Weight Change

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)} & , \text{if } c > 0 \\ +\Delta_{ij}^{(t)} & , \text{if } c < 0 \\ 0 & , \text{otherwise} \end{cases}$$

This equation calculates the actual weight change for each iteration. If the value of **c** is positive, then the weight change will be equal to the negative of the weight update value. Similarly, if the value of **c** is negative, the weight change will be equal to the positive of the weight update value. Finally, if the value of **c** is near zero, then there will be no weight change.

Modify Update Values

The weight update values seen in the previous section are used in each iteration to update the weights of the neural network. There is a separate weight update value for every weight in the neural network. This works much better than a single learning rate for the entire neural network, such as was used in backpropagation. These weight update values are modified during each training iteration, as seen in Equation 5.4.

Equation 5.4: Modify Update Values

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ \cdot \Delta_{ij}^{(t-1)} & \text{, if } e > 0 \\ \eta^- \cdot \Delta_{ij}^{(t-1)} & \text{, if } e < 0 \\ \Delta_{ij}^{(t-1)} & \text{, otherwise} \end{cases}$$

The weight update values are modified in a way that is very similar to how the weights themselves are modified. Just like the weights, the weight update values are changed based on the value **e**, as previously calculated.

If the value of **e** is positive, then the weight update value will be multiplied by the value of positive eta. Similarly, if the value of **e** is negative, the weight update value will be multiplied by negative eta. Finally, if the value of **e** is near zero, then there will be no change to the weight update value.

RPROP Update Examples

We will now look at a few iterations of a RPROP train of a neural network. The critical stats on this neural network are listed here:

```
Layers: 3(input, hidden, output)
Input Neurons: 2
Hidden Neurons: 2
Output Neurons: 1
Activation Function: Sigmoid
Bias Neurons: Yes
Total weights: 9
```

The initial random weights for this neural network are given here:

```
Weight 0: H1->O1: -0.22791948943117624
Weight 1: H2->O1: 0.581714099641357
Weight 2: B2->O1: 0.7792991203673414
Weight 3: I1->H1: -0.06782947598673161
Weight 4: I2->H1: 0.22341077197888182
Weight 5: B1->H1: -0.4635107399577998
Weight 6: I1->H2: 0.9487814395569221
Weight 7: I2->H2: 0.46158711646254
Weight 8: B1->H2: 0.09750161997450091
```

The neural network will be trained for the XOR operator, just as was done in Chapter 4. However, in this chapter RPROP will be used in place of backpropagation.

Training Iteration #1

For the first training iteration, the weight update values are all set to 0.1. This is their default starting point. We begin by calculating the gradients of each of the weights. This is the same gradient calculation as was performed for backpropagation, so the calculation will not be covered here. The gradients are provided here:

```
Gradient 0: H1->O1:-0.07544358513197481
Gradient 1: H2->O1:-0.12346935587390481
Gradient 2: B2->O1:-0.18705713395934637
Gradient 3: I1->H1:0.005292326734004241
Gradient 4: I2->H1:0.0049016107791925246
Gradient 5: B1->H1:0.010264148244428655
Gradient 6: I1->H2:-0.0068740307089347
Gradient 7: I2->H2:-0.005236293038814788
Gradient 8: B1->H2:-0.02103299467864286
```

Now that we have the gradients, we must calculate the weight change from the previous gradients. There are no previous gradients, so their values are all

zero. The calculation of **c** is shown here:

```
c 0: H1->O1:0.0 * -0.07544358513197481 = 0
c 1: H2->O1:0.0 * -0.12346935587390481 = 0
c 2: B2->O1:0.0 * -0.18705713395934637 = 0
c 3: I1->H1:0.0 * 0.005292326734004241 = 0
c 4: I2->H1:0.0 * 0.0049016107791925246 = 0
c 5: B1->H1:0.0 * 0.010264148244428655 = 0
c 6: I1->H2:0.0 * -0.0068740307089347 = 0
c 7: I2->H2:0.0 * -0.005236293038814788 = 0
c 8: B1->H2:0.0 * -0.02103299467864286 = 0
```

From this we can determine each of the weight change values. The value of **c** is zero in all cases. Because of this, the weight change value is the negative of the weight update value. This results in a weight change of -0.01 for every weight.

```
Weight Change 0: -0.01
Weight Change 1: -0.01
Weight Change 2: -0.01
Weight Change 3: -0.01
Weight Change 4: -0.01
Weight Change 5: -0.01
Weight Change 6: -0.01
Weight Change 7: -0.01
Weight Change 8: -0.01
```

This leaves us with the following weights:

```
Weight 0: H1->O1: -0.3279194894311762
Weight 1: H2->O1: 0.48171409964135703
Weight 2: B2->O1: 0.6792991203673414
Weight 3: I1->H1: 0.03217052401326839
Weight 4: I2->H1: 0.3234107719788818
Weight 5: B1->H1: -0.3635107399577998
Weight 6: I1->H2: 0.8487814395569221
Weight 7: I2->H2: 0.36158711646254804
Weight 8: B1->H2: -0.0024983800254990973
```

This ends the first iteration of the RPROP. Some implementations of RPROP will suppress any weight changes for the first training iteration to allow it to "initialize". However, skipping the weight update is generally unnecessary, as the weights are starting from random values anyhow.

The first iteration is now complete. The first iteration serves as little more than an initialization iteration.

Training Iteration #2

We begin the second iteration by again calculating the gradients of each of

the weights. The gradients will have changed, because the underlying weights changed. We will use these new gradient values to calculate a new value of **e** for each weight.

```
c 0: H1->O1: -0.0754435851317481 * -0.07564714780823276 = 1
c 1: H2->O1: -0.12346935587390481 * -0.10495082682420408 = 1
c 2: B2->O1: -0.18705713395934637 * -0.16712652502209419 = 1
c 3: I1->H1: 0.005292326734004241 * 0.007147520399328029 = 1
c 4: I2->H1: 0.0049016107791925246 * 0.0065760422990062 1 = 1
c 5: B1->H1: 0.010264148244428655 * 0.013445893781261988 = 1
c 6: I1->H2: -0.0068740307089347 * -0.006335334269910348 = 1
c 7: I2->H2: -0.005236293038814788 * -0.004772389693042953 = 1
c 8: B1->H2: -0.02103299467864286 * -0.01641086135590903 = 1
```

From this we can determine each of the weight change values. The value of **e** is one in all cases. This means that none of the gradients' signs changed. Because of this, the weight change value is the negative of the weight update value. This results in a weight change of -0.01 for every weight.

```
Weight Change 0: -0.01
Weight Change 1: -0.01
Weight Change 2: -0.01
Weight Change 3: -0.01
Weight Change 4: -0.01
Weight Change 5: -0.01
Weight Change 6: -0.01
Weight Change 7: -0.01
Weight Change 8: -0.01
```

This results in the following weights:

```
-0.4479194894311762
0.36171409964135703
0.5592991203673414
0.1521705240132684
0.4434107719788818
-0.24351073995779982
0.7287814395569221
0.24158711646254805
-0.12249838002549909
```

Everything continues to move in the same direction as in the first iteration. The update values will also move up from 0.1 to 0.12, which is the direction specified by positive eta. This ends iteration 2.

As you can see, the training will continue in the present path until one of the gradients changes signs. The update values will continue to grow, and the weights will continue to change by a value based on the sign of their gradient and the weight update value. This continues until iteration #8, when the update values will begin to take on different values.

Training Iteration #8

We begin the eight iteration by again calculating the gradients of each of the weights. The weights have changed over the iterations that we skipped. This will cause the gradients to change. We will use the previous gradients along with these new gradients to calculate a new value of **c** for each weight.

```
c 0: H1->O1: -0.024815885942307825 * -0.009100949661316388 = 1
c 1: H2->O1: -0.023266017866306714 * -0.0091094801332018 = 1
c 2: B2->O1: -0.045150613856680816 * -0.01879394429346648 = 1
c 3: I1->H1: 7.967367771930835E-4 * -0.001991948196987281 = -1
c 4: I2->H1: 8.920300028002447E-4 * -0.0015368274160319773 = -1
c 5: B1->H1: 0.005021027721309641 * 0.0011945995248252954 = 1
c 6: I1->H2: -0.0010888731437352726 * -3.140593778629495E-4 = 1
c 7: I2->H2: -7.180424572079871E-4 * -1.4136606514778948E-4 = 1
c 8: B1->H2: -0.002215756957751366 * -7.264418447125096E-4 = 1
```

As you can see, there are two weights that now have a -1 value for c. This indicates that the gradient sign has changed. This means we have passed a local minimum, and error is now climbing. We must do something to stop this increase in error for weights three and four.

Instead of multiplying the weight update value by positive eta, we will now use negative eta for weights three and four. This will scale the weight back from the value of 0.04319 to 0.021599. As previously discussed, the value for negative eta is a constant defined to be 0.5. At the end of this iteration, the weight update values are:

```
Update 0: H1->O1: 0.05183999999999999
Update 1: H2->O1: 0.05183999999999999
Update 2: B2->O1: 0.05183999999999999
Update 3: I1->H1: 0.021599999999999998
Update 4: I2->H1: 0.021599999999999998
Update 5: B1->H1: 0.05183999999999999
Update 6: I1->H2: 0.05183999999999999
Update 7: I2->H2: 0.05183999999999999
Update 8: B1->H2: 0.05183999999999999
```

As you can see, all weights have continued increasing except for weights three and four. These two weights have been scaled back, and will receive a much smaller update in the next iteration. Additionally, the last gradient for each of these two weights is set to zero. This will prevent a modification of the weight update values in the next training iteration. This can be accomplished because **c** will be zero in the next step since the previous gradient is zero.

This process will continue until the global error of the neural network falls to an acceptable level. You can see the complete training process here.

```
Epoch #1 Error:0.3100155809627523
```

```
Epoch #2 Error:0.2888003866116162
Epoch #3 Error:0.267380775814409
Epoch #4 Error:0.25242444534566344
Epoch #5 Error:0.25517114662144347
Epoch #6 Error:0.25242444534566344
Epoch #7 Error:0.2508797332883249
Epoch #8 Error:0.25242444534566344
Epoch #9 Error:0.2509114256134314
...
Epoch #127 Error:0.029681452838256468
Epoch #128 Error:0.026157454894821013
Epoch #129 Error:0.023541442841907054
Epoch #130 Error:0.0253591989944982
Epoch #131 Error:0.020825411676740083
Epoch #132 Error:0.01754524879617848
Epoch #133 Error:0.015171808565942009
Epoch #134 Error:0.012948657050164597
Epoch #135 Error:0.011092515418846417
Epoch #136 Error:0.009750156492866442
```

This same set of weights took 579 iterations with backpropagation. As you can see, RPROP outperforms backpropagation by a considerable margin. With more advanced random weight generation, the number of iterations needed for RPROP can be brought down even further. This will be demonstrated in the next chapter.

Chapter Summary

In this chapter, we saw the resilient propagation (RPROP) training method. This training method is much more efficient than backpropagation. Considerably fewer training iterations are necessary for RPROP compared to backpropagation.

RPROP works by keeping an array of update values used to modify the weights of the neural network. The gradients are not used to update the weights of the neural network directly. Rather, the gradients influence how the weight update values are changed for each iteration. Only the sign of the gradient is used to determine the direction in which to take the update values. This makes for a considerable improvement over backpropagation.

So far, we have randomized each neural network with purely random numbers. This does not always lead to the fastest training times. As we will see in the next chapter, we can make modifications to the random numbers to which the neural network is initialized. These small changes will yield faster training times.

Chapter 6: Weight Initialization

- Ranged Random Weights
- Trained and Untrained Neural Networks
- Nguyen-Widrow Weight Initialization

Neural networks must start with their weights initialized to random numbers. These random weights provide the training algorithms with a starting point for the weights. If all of the weights of a neural network were set to zero, the neural network would never train to an acceptable error level. This is because a zeroed weight matrix would place the neural network into a local minimum from which it can never escape.

Often, the weights of neural networks are simply initialized with random numbers between a specific range. The range -1 to +1 is very popular. These random weights will change as the neural network is trained to produce acceptable outputs. In the next section, we will find out what trained and untrained neural networks look like.

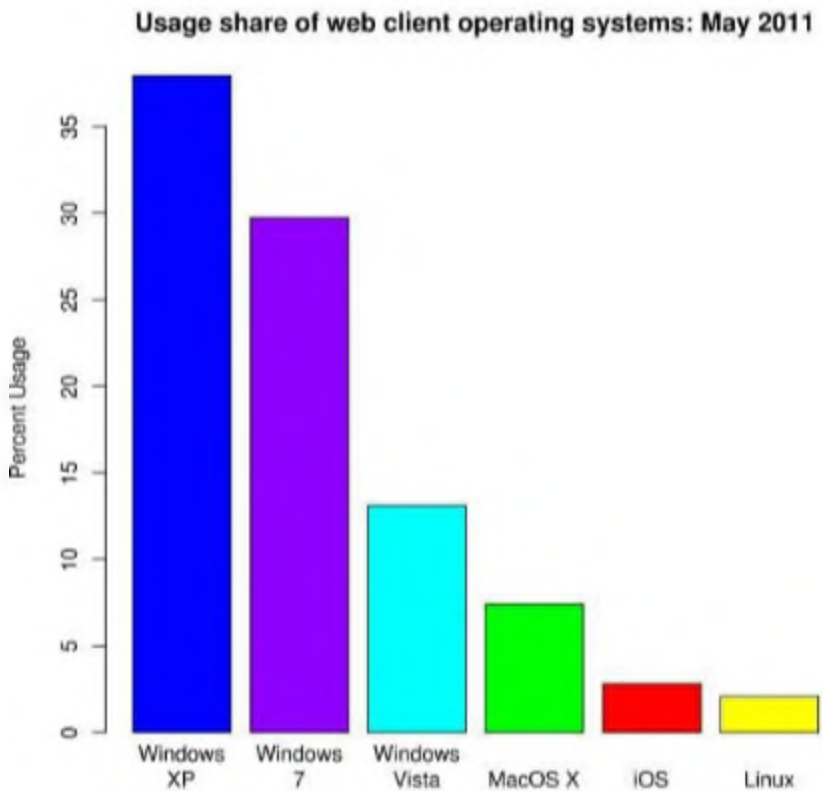
Later in this chapter, we will see that these random weights can be modified to help the neural network to train faster. The Nguyen-Widrow weight initialization algorithm is a popular technique to adjust these starting weights. This algorithm puts the weights into a position that is much more conducive to training. This means that you need fewer training iterations to get the neural network to an acceptable error rate.

Looking at the Weights

In previous chapters, we looked at the weights of a neural network as an array of numbers. You can't typically glance at a weight array and see any sort of meaningful pattern. However, if the weights are represented graphically, patterns begin to emerge.

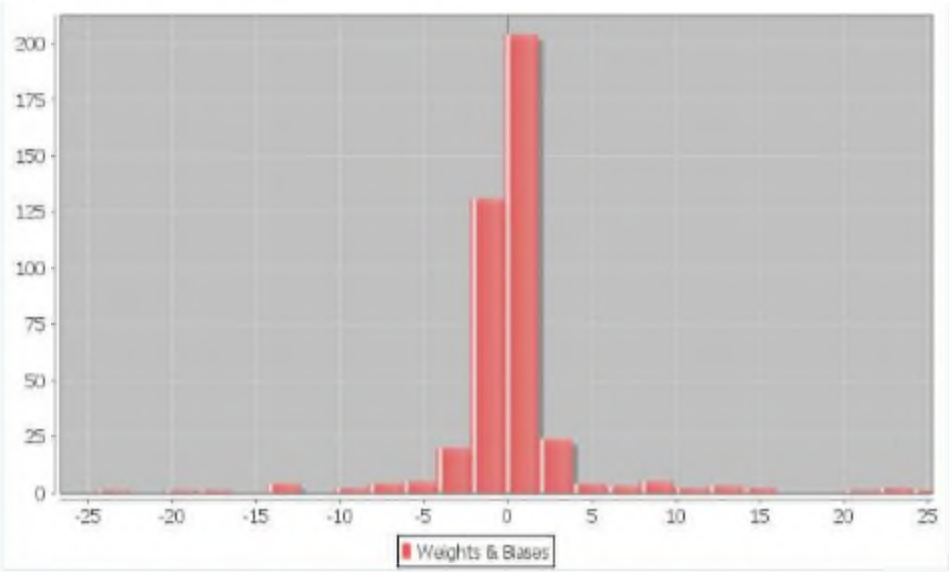
One common way to view the weights of a neural network is using a special type of chart called a histogram. You've probably seen histograms many times before – a histogram is a chart made up of vertical bars that count the number of occurrences in a population. Figure 6.1 is a histogram showing the popularity of operating systems. The y-axis shows the number of occurrences of each of the groups in the x-axis.

Figure 6.1: Histogram of OS Popularity (from Wikipedia)



We can use a histogram to look at the weights of a neural network. You can typically tell a trained from an untrained neural network by looking at this histogram. Figure 6.2 shows a trained neural network.

Figure 6.2: A Trained Neural Network



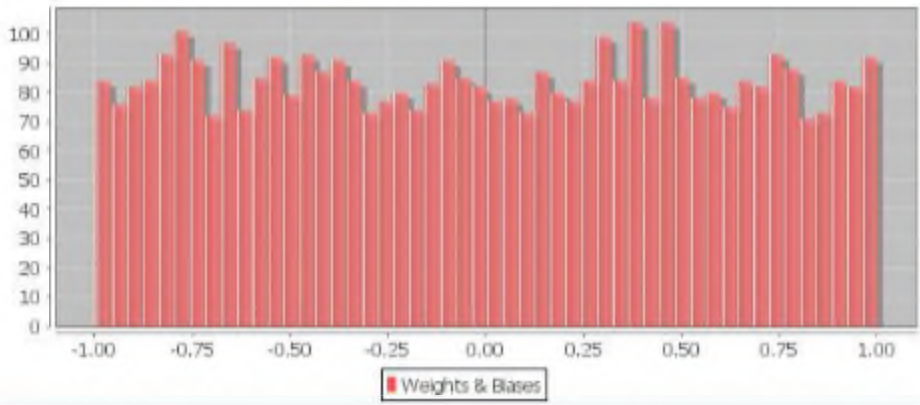
A neural network histogram uses the same concept as the operating system histogram shown earlier. The y-axis specifies how many weights fell into the ranges specified by the numbers on the x-axis. This allows you to see the distribution of the weights.

Most trained neural networks will look something like the above chart. Their weights will be very tightly clustered around zero. A trained neural network will typically look like a very narrow Gaussian curve.

Range Randomization

In the last section, we saw what a trained neural network looks like in a weight histogram. Untrained neural networks can have a variety of appearances. The appearance of the weight histogram will be determined by the weight initialization method used.

Figure 6.3: A Ranged Randomization

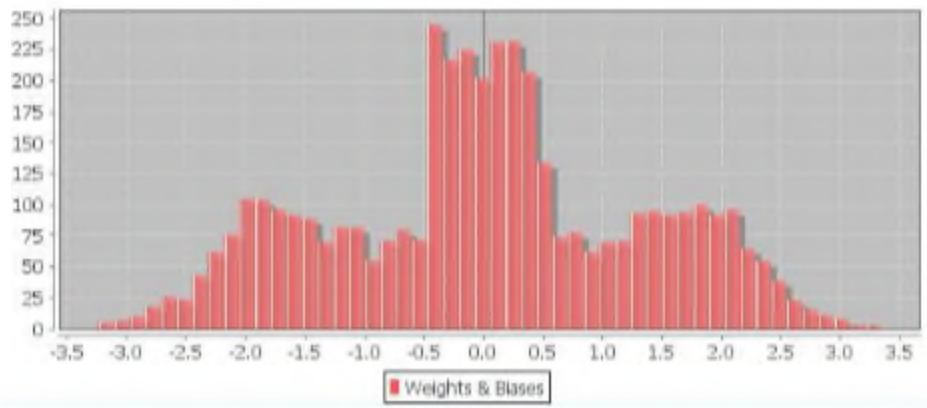


Range randomization produces a very simple looking chart. The more weights there are, the flatter the top will be. This is because the random number generator should give you an even distribution of numbers. If you are randomizing to the range of -1 to 1, you would expect to have approximately the same number of weights above zero as below.

Using Nguyen-Widrow

We will now look at the Nguyen-Widrow weight initialization method. The Nguyen-Widrow method starts out just like the range randomized method. Random values are chosen between -0.5 and +0.5. However, a special algorithm is employed to modify the weights. The histogram of a Nguyen-Widrow weight initialization looks like Figure 6.4.

Figure 6.4: The Nguyen-Widrow Initialization



As you can see, the Nguyen-Widrow initialization has a very distinctive pattern. There is a large distribution of weights between -0.5 and 0.5. It gradually rises and then rapidly falls off to around -3.0 and +3.0.

Performance of Nguyen-Widrow

You may be wondering how much advantage there is to using Nguyen-Widrow. Take a look at the average number of training iterations needed to train a neural network initialized by range randomization and Nguyen-Widrow.

```
Average iterations needed (lower is better)
Range random: 502.86
Nguyen-Widrow: 454.88
```

As you can see from the above information, the Nguyen-Widrow outperforms the range randomizer.

Implementing Nguyen-Widrow

The technique was invented by Derrick Nguyen and Bernard Widrow. It was first introduced in their paper, "Improving the learning speed of 2-layer

neural networks by choosing initial values of the adaptive weights", in the Proceedings of the International Joint Conference on Neural Networks, 3:21-26, 1990.

To implement an Nguyen-Widrow randomization, first initialize the neural network with random weight values in the range -0.5 to +0.5. This is exactly the same technique as was described earlier in this chapter for the ranged random numbers.

The Nguyen-Widrow randomization technique is efficient because it assigns each hidden neuron to a range of the problem. To do this, we must map the input neurons to the hidden neurons. We calculate a value, called beta, that establishes these ranges. You can see the calculation of beta in Equation 6.1.

Equation 6.1: Calculation of Beta

$$\beta = 0.7h^{\frac{1}{i}}$$

The variable **h** represents the number of hidden neurons in the first hidden layer, whereas the variable **i** represents the number of input neurons. We will calculate the weights, taking each hidden neuron one at a time. For each hidden neuron, we calculate the Euclidean norm for of all inputs to the current hidden neuron. This is done using Equation 6.2.

Equation 6.2: Calculation of the Euclidean Norm

$$n = \sqrt{\sum_{i=0}^{i<w_{inputs}} w_i^2}$$

Beta will stay the same for every hidden neuron. However, the norm must be recalculated for each hidden neuron. Once the beta and norm values have been calculated, the random weights can be adjusted. The equation below shows how weights are adjusted using the previously calculated values.

Equation 6.3: Updating the Weights

$$w_{t+1} = \frac{\beta w_t}{n}$$

All inbound weights to the current hidden neuron are adjusted using the same norm. This process is repeated for each hidden neuron.

You may have noticed that we are only specifying how to calculate the weights between the input layer and the first hidden layer. The Nguyen-Widrow method does not specify how to calculate the weights between a hidden layer and the output. Likewise, the Nguyen-Widrow method does not specify how to calculate the weights between multiple hidden layers. All weights outside of the

first layer and first hidden layer are simply initialized to a value between -0.5 and +0.5.

Nguyen-Widrow in Action

We will now walk through the random weight initialization for a small neural network. We will look at a neural network that has two input neurons and a single output neuron. There is a single hidden layer with two neurons. Bias neurons are present on the input and hidden layers.

We begin by initializing the weights to random numbers in the range -0.5 to +0.5. The starting weights are shown here.

```
Weight 0: H1->O1: 0.23773012320107711
Weight 1: H2->O1: 0.2200753094723884
Weight 2: B2->O1: 0.12169691073037914
Weight 3: I1->H1: 0.5172524211645029
Weight 4: I2->H1: -0.5258712726818855
Weight 5: B1->H1: 0.8891383322123643
Weight 6: I1->H2: -0.007687742622070948
Weight 7: I2->H2: -0.48985643968339754
Weight 8: B1->H2: -0.6610227585583137
```

First we must calculate beta, which is given in Equation 6.1. This calculation is shown here.

```
Beta = 0.7 * ( hiddenNeurons ^ (1.0/inputCount)
```

Filling in the variables, we have:

```
Beta = 0.7 * ( 2.0 ^ (1.0/2.0)) = 0.9899
```

We are now ready to modify the weights. We will only modify weights three through eight. Weights zero through two are not covered by the Nguyen-Widrow algorithm, and are simply set to random values between -0.5 and 0.5.

The weights will be recalculated in two phases. First, we will recalculate all of the weights from the bias and input neurons to hidden neuron one. To do this, we must calculate the Euclidean norm, or magnitude, for hidden neuron one. From Equation 6.2, we have the following:

```
Norm Hidden 1 = sqrt( (weight 3)^2 + (weight 4)^2 + (weight 5)^
```

Filling in the weights, we get:

```
Norm Hidden 1 = sqrt((0.5172^2) + (-
0.5258)^2 + (0.8891^2)) = 1.155
```

We will now look at how to calculate the first weight:

```
New Weight =( beta * Old Weight)/Norm Hidden 1
```

Filling in values, we have:

```
New Weight =( 0.9899 * 0.5172) / 1.155 = 0.4432
```

All three weights feeding the first hidden neuron are transformed in this way.

```
0.5172524211645029 -> 0.44323151482681195 (as just seen)
-0.5258712726818855 -> -0.4506169739523903
0.8891383322123643 -> 0.7618990530577658
```

We now repeat the same process for neuron two. The value for beta stays the same. However, the magnitude must be recalculated. The recalculated magnitude for neuron two is given here:

```
Neuron Two Magnitude = 0.8227815750355376
```

Using this, we can now recalculate weights six through eight.

```
-0.007687742622070948 -> -0.009249692928269318
-0.48985643968339754 -> -0.5893825884595142
-0.6610227585583137 -> -0.79532547274779
```

This results in the final values for all of the weights:

```
Weight 0: H1->O1: 0.23773012320107711
Weight 1: H2->O1: 0.2200753094723884
Weight 2: B2->O1: 0.12169691073037914
Weight 3: I1->H1: 0.44323151482681195
Weight 4: I2->H1: -0.4506169739523903
Weight 5: B1->H1: 0.7618990530577658
Weight 6: I1->H2: -0.009249692928269318
Weight 7: I2->H2: -0.5893825884595142
Weight 8: B1->H2: -0.79532547274779
```

These weights, though still random, utilize the hidden neurons better. These are the weights that we will begin training with.

Chapter Summary

In this chapter, we saw how the weights of a neural network are initialized. These weights must be set to random values. If the weights were all set to zero, the neural network would never train properly. There are several different ways in which the weights of a neural network are commonly initialized.

The first is range randomization. This process initializes each weight of the neural network to a random value in a specific range. The range used is typically -1 to 1 or 0 to 1. Though range randomization can provide very good results for neural network training, there are better methods.

Nguyen-Widrow weight initialization can provide significantly better training times than simple range randomization. The Nguyen-Widrow randomization technique is efficient because it assigns each hidden neuron to a range of the problem.

So far, all training presented has focused on attempting to minimize the error. In the next chapter, we will look the Levenberg Marquardt (LMA) training algorithm. LMA can often train in fewer iterations than resilient propagation (RPROP).

Chapter 7: LMA Training

- Newton's Method
- Calculating the Hessian
- The Levenberg-Marquardt Algorithm
- Multiple Output Neurons

The Levenberg–Marquardt algorithm (LMA) is a very efficient training method for neural networks. In many cases, LMA will outperform RPROP. Because of this, it is an important training algorithm that should be considered by any neural network programmer.

LMA is a hybrid algorithm that is based both on Newton's Method (GNA) and on gradient descent (backpropagation). This allows LMA to combine the strengths of both. Gradient descent is guaranteed to converge to a local minimum, but it is quite slow. Newton's Method is quite fast but often fails to converge. By using a damping factor to interpolate between the two, a hybrid method is created. To understand how this works, we will first examine Newton's method. Newton's Method is shown in Equation 7.1.

Equation 7.1: Newton's Method (GNA)

$$W_{min} = W_0 - H^{-1} g$$

You will notice several variables in the equation above. The result of the equation is deltas that can be applied to the weights of the neural network. The variable **H** represents the Hessian. This will be covered in the next section. The variable **g** represents the gradients of the neural network. You will also notice the -1 "exponent" on the variable **H**. This specifies that we are doing a matrix decomposition of the variables **H** and **g**.

We could easily spend an entire chapter on matrix decomposition. But instead, in this book, we will simply treat matrix decomposition as a black box atomic operator. Much as if I tell you to take the square root of a number, I do not explain how to calculate a square root – I simply assume that you will use the square root function in your programming language of choice. There is a common piece of code for matrix decomposition included in the JAMA package. This public domain code, which was adapted from a FORTRAN program, has been used in many mathematical computer applications. You can use JAMA or another source to perform matrix decomposition.

There are several types of matrix decomposition. The decomposition that we are going to use is the LU decomposition. This decomposition requires a square matrix. This works well because the Hessian matrix has the same number

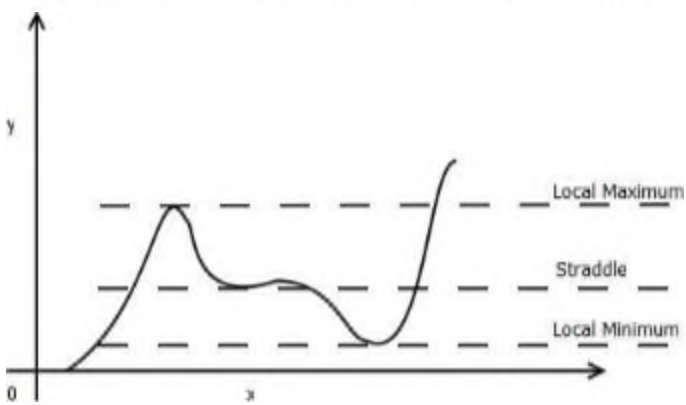
of rows as columns. There is a row and column for every weight in the neural network. The LU decomposition takes the Hessian, which is a matrix of the second derivatives of the partial derivatives of the output of each of the weights. The LU decomposition solves the Hessian by the gradients. These gradients are the square of the error of each weight. These are the same gradients that we calculated in Chapter 4, except they are squared. Because the errors are squared, we are required to use the sum of square error when dealing with LMA.

If you have never heard the term "second derivative" before, the second derivative is the derivative of the first derivative. Recall from Chapter 3 that the derivative of a function is the slope at any point. This slope points in the direction that the curve is approaching a local minimum. The second derivative is also a slope. It points in a direction to minimize the first derivative. The goal of Newton's method, as well as of the LMA algorithm, is to reduce all of the gradients to zero.

Interestingly, this goal does not include the error. Newton's Method and LMA can seem to be oblivious to the error, because they simply seek to squash all of the gradients to zero. In reality, they are not completely oblivious to the error, as the error is used to calculate the gradients.

Newton's Method will converge the weights of a neural network to a local minimum, a local maximum or a straddle position. This is done by minimizing all of the gradients (first derivatives) to zero. The derivatives will all be zero at local minima, maxima or straddle position. These three points are shown in Figure 7.1.

Figure 7.1: Local Minimum, Straddle and Local Maximum



It is up to the algorithm implementation to ensure that local maximums and straddle points are filtered out. The above algorithm works by taking the matrix decomposition of the Hessian matrix and the gradients. The Hessian matrix is

typically estimated. There are various means of doing this. If the Hessian is inaccurate, it can greatly throw off Newton's Method.

LMA enhances Newton's Algorithm to the following formula:

Equation 7.2: Levenberg–Marquardt Algorithm

$$W_{min} = W_0 - (H + \lambda I)^{-1} g$$

Here we add a damping factor multiplied by an identity matrix. Lambda is the damping factor and "I" represents the identity matrix. An identity matrix is a square matrix with all zeros except for a NW line of ones. As lambda increases, the Hessian will be factored out of the above equation. As lambda decreases, the Hessian becomes more significant than gradient descent. This allows the training algorithm to interpolate between gradient descent and Newton's Method. Higher lambda favors gradient descent, lower lambda favors Newton. A training iteration of LMA begins with a low lambda and increases it until a desirable outcome is produced.

Calculation of the Hessian

The Hessian matrix is a square matrix with rows and columns equal to the number of weights in the neural network. Each cell in this matrix represents the second order derivative of the output of the neural network with respect to a given weight combination. The Hessian is shown in Equation 7.3.

Equation 7.3: The Hessian Matrix

$$H(e) = \begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 e}{\partial w_n \partial w_1} & \frac{\partial^2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix}$$

It is important to note that the Hessian is symmetrical about the diagonal. This can be used to enhance performance of the calculation. You can calculate the Hessian by calculating the gradients.

Equation 7.4: Calculating the Gradients

$$\frac{\partial E}{\partial w_{(i)}} = 2(y - t)\frac{\partial y}{\partial w_{(i)}}$$

The second derivative of the above equation becomes an element of the Hessian matrix. This is calculated using the following formula:

Equation 7.5: Calculating the Exact Hessian

$$\frac{\partial^2 E}{\partial w_i w_j} = 2\left(\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j} + (y - t)\frac{\partial^2 y}{\partial w_j \partial w_j}\right)$$

The above formula could be easily calculated if not for the second component. This component involves the second partial derivative and is difficult to calculate. This component is actually not that important and can be dropped. For an individual training case, it might be very important. However, the second component is multiplied by the error of that training case. We assume that the errors in a training set are independent and evenly distributed about zero. On an entire training set, they should essentially cancel each other out. This is not a perfect assumption. However, we only seek to approximate the Hessian.

This results in the following equation:

Equation 7.6: Approximating the Exact Hessian

$$\frac{\partial^2 E}{\partial w_i w_j} = 2\left(\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j}\right)$$

Using the above equation is, of course, only an approximation of the true Hessian. However, the simplification of the algorithm to calculate the second derivative is well worth the loss in accuracy. Any loss in accuracy will most likely be accounted for with an increase in lambda.

To calculate the Hessian and gradients, we must calculate the partial first derivatives of the output of the neural network. Once we have these partial first derivatives, the above equations allow us to easily calculate the Hessian and gradients.

Calculation of the first derivatives of the output of the neural network is very similar to the process that we used to calculate the gradients for backpropagation. The main difference is that this time, the derivative of the output is taken. In standard backpropagation, the derivative of the error function is taken. We will not review the entire backpropagation process here. Chapter 4 covers backpropagation and gradient calculation.

LMA with Multiple Outputs

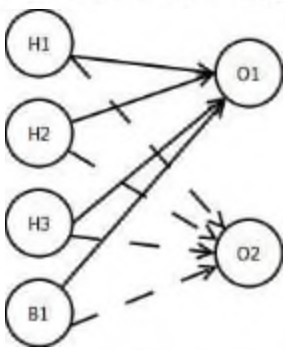
Some implementations of the LMA algorithm only support a single output neuron. This is primarily because the LMA algorithm has its roots in mathematical function approximation. In mathematics, functions typically only return a single value. As a result, there is not a great deal of information on support for multiple neurons.

Support for multiple output neurons involves simply summing each cell of the Hessian as the additional output neurons are calculated. It is as if you calculated a separate Hessian matrix for each output neuron, and then summed the Hessian matrices together. This is the approach that Encog uses, and it leads to very fast convergence times.

One important aspect to consider with multiple outputs is that not every connection will be used. Depending on which output neuron you are currently calculating for, there will be unused connections for the other output neurons. It is very important that the partial derivative for each of these unused connections be set to zero when the other output neuron is being calculated for.

For example, consider a neural network that has two output neurons and three hidden neurons. Each of these two output neurons would have a total of four connections from the hidden layer. There would be three from the three hidden neurons, and a fourth for the bias neuron. This segment of the neural network would look like Figure 7.2.

Figure 7.2: Calculating Output Neuron 1



Here we are calculating output neuron one. Notice that there are also four connections for output neuron two? It is critical that the derivatives of the four connections to output neuron two calculate to zero when output neuron one is calculated. If this process is not followed, the Hessian will not work for multiple outputs.

Overview of the LMA Process

So far, we have only seen the math behind LMA. LMA must be part of an algorithm for it to be effective. The LMA process can be summarized in the following steps:

```
1. Calculate the first derivative of output of the neural netwo
2. Calculate the Hessian
3. Calculate the gradients of the error (ESS) with respect to e
4. Either set lambda to a low value (first iteration) or the la
5. Save the weights of the neural network
6. Calculate delta weight based on the lambda, gradients and He
7. Apply the deltas to the weights and evaluate error
8. If error has improved, end the iteration
9. If error has not improved increase lambda (up to a max lambd
```

As you can see, the process for LMA revolves around setting the lambda value low and then slowly increasing it if the error rate does not improve. It is important to save the weights at each change in lambda so that they can be restored if the error does not improve.

Chapter Summary

In this chapter, we found out how to use the Levenberg–Marquardt Algorithm (LMA) to train a neural network. The LMA algorithm is very efficient and will often outperform backpropagation and RPROP. LMA works by attempting to minimize all of the error gradients to zero.

LMA is a hybrid algorithm. LMA works by using both Newton's Method and regular gradient descent. Gradient descent is essentially the same thing as backpropagation. The beauty of LMA is that a damping factor, called lambda, is used to interpolate between gradient descent and Newton's Method. We start out favoring Newton's Method. But if the error fails to improve, we introduce more gradient descent learning.

So far, we have only looked at the feedforward neural network. In the next two chapters, we will introduce two new types of neural network. You will learn about the Self Organizing Map (SOM) and a Radial Basis Function neural network. Both types of neural network have a great deal in common with the feedforward neural networks you have seen in this book.

Chapter 8: Self-Organizing Maps

- SOM Structure
- Training a SOM
- Neighborhood Functions
- SOM Error Calculation

So far, this book has focused on the feedforward neural network. In this chapter, we will look at a different type of neural network. This chapter will focus on the Self-Organizing Map (SOM). Though the SOM can be considered a type of feedforward neural network, it is used very differently. A SOM is used for unsupervised classification.

Self-Organizing Maps start with random weights, just like the neural networks we've seen so far. However, the means by which these weights are organized to produce meaningful output is very different than a feedforward neural network. SOMs make use of unsupervised training.

Unsupervised training works very differently from the supervised training that we have been using up until now. Unsupervised training sets only specify the input to the neural network. There is no ideal output provided. Because of this, the SOM learns to cluster, or map, your data into a specified number of classes.

The number of output neurons in the SOM defines the number of output classes into which you would like to cluster the input data. The number of input neurons defines the attributes of each of the training set items based on which you want the neural network to cluster. This is exactly the same as with feedforward neural networks. You must present data to the neural network through a fixed number of input neurons.

SOMs are only used for classification, which means they divide the input into classes. SOMs are not generally used for regression. Regression is where the neural network predicts one or more numeric values. With a SOM, you simply provide the total number of classes, and the SOM automatically organizes your training data into these classes. Also, the SOM should be able to organize data that it was never trained with into these classes.

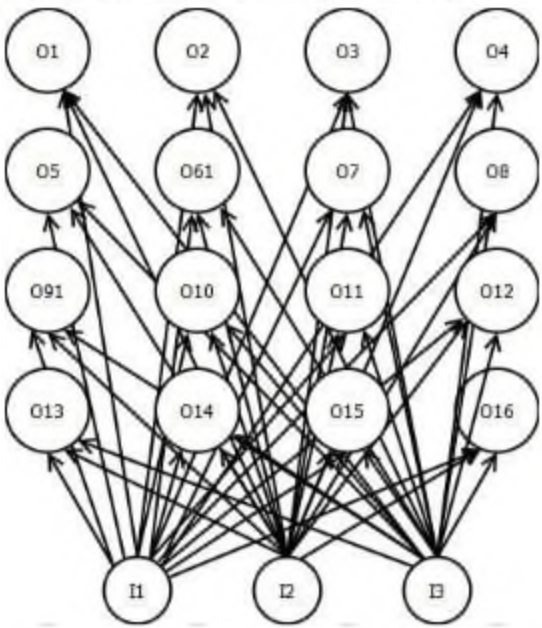
SOM Structure

The SOM has a structure somewhat similar to the neural networks that we have seen already in this book. However, there are some very important differences. Some of the differences between SOMs and traditional feedforward neural networks are shown here.

- SOM networks have no bias neurons
- SOM networks have no hidden layers
- SOM networks have no activation functions
- SOM networks must have more than one output neuron
- Output neurons in a 1D, 2D, 3D, etc lattice

Consider a SOM that has six input neurons and three output neurons. This SOM is designed to classify data into three groups. The incoming data items that will be classified each have six values. This SOM can be seen in Figure 8.1.

Figure 8.1: A 2D SOM with 16 Outputs and 3 Inputs



As you can see from the above diagram, there are no hidden layers and there are no bias neurons. You will also notice that the output neurons are arranged in a lattice. In this case, the output neurons are arranged in a 2D grid. This grid will become very important when the neural network is trained. Neurons near each other on the grid will be trained together. The grid does not

need to be 2D. Grids can be 1D, 3D or of an even higher number of dimensions. The topology of this grid is defined by the neighborhood function that is used to train the neural network. Neighborhood functions will be described later in this chapter.

A SOM can be thought of as reducing a high number of dimensions to a lower number of dimensions. The higher number of dimensions is provided by the input neurons, using one dimension for each input neuron. The lower number of dimensions is the configuration of the output neuron lattice. In the case of Figure 8.1, we are reducing three dimensions to two.

In Chapter 1, we saw how a feedforward neural network calculates its output. The method by which a SOM calculates its output is very different. The output of a SOM is the "winning" output neuron for a given input. This winning neuron is also called the Best Matching Unit or BMU. We will find out how to calculate the BMU in the next section.

Best Matching Unit

The Best Matching Unit is the output neuron whose weights most closely match the input being provided to the neural network. Consider the SOM shown in Figure 8.1. There are 16 output neurons and 3 input neurons. This means that each of the 16 output neurons has 3 weights, one from each of the input neurons. The input to the SOM would also be three numbers, as there are three input neurons. To determine the BMU, we find the output neuron whose three weights most closely match the three input values fed into the SOM.

It is easy enough to calculate the BMU. To do this, we loop over every output neuron and calculate the Euclidean distance between the output neuron's weights and the input values. The output neuron that has the lowest Euclidean distance is the BMU. The Euclidean distance is simply the distance between two multi-dimensional points. If you are dealing with two dimensions, the Euclidean distance is simply the length of a line drawn between the two points.

The Euclidean distance is used often in Machine Learning. It is a quick way to compare two arrays of numbers that have the same number of elements. Consider three arrays, named array **a**, array **b** and array **c**. The Euclidean distance between array **a** and array **b** is 10. The Euclidean distance between array **a** and array **c** is 20. In this case, the contents of array **a** more closely match array **b** than they do array **c**.

Equation 8.1 shows the formula for calculating the Euclidean distance.

Equation 8.1: The Euclidean Distance

$$\mathrm{d}(\mathbf{p}, \mathbf{q}) = \mathrm{d}(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

The above equation shows us the Euclidean distance **d** between two arrays **p** and **q**. The above equation also states that **d(p,q)** is the same as **d(q,p)**. This simply means that the distance is the same no matter which end you start at. Calculation of the Euclidean distance is no more than summing the squares of the difference of each array element. Finally, the square root of this sum is taken. This square root is the Euclidean distance. The output neuron with the lowest Euclidean distance is the BMU.

Training a SOM

In the previous chapters, we learned several methods for training a feedforward neural network. We learned about such techniques as backpropagation, RPROP and LMA. These are all supervised training methods. Supervised training methods work by adjusting the weights of a neural network to produce the correct output for a given input.

A supervised training method will not work for a SOM. SOM networks require unsupervised training. In this section, we will learn to train a SOM with an unsupervised method. The training technique generally used for SOM networks is shown in Equation 8.2.

Equation 8.2: Training a SOM

$$W_v(t+1) = W_v(t) + \theta(v,t)\alpha(t)(D(t) - W_v(t))$$

The above equation shows how the weights of a SOM neural network are updated as training progresses. The current training iteration is noted by the letter **t**, and the next training iteration is noted by **t+1**. Equation 8.2 allows us to see how weight **v** is adjusted for the next training iteration.

The variable **v** denotes that we are performing the same operation on every weight. The variable **W** represents the weights. The symbol **theta** is a special function, called a neighborhood function. The variable **D** represents the current training input to the SOM.

The symbol alpha denotes a learning rate. The learning rate changes for each iteration. This is why Equation 8.2 shows the learning rate with the symbol **t**, as the learning rate is attached to the iteration. The learning rate for a SOM is said to be monotonically decreasing. Monotonically decreasing means that the learning rate only falls, and never increases.

SOM Training Example

We will now find out how to train a SOM. We will apply the equation presented in the previous section. However, we will approach it more from an algorithm perspective so we can see the actual learning strategy behind the equation. To see the SOM in action, we need a very simple example. We will use a SOM very similar to the one that we saw in Figure 8.1. This SOM will attempt to match colors. However, instead of the 4x4 lattice we saw in Figure 8.1, we will have a lattice of 50x50. This results in a total of 2,500 output neurons.

The SOM will use its three input neurons to match colors to the 2,500

output neurons. The three input neurons will contain the red, blue and green components of the color that is currently being submitted to the SOM. For training, we will generate 15 random colors. The SOM will learn to cluster these colors.

This sort of training is demonstrated in one of the Encog examples. You can see the output from this example program in Figure 8.2.

Figure 8.2: Mapping Colors

As you can see from the above figure, similar colors are clustered together. Additionally, there are 2,500 output neurons, and only 15 colors that were trained with. This network could potentially recognize up to 2,500 colors. The fact that we trained with only 15 colors means we have quite a few unutilized output neurons. These output neurons will learn to recognize colors that are close to the 15 colors that we trained with.

What you are actually seeing in Figure 8.2 are the weights of SOM network that has been trained. As you can see, even though the SOM was only trained to recognize 15 colors, it is able to recognize quite a few more colors. Any new color provided to the SOM will be mapped to one of the 2,500 colors seen in the above image. The SOM can be trained to recognize more classes than are in its provided training data. This is definitely the case in Figure 8.2. The unused output neurons will end up learning to recognize data that falls between elements of the smaller training set.

Training the SOM Example

We will now look at how the SOM network is trained for the colors example. To begin with, all of the weights of the SOM network are randomized to values between -1 and +1. A training set is now generated for 15 random colors. Each of these 15 random colors will have three input values. Each of the three input values will be a random value between -1 and +1. For the red, green and blue values, -1 represents that the color is totally off, and +1 represents that the color is totally on.

We will see how the SOM is trained for just one of these 15 colors. The same process would be used for the remaining 14 colors. Consider if we were training the SOM for the following random color:

```
-1,1,0.5
```

We will see how the SOM will be trained with this training element.

BMU Calculation Example

The first step would be to compare this input against every output neuron in the SOM and find the Best Matching Unit (BMU). BMU was discussed earlier in this chapter. The BMU can be calculated by finding the smallest Euclidean distance in the SOM. The random weights in the SOM are shown here.

```
Output Neuron 1: -0.2423, 0.4837, 0.8723
Output Neuron 2: -0.5437, -0.8734, 0.2234
…
Output Neuron 2500: -0.1287, 0.9872, -0.8723
```

Of course, we are skipping quite a few of the output neurons. Normally, you would calculate the Euclidean distance for all 2,500 output neurons. Just calculating the Euclidean distance for the above three neurons should give you an idea of how this is done. Using Equation 8.1, we calculate the Euclidean distance between the input and neuron one.

```
sqrt((-0.2423 - -1)^2 + (0.4837-1)^2 + (0.8723-0.5)^2) = 0.9895
```

A similar process can be used to calculate neuron two.

```
sqrt( (-0.5437 - -1)^2 + (-0.8734-1)^2 + (0.2234-
0.5)^2 ) = 1.947
```

Similarly, we can also calculate neuron 2,500.

```
sqrt((-0.1287 - -1)^2 + (0.9872-1)^2 + (-0.8723-
```

```
0.5)^2) = 1.6255
```

Now that we have calculated all of the Euclidean distances, we can determine the BMU. The BMU is neuron one. This is because the distance of 0.9895 is the lowest. Now that we have a BMU, we can update the weights.

Example Neighborhood Functions

We will now loop over every weight in the entire SOM and use Equation 8.2 to update them. The idea is that we will modify the BMU neuron to be more like the training input. However, as well as modifying the BMU neuron to be more like the training input, we will also modify neurons in the neighborhood around the BMU to be more like the input as well. The further a neuron is from the BMU, the less impact this weight change will have.

Determining the amount of change that will happen to a weight is the job of the neighborhood function. Any radial basis function (RBF) can be used as a neighborhood function. A radial basis function is a real-valued function whose value depends only on the distance from the origin.

The Gaussian function is the most common choice for a neighborhood function. The Gaussian function is shown in Equation 8.3.

Equation 8.3: Gaussian Function

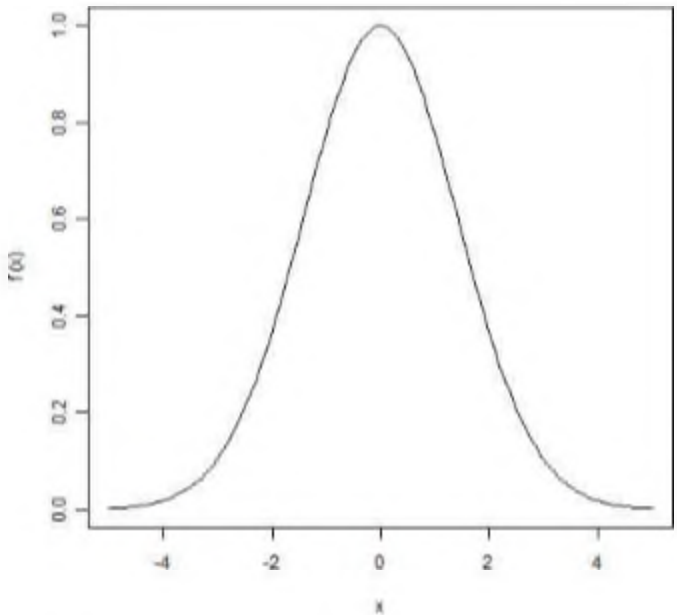
$$f(x_1, x_2, \ldots, x_n) = e^{-v},$$

This equation continues as follows:

$$v = \sum_{i=0}^{i<n} \frac{x_i - c^2}{2w^2}$$

You can see the Gaussian function graphed in Figure 8.3, where **n** is the number of dimensions, **c** is the center and **w** is the width of the Gaussian curve. The number of dimensions is equal to the number of input neurons. The width starts out at some fixed number and decreases as learning progresses. By the final training iteration, the width should be one.

Figure 8.3: Gaussian Function Graphed



From the above figure, you can see that the Gaussian function is a radial basis function. The value only depends on the distance from the origin. That is to say, **f(x)** has the same value regardless of whether **x** is -1 or +1.

Looking at Figure 8.3, you can see how the Gaussian function scales the amount of training received by each neuron. The BMU would have zero distance from itself, so the BMU would receive full training of 1.0. As you move further away from the BMU in either direction, the amount of training quickly falls off. A neuron that was -4 or +4 from the BMU would receive hardly any training at all.

The Gaussian function is not the only function available for SOM training. Another common choice is the Ricker wavelet or "Mexican hat" neighborhood function. This function is generally only known as the "Mexican Hat" function in the Americas, due to its resemblance to a "sombrero". In technical nomenclature, the function is known as the Ricker wavelet, and it is frequently employed to model seismic data. The equation for the Mexican Hat function is shown in Equation 8.4.

Equation 8.4: Mexican Hat Neighborhood function

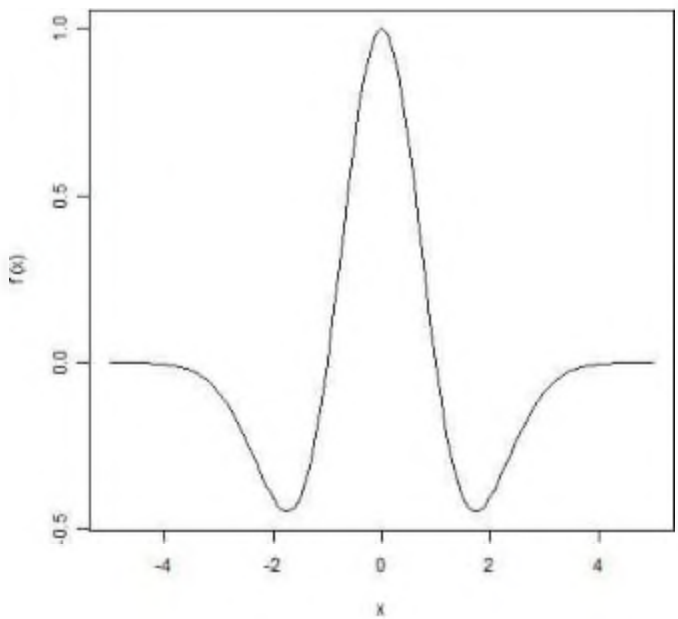
$$f(x_1, x_2, ..., x_n) = (1 - v)e^{-0.5v},$$

This equation continues with the following:

$$v = \sum_{i=0}^{i<n} (x - c)^2$$

You will see the value of the Mexican Hat neighborhood function when you examine its graph in Figure 8.4.

Figure 8.4: Graph of the Mexican Hat



Just as before, the BMU is at the origin of **x**. As you can see from the above chart, the Mexican Hat function actually punishes neurons just at the edge of the radius of the BMU. Their share of the learning will be less than zero. As you get even further from the BMU, the learning again returns to zero and remains at zero. However, neurons just at the edge will have negative learning applied. This can allow the SOM to make classes differentiate better among each other.

Example Weight Update

Now that we have seen how to calculate the BMU and the neighborhood function, we are finally ready to calculate the actual weight update. The formula for the weight update was given in Equation 8.2. We will now calculate the components of this equation for an actual weight update. Recall the weights

given earlier in this chapter. Neuron one had the following weights:

```
Output Neuron 1: -0.2423, 0.4837, 0.8723
```

Neuron one was the BMU when provided with the following input:

```
-1,1,0.5
```

As you can see, the BMU is somewhat similar to the input. During training, we now want to modify the BMU to be even more like the input. We will also modify the neighbors to receive some of this learning as well.

The "ultimate" learning is to simply copy the input into the weights. Then the Euclidean distance of the BMU will become zero and the BMU is perfectly trained for this input vector. However, we do not want to go that extreme. We simply want to move the weights in the direction of the input. This is where equation 8.2 comes in. We will use Equation 8.2 for every weight in the SOM, not just the BMU weights. For this example, we will start with the BMU. Calculating the update to the first weight, which is -1, we have:

$$w = -0.2423 + (N * r * (-1 \hat{a} \breve{A} \S (-0.2423)))$$

Before we calculate the above equation, we will take a look at what is actually happening. Look at the term at the far right. We are taking the difference between the input and the weight. As I said before, the ultimate is just to assign the input to the weight. If we simply added the last term to the weight, the weight would be the same as the input vector.

We do not want to be this extreme. Therefore, we scale this difference. First we scale it by the learning rate, which is the variable **r**. Then we scale it by the result of the neighborhood function, which is **N**. For the BMU, **N** is the value 1.0 and has no effect. If the learning rate were 1.0 as well, then the input would be copied to the weight. However, a learning rate is never above 1.0. Additionally, the learning rate typically decays as the learning iterations progress. Considering that we have a learning rate of 0.5, our weight update becomes:

$$w = -0.2423 + (1.0 * 0.5 * (-1 - -0.2423)) = -0.62115$$

As you can see, the weight moved from -0.2423 to -0.62115. This moved the weight closer to the input of -1. We can perform a similar update for the other two weights that feed into the BMU. This can be seen here:

$$w = 0.4837 + (1.0 * 0.5 * (1 - 0.4837)) = 0.74185 w = 0.8723 + (1.0 * 0.5 * (0.5 - 0.8723)) = 0.68615$$

As you can see, both weights moved closer to the input values.

The neighborhood function is always 1.0 for the BMU. However, consider if we were to calculate the weight update for neuron two, which is not the BMU. We would need to calculate the neighborhood function, which was given in

Equation 8.3. This assumes we are using a Gaussian neighborhood function. The Mexican Hat function could also be used.

The first step is to calculate **v**. Here we use a width **w** of 3.0. When using Gaussian for a neighborhood function, the center **c** is always 0.0.

$$v = ((x1 - 0)/2w^2) + ((x2 - 0)/2w^2)$$

We will plug in these values. The values **x1** and **x2** specify how far away the current neuron is from the BMU. The value for **x1** specifies the column distance, and the value of **x2** specifies the row distance. Because neuron two is on the same row, then **x2** will be zero. Neuron two is only one column forward of the BMU, so **x1** will be 1. This gives us the following calculation for **v**:

$$v = ((0 - 0)/(2 * 3)^2) + ((1 - 0)/(2 * 3)^2) = 0.0277$$

Now that **v** has been calculated, we can calculate the Gaussian.

$$exp(-v) = 0.9726$$

As you can see, a neuron so close to the BMU gets nearly all the training that the BMU received. Other than using the above value for the neighborhood function, the weight calculation for neuron two is the same as neuron one.

SOM Error Calculation

When training feedforward neural networks, we would typically calculate an error number to indicate whether training has been successful. A SOM is a unsupervised neural network. Because it is unsupervised, the error cannot be calculated by normal means. In fact, because it is unsupervised, there really is no error at all.

If there is no known, ideal data, then there is nothing to calculate an error against. It would be helpful to see some sort of number to indicate the progression of training.

Many neural network implementations do not even report an error for the SOM. Additionally, many articles and books about SOMs do not provide a way to calculate an error. As a result, there is no standard way to calculate an error for a SOM.

But there is a way to report an error. The error for a SOM can be thought of as the worst (or longest) Euclidean distance of any of the best matching units. This is a number that should decrease as training progresses. This can give you an indication of when your SOM is no longer learning. However, it is only an indication. It is not a true error rate. It is also more of a "distance" than an error percent.

Chapter Summary

This chapter introduced a new neural network and training method. The Self-Organizing Map automatically organizes data into classes. No ideal data is provided; the SOM simply groups the input data into similar classes. The number of grouping classes must remain constant.

Training a SOM requires you to determine the best matching unit (BMU). The BMU is the neuron whose weights most closely match the input that we are training. The weights of the BMU are adjusted to more closely match the input that we are training. Weights near the BMU are also adjusted.

In the next chapter, we will look how to prepare data for neural networks. Neural networks typically require numbers to be in specific ranges. In the next chapter we will see how to adjust numbers to fall in specific ranges. This process is called normalization.

Chapter 9: Normalization

- What is Normalization
- Reciprocal Normalization and Denormalization
- Range Normalization and Denormalization

The input to neural networks in this book has so far mostly been in either the range of -1 to +1 or 0 to +1. The XOR operator, for example, had input of either 0 or 1. In Chapter 8, we normalized colors to a range between -1 and +1. Neural networks generally require their input and output to be in either of these two ranges. The problem is that most real-world data is not in this range.

To account for this, we must convert real-world data into one of these ranges before that data is fed to the neural network. Additionally, we must convert data coming out of the neural network back to its normal numeric range. The process of converting real-world data to a specific range is called normalization.

You should normalize both input and ideal data. Because you are normalizing the ideal data, your neural network will be trained to return normalized data from the output neurons. Usually, you will want to convert the normalized output back to real-world numbers. The reverse of normalization is called denormalization. Both normalization and denormalization will be covered in this chapter.

Simple Normalization and Denormalization

We will begin by looking at a very simple means of normalization. This method is called reciprocal normalization. This normalization method supports both normalization and denormalization. However, reciprocal normalization is limited in that you cannot specify the range into which to normalize. Reciprocal normalization is always normalizing to a number in the range between -1 and 1.

Reciprocal Normalization

Reciprocal normalization is very easy to implement. It requires no analysis of the data. As you will see, with range normalization the entire data set must be analyzed prior to normalization. Equation 9.1 shows how to use reciprocal normalization.

Equation 9.1: Reciprocal Normalization

$$f(x) = \frac{1}{x}$$

To see Equation 9.1 in use, consider normalizing the number five.

```
f(5.0) = 1.0/5.0 = 0.2
```

As you can see, the number five has been normalized to 0.2. This is useful, as 5.0 is outside of the range of 0 to 1.

Reciprocal Denormalization

You will also likely need to denormalize the output from a neural network. It is very easy to denormalize a number that has been normalized reciprocally. This can be done with Equation 9.2.

Equation 9.2: Reciprocal Denormalization

$$f(x) = x^{-1}$$

To see Equation 9.2 in use, consider denormalizing the number 0.2.

```
f(0.2) = (0.2)^(-1) = 5.0
```

As you can see, we have now completed a round trip. We normalized 5.0 to 0.2, and then denormalized 0.2 back to 5.0.

Range Normalization and Denormalization

Range normalization is more advanced than simple reciprocal normalization. With range normalization, you are allowed to specify the range that you will normalize into. This allows you to more effectively utilize the range offered by your activation function. In general, if you are using the sigmoid activation function, you should use a normalization range of 0 to 1. If you are using the hyperbolic tangent activation function, you should use the range from -1 to 1.

Range normalization must know the high and low values for the data that it is to normalize. To do this, the training set must typically be analyzed to obtain a high and low value. If you are going to use additional data that is not contained in the training set, your data range must include this data as well. Once you choose the input data, range data outside of this range will not produce good results. Because of this, it may be beneficial to enlarge the analyzed range by some amount to account for the true range.

Range Normalization

Once you have obtained the high and low values of your data set, you are ready to normalize. Equation 9.3 can be used to perform the range normalization.

Equation 9.3: Range Normalization

$$f(x) = \frac{(x-d_L)(n_H-n_L)}{(d_H-d_L)} + n_L$$

The above equation uses several constants. The constant **d** represents the high and low of the data to be normalized. The constant **n** represents the high and low of the range that we are to normalize into.

Consider if we were to normalize into the range of -1 to 1. The data range is 10 to 30. The number we would like to normalize is 12. Plugging in numbers, we have:

```
(((12-10)*(1- (-1)))/(30-10))+(-1) = -0.8
```

As you can see, the number 12 has been normalized to -0.8.

Range Denormalization

We will now look at how to denormalize a ranged number. Equation 9.3 will perform this denormalization.

Equation 9.4: Range Denormalization

$$f(x) = \frac{(d_L - d_H)x - (n_H \cdot d_L) + d_H \cdot n_L}{(n_L - n_H)}$$

Plugging in the numbers to denormalize -0.8, we are left with the following:

```
((10-30) * -0.8 - 1 * 10 + 30 * -1)/ (-1 - 1) = 12
```

Again we have made a round trip. We normalized 12 to -0.8. We then denormalized -0.8 back to 12.

Chapter Summary

This chapter covered normalization. Normalization is the process by which data is forced to conform to a specific range. The usual range is either -1 to +1, or 0 to 1. The range you will choose is usually dependent on the activation function you are using. This chapter covered two different types of normalization.

Reciprocal normalization is a very simple normalization technique. This normalization technique normalizes numbers to the range -1 to 1. Reciprocal normalization simply takes the reciprocal normalization and divides the number to normalize by 1.

Range normalization is more complex. However, range normalization allows you to normalize to any range you like. Additionally, range normalization must know the range of the input data. While this does allow you to make use of the entire normalization range, it also means that the entire data set must be analyzed ahead of time.

This chapter completes this book. I hope you have learned more about the lower levels of neural networks, as well as about some of the math behind them. If you would like to apply code to these techniques, you may find my books "Introduction to Neural Networks for Java" or "Introduction to Neural Networks for C#" useful. These books focus less on the mathematics and more on how to implement these techniques in Java or C#.

Table of Contents

Copyright Info
Front Matter
Introduction
Chapter 1: Neural Network Activation
Chapter 2: Error Calculation Methods
Chapter 3: Derivatives
Chapter 4: Backpropagation
Chapter 5: Faster Training with RPROP
Chapter 6: Weight Initialization
Chapter 7: LMA Training
Chapter 8: Self-Organizing Maps
Chapter 9: Normalization