CHAPTER 5: FEEDFORWARD NEURAL NETWORKS

- Introducing the Feedforward Backpropagation Neural Network
- Understanding the Feedforward Algorithm
- Implementing the Feedforward Algorithm
- Understanding the Backpropagation Algorithm
- Implementing the Backpropagation Algorithm

In this chapter we shall examine one of the most common neural network architectures, the feedforword backpropagation neural network. This neural network architecture is very popular, because it can be applied to many different tasks. To understand this neural network architecture, we must examine how it is trained and how it processes a pattern.

The first term, "feedforward" describes how this neural network processes and recalls patterns. In a feedforward neural network, neurons are only connected foreword. Each layer of the neural network contains connections to the next layer (for example, from the input to the hidden layer), but there are no connections back. This differs from the Hopfield neural network that was examined in chapter 3. The Hopfield neural network was fully connected, and its connections are both forward and backward. Exactly how a feedforward neural network recalls a pattern will be explored later in this chapter.

The term "backpropagation" describes how this type of neural network is trained. Backpropagation is a form of supervised training. When using a supervised training method, the network must be provided with both sample inputs and anticipated outputs. The anticipated outputs are compared against the actual outputs for given input. Using the anticipated outputs, the backpropagation training algorithm then takes a calculated error and adjusts the weights of the various layers backwards from the output layer to the input layer. The exact process by which backpropagation occurs will also be discussed later in this chapter.

The backpropagation and feedforward algorithms are often used together; however, this is by no means a requirement. It would be quite permissible to create a neural network that uses the feedforward algorithm to determine its output and does not use the backpropagation training algorithm. Similarly, if you choose to create a neural network that uses backpropagation training methods, you are not necessarily limited to a feedforward algorithm to determine the output of the neural network. Though such cases are less common than the feedforward backpropagation neural network, examples can be found. In this book, we will examine only the case in which the feedforward and backpropagation algorithms are used together. We will begin this discussion by examining how a feedforward neural network functions.

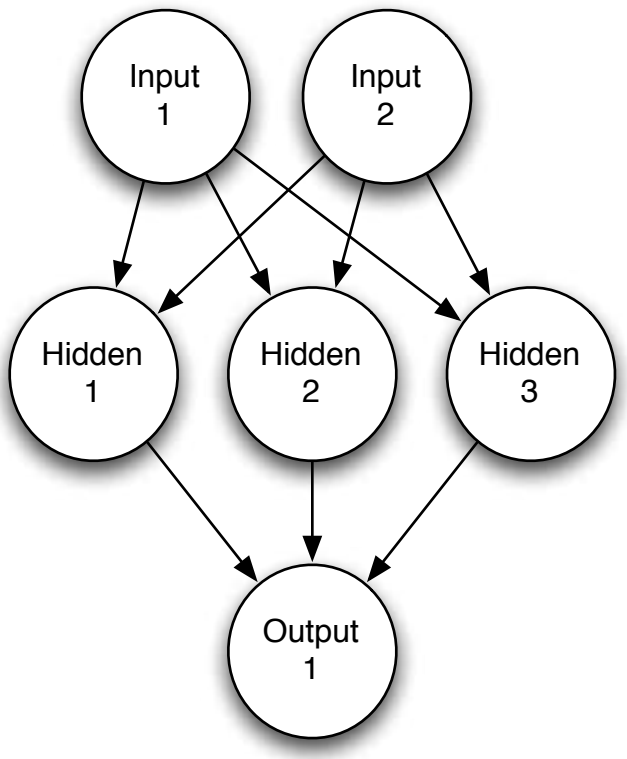
A Feedforward Neural Network

A feedforward neural network is similar to the types of neural networks that we have already examined. Just like many other types of neural networks, the feedforward neural network begins with an input layer. The input layer may be connected to a hidden layer or directly to the output layer. If it is connected to a hidden layer, the hidden layer can then be connected to another hidden layer or directly to the output layer. There can be any number of hidden layers, as long as there is at least one hidden layer or output layer provided. In common use, most neural networks will have one hidden layer, and it is very rare for a neural network to have more than two hidden layers.

The Structure of a Feedforward Neural Network

Figure 5.1 illustrates a typical feedforward neural network with a single hidden layer.

Figure 5.1: A typical feedforward neural network (single hidden layer).



Neural networks with more than two hidden layers are uncommon.

Choosing Your Network Structure

As we saw in the previous section, there are many ways that feedforward neural networks can be constructed. You must decide how many neurons will be inside the input and output layers. You must also decide how many hidden layers you are going to have and how many neurons will be in each of them.

There are many techniques for choosing these parameters. In this section we will cover some of the general "rules of thumb" that you can use to assist you in these decisions; however, these rules will only take you so far. In nearly all cases, some experimentation will be required to determine the optimal structure for your feedforward neural network. There are many books dedicated entirely to this topic. For a thorough discussion on structuring feedforward neural networks, you should refer to the book Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks (MIT Press, 1999).

The Input Layer

The input layer is the conduit through which the external environment presents a pattern to the neural network. Once a pattern is presented to the input layer, the output layer will produce another pattern. In essence, this is all the neural network does. The input layer should represent the condition for which we are training the neural network. Every input neuron should represent some independent variable that has an influence over the output of the neural network.

It is important to remember that the inputs to the neural network are floating point numbers. These values are expressed as the primitive Java data type "double." This is not to say that you can only process numeric data with the neural network; if you wish to process a form of data that is non-numeric, you must develop a process that normalizes this data to a numeric representation. In chapter 12, "OCR and the Self-Organizing Map," I will show you how to communicate graphic information to a neural network.

The Output Layer

The output layer of the neural network is what actually presents a pattern to the external environment. The pattern presented by the output layer can be directly traced back to the input layer. The number of output neurons should be directly related to the type of work that the neural network is to perform.

To determine the number of neurons to use in your output layer, you must first consider the intended use of the neural network. If the neural network is to be used to classify items into groups, then it is often preferable to have one output neuron for each group that input items are to be assigned into. If the neural network is to perform noise reduction on a signal, then it is likely that the number of input neurons will match the number of output neurons. In this sort of neural network, you will want the patterns to leave the neural network in the same format as they entered.

For a specific example of how to choose the number of input neurons and the number of output neurons, consider a program that is used for optical character recognition (OCR), such as the program presented in the example in chapter 12, "OCR and the Self-Organizing Map." To determine the number of neurons used for the OCR example, we will first consider the input layer. The number of input neurons that we will use is the number of pixels that might represent any given character. Characters processed by this program are normalized to a universal size that is represented by a 5x7 grid. A 5x7 grid contains a total of 35 pixels. Therefore, the OCR program has 35 input neurons.

The number of output neurons used by the OCR program will vary depending on how many characters the program has been trained to recognize. The default training file that is provided with the OCR program is used to train it to recognize 26 characters. Using this file, the neural network will have 26 output neurons. Presenting a pattern to the input neurons will fire the appropriate output neuron that corresponds to the letter that the input pattern represents.

Solving the XOR Problem

Next, we will examine a simple neural network that will learn the XOR operator. The XOR operator was covered in chapter 1. You will see how to use several classes from this neural network. These classes are provided in the companion download available with the purchase of this book. Appendix A describes how to obtain this download. Later in the chapter, you will be shown how these classes were constructed.

Listing 5.1: The XOR Problem (XOR.java)

```
package com.heatonresearch.book.introneuralnet.ch5.xor;

import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardLayer;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.Train;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.backpropagation.Backpropagation;
```

```
/**
 * Chapter 5: The Feedforward Backpropagation Neural Network
 *
 * XOR: Learn the XOR pattern with a feedforward neural network
 * that uses backpropagation.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class XOR {

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

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

      public static void main(final String args[]) {
            final FeedforwardNetwork network =
                  new FeedforwardNetwork();
            network.addLayer(new FeedforwardLayer(2));
            network.addLayer(new FeedforwardLayer(3));
            network.addLayer(new FeedforwardLayer(1));
            network.reset();

            // train the neural network
            final Train train = new Backpropagation(network,
                        XOR_INPUT, XOR_IDEAL,
                        0.7, 0.9);

            int epoch = 1;

            do {
                  train.iteration();
                  System.out
                              .println("Epoch #" + epoch
                        + " Error:" + train.getError());
                  epoch++;
            } while ((epoch < 5000) &&
                  (train.getError() > 0.001));

            // test the neural network
            System.out.println("Neural Network Results:");
            for (int i = 0; i < XOR_IDEAL.length; i++) {
                  final double actual[] = network.
```

```
computeOutputs(XOR_INPUT[i]);
                    System.out.println(XOR_INPUT[i][0] + ","
                    + XOR_INPUT[i][1]
                    + ", actual=" + actual[0] + ",ideal="
                    + XOR_IDEAL[i][0]);
            }
      }
}
```

The last few lines of output from this program are shown here.

```
Epoch #4997 Error:0.006073963240271441 Epoch #4998 Er-
ror:0.006073315333403568 Epoch #4999 Error:0.0060726676304029325
Neural Network Results: 0.0,0.0, actual=0.0025486129720869773,ide
al=0.0 1.0,0.0, actual=0.9929280525379659,ideal=1.0 0.0,1.0, actu
al=0.9944310234678858,ideal=1.0 1.0,1.0, actual=0.007745179145434
604,ideal=0.0
```

As you can see, the network ran through nearly 5,000 training epochs. This produced an error of just over half a percent and took only a few seconds. The results were then displayed. The neural network produced a number close to zero for the input of 0,0 and 1,1. It also produced a number close to 1 for the inputs of 1,0 and 0,1.

This program is very easy to construct. First, a two dimensional **double** array is created that holds the input for the neural network. These are the training sets for the neural network.

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

Next, a two dimensional **double** array is created that holds the ideal output for each of the training sets given above.

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

It may seem as though a single dimensional **double** array would suffice for this task. However, neural networks can have more than one output neuron, which would produce more than one **double** value. This neural network has only one output neuron.

A **FeedforwardNetwork** object is now created. This is the main object for the neural network. Layers will be added to this object.

```
final FeedforwardNetwork network =
new FeedforwardNetwork();
```

The first layer to be added will be the input layer. A **FeedforwardLayer** object is created. The value of two specifies that there will be two input neurons.

```
network.addLayer(new FeedforwardLayer(2));
```

The second layer to be added will be a hidden layer. If no additional layers are added beyond this layer, then it will be the output layer. The first layer added is always the input layer, the last layer added is always the output layer. Any layers added between those two layers are the hidden layers. A **FeedforwardLayer** object is created to serve as the hidden layer. The value of three specifies that there will be three neurons in the hidden layer.

```
network.addLayer(new FeedforwardLayer(3));
```

The final layer to be added will be the output layer. A **FeedforwardLayer** object is created. The value of one specifies that there will be a single output neuron.

```
network.addLayer(new FeedforwardLayer(1));
```

Finally, the neural network is reset. This randomizes the weight and threshold values. This random neural network will now need to be trained.

```
network.reset();
```

The backpropagation method will be used to train the neural network. To do this, a **Backpropagation** object is created.

```
final Train train = new Backpropagation(network, XOR_INPUT,
      XOR_IDEAL, 0.7, 0.9);
```

The training object requires several arguments to be passed to its constructor. The first argument is the network to be trained. The second argument is the **XOR_INPUT** and **XOR_IDEAL** variables, which provide the training sets and expected results. Finally, the learning rate and momentum are specified.

The learning rate specifies how fast the neural network will learn. This is usually a value around one, as it is a percent. The momentum specifies how much of an effect the previous training iteration will have on the current iteration. The momentum is also a percent, and is usually a value near one. To use no momentum in the backpropagation algorithm, you will specify a value of zero. The learning rate and momentum values will be discussed further later in this chapter.

Now that the training object is set up, the program will loop through training iterations until the error rate is small, or it performs 5,000 epochs, or iterations.

```
int epoch = 1;
do {
  train.iteration();
```

```
  System.out .println("Epoch #"
  + epoch
  + " Error:"
  + train.getError());
  epoch++;
} while ((epoch < 5000) && (train.getError() > 0.001));
```

To perform a training iteration, simply call the **iteration** method on the training object. The loop will continue until the error is smaller than one-tenth of a percent, or the program has performed 5,000 training iterations.

Finally, the program will display the results produced by the neural network.

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

for (int i = 0; i < XOR_IDEAL.length; i++) {
```

As the program loops through each of the training sets, that training set is presented to the neural network. To present a pattern to the neural network, the **computeOutputs** method is used. This method accepts a **double** array of input values. This array must be the same size as the number of input neurons or an exception will be thrown. This method returns an array of **double** values the same size as the number of output neurons.

```
  final double actual[] =
    network.computeOutputs(XOR_INPUT[i]);
```

The output from the neural network is displayed.

```
    System.out.println(XOR_INPUT[i][0] + ","
      + XOR_INPUT[i][1]
      + ", actual="
      + actual[0] + ",ideal=" + XOR_IDEAL[i][0]);
  }
}
```

This is a very simple neural network. It used the default sigmoid activation function. As you will see in the next section, other activation functions can be specified.

Activation Functions

Most neural networks pass the output of their layers through activation functions. These activation functions scale the output of the neural network into proper ranges. The neural network program in the last section used the sigmoid activation function. The sigmoid activation function is the default choice for the **FeedforwardLayer** class. It is possible to use others. For example, to use the hyperbolic tangent activation function, the following lines of code would be used to create the layers.

```
network.addLayer(new FeedforwardLayer(new ActivationTANH(),2));
```

```
network.addLayer(new FeedforwardLayer(new ActivationTANH(),3));
network.addLayer(new FeedforwardLayer(new ActivationTANH(),1));
```

As you can see from the above code, a new instance of **ActivationTANH** is created and passed to each layer of the network. This specifies that the hyperbolic tangent should be used, rather than the sigmoid function.

You may notice that it would be possible to use a different activation function for each layer of the neural network. While technically there is nothing stopping you from doing this, such practice would be unusual.

There are a total of three activation functions provided:

- Hyperbolic Tangent
- Sigmoid
- Linear

It is also possible to create your own activation function. There is an interface named **ActivationFunction**. Any class that implements the **ActivationFunction** interface can serve as an activation function. The three activation functions provided will be discussed in the following sections.

Using a Sigmoid Activation Function

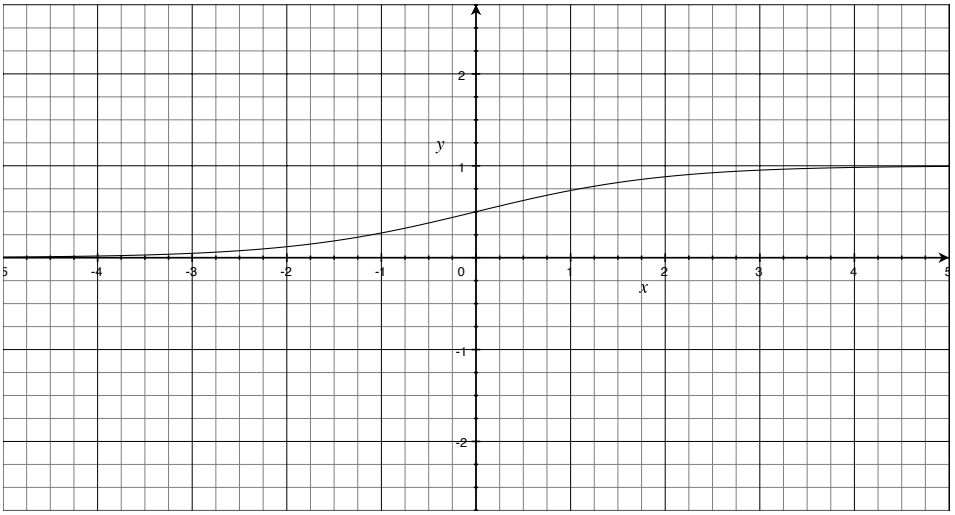
A sigmoid activation function uses the sigmoid function to determine its activation. The sigmoid function is defined as follows:

Equation 5.1: The Sigmoid Function

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

The term sigmoid means curved in two directions, like the letter "S." You can see the sigmoid function in Figure 5.2.

Figure 5.2: The Sigmoid function.



One important thing to note about the sigmoid activation function is that it only returns positive values. If you need the neural network to return negative numbers, the sigmoid function will be unsuitable. The sigmoid activation function is implemented in the **`ActivationFunctionSigmoid`** class. This class is shown in Listing 5.2.

Listing 5.2: The Sigmoid Activation Function Class (ActivationSigmoid.java)

```
package com.heatonresearch.book.introneuralnet.neural.activation;

/**
 * ActivationSigmoid: The sigmoid activation function takes on a
 * sigmoidal shape.  Only positive numbers are generated.  Do not
 * use this activation function if negative number output is
 * desired.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class ActivationSigmoid implements ActivationFunction {
      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
5622349801036468572L;

      /**
       * A threshold function for a neural network.
       * @param The input to the function.
```

```
     * @return The output from the function.
     */
    public double activationFunction(final double d) {
          return 1.0 / (1 + Math.exp(-1.0 * d));
    }

    /**
     * Some training methods require the derivative.
     * @param The input.
     * @return The output.
     */
    public double derivativeFunction(double d) {
          return d*(1.0-d);
    }

}
```

As you can see, the sigmoid function is defined inside the **activationFunction** method. This method was defined by the **ActivationFunction** interface. If you would like to create your own activation function, it is as simple as creating a class that implements the **ActivationFunction** interface and providing an **activationFunction** method.

The **ActivationFunction** interface also defines a method named **derivativeFunction** that implements the derivative of the main activation function. Certain training methods require the derivative of the activation function. **Backpropagation** is one such method. **Backpropagation** cannot be used on a neural network that uses an activation function that does not have a derivative. However, a genetic algorithm or simulated annealing could still be used. These two techniques will be covered in the next two chapters.

Using a Hyperbolic Tangent Activation Function

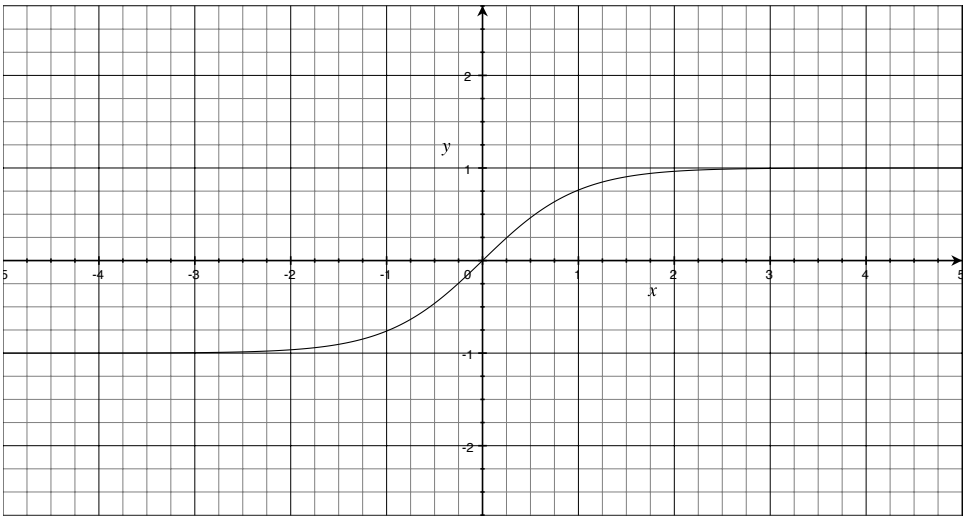
As previously mentioned, the sigmoid activation function does not return values less than zero. However, it is possible to "move" the sigmoid function to a region of the graph so that it does provide negative numbers. This is done using the hyperbolic tangent function. The equation for the hyperbolic activation function is shown in Equation 5.2.

Equation 5.2: The TANH Function

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

Although this looks considerably more complex than the sigmoid function, you can safely think of it as a positive and negative compatible version of the sigmoid function. The graph for the hyperbolic tangent function is provided in Figure 5.3.

Figure 5.3: The hyperbolic tangent function.



One important thing to note about the hyperbolic tangent activation function is that it returns both positive and negative values. If you need the neural network to return negative numbers, this is the activation function to use. The hyperbolic tangent activation function is implemented in the **`ActivationFunctionTanH`** class. This class is shown in Listing 5.3.

Listing 5.3: The Hyperbolic Tangent Function Class (ActivationTANH.java)

```
package com.heatonresearch.book.introneuralnet.neural.activation;

/**
 * ActivationTANH: The hyperbolic tangent activation function
 * takes the curved shape of the hyperbolic tangent.  This
 * activation function produces both positive and negative
 * output.  Use this activation function if both negative and
 * positive output is desired.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class ActivationTANH implements ActivationFunction {
```

```
        /**
         * Serial id for this class.
         */
        private static final long serialVersionUID =
9121998892720207643L;

        /**
         * A threshold function for a neural network.
         * @param The input to the function.
         * @return The output from the function.
         */
        public double activationFunction(double d) {
              final double result = (Math.exp(d*2.0)-1.0)/
                    (Math.exp(d*2.0)+1.0);
              return result;
        }

        /**
         * Some training methods require the derivative.
         * @param The input.
         * @return The output.
         */
        public double derivativeFunction(double d) {
              return( 1.0-Math.pow(activationFunction(d), 2.0) );
        }

}
```

As you can see, the hyperbolic tangent function is defined inside the **`activationFunction`** method. This method was defined by the **`ActivationFunction`** interface. The **`derivativeFunction`** is also defined to return the result of the derivative of the hyperbolic tangent function.

Using a Linear Activation Function

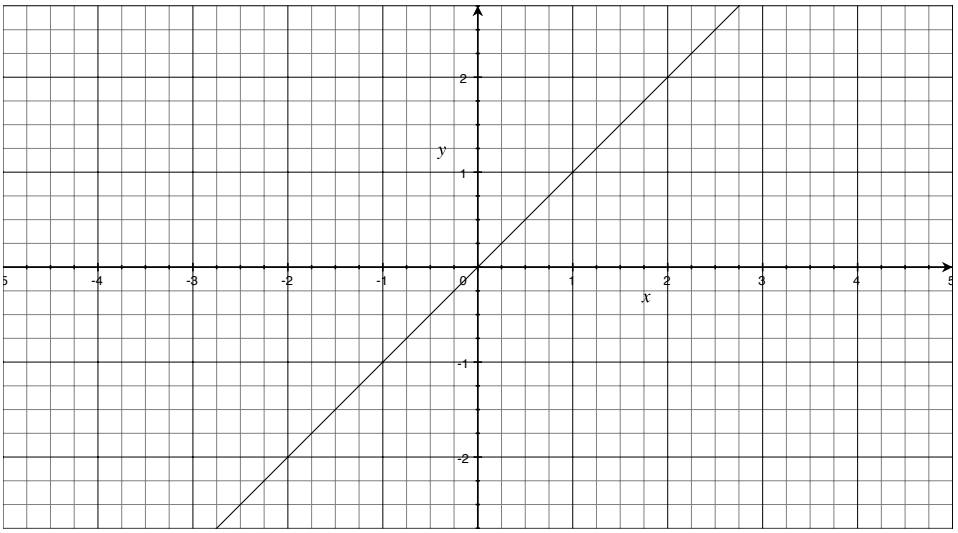
The linear activation function is essentially no activation function at all. It is probably the least commonly used of the activation functions. The linear activation function does not modify a pattern before outputting it. The function for the linear layer is given in Equation 5.3.

Equation 5.3: A Linear Function

$$f(x) = x$$

The linear activation function might be useful in situations when you need the entire range of numbers to be output. Usually, you will want to think of your neurons as active or non-active. Because the hyperbolic tangent and sigmoid activation functions both have established upper and lower bounds, they tend to be used more for Boolean (on or off) type operations. The linear activation function is useful for presenting a range. A graph of the linear activation function is provided in Figure 5.4.

Figure 5.4: The linear activation function.



The implementation for the linear activation function is fairly simple. It is shown in Listing 5.4.

Listing 5.4: The Linear Activation Function (ActivationLinear.java)

```
package com.heatonresearch.book.introneuralnet.neural.activation;

import com.heatonresearch.book.introneuralnet.neural.exception.
NeuralNetworkError;

/**
 * ActivationLinear: The Linear layer is really not an
 * activation function at all.  The input is simply passed on,
 * unmodified, to the output. This activation function is
 * primarily theoretical and of little actual use.  Usually an
 * activation function that scales between 0 and 1 or
 * -1 and 1 should be used.
 *
 * @author Jeff Heaton
```

```
 * @version 2.1
 */
public class ActivationLinear implements ActivationFunction {

      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
-5356580554235104944L;

      /**
       * A threshold function for a neural network.
       * @param The input to the function.
       * @return The output from the function.
       */
      public double activationFunction(final double d) {
            return d;
      }

      /**
       * Some training methods require the derivative.
       * @param The input.
       * @return The output.
       */
      public double derivativeFunction(double d) {
            throw new NeuralNetworkError("Can't use the linear"
 +" activation function where a derivative is required.");
      }

}
```

As you can see, the linear activation function does no more than return what it is given. The derivative of the linear function is 1. This is not useful for training, thus the **`derivativeFunction`** for the linear activation function throws an exception. You cannot use backpropagation to train a neural network that makes use of the linear function.

The Number of Hidden Layers

There are really two decisions that must be made regarding the hidden layers: how many hidden layers to actually have in the neural network and how many neurons will be in each of these layers. We will first examine how to determine the number of hidden layers to use with the neural network.

Problems that require two hidden layers are rarely encountered. However, neural networks with two hidden layers can represent functions with any kind of shape. There is currently no theoretical reason to use neural networks with any more than two hidden layers. In fact, for many practical problems, there is no reason to use any more than one hidden layer. Table 5.1 summarizes the capabilities of neural network architectures with various hidden layers.

Table 5.1: Determining the Number of Hidden Layers

Number of Hidden Layers	Result
none	Only capable of representing linear separable functions or decisions.
1	Can approximate any function that contains a continuous mapping from one finite space to another.
2	Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

Deciding the number of hidden neuron layers is only a small part of the problem. You must also determine how many neurons will be in each of these hidden layers. This process is covered in the next section.

The Number of Neurons in the Hidden Layers

Deciding the number of neurons in the hidden layers is a very important part of deciding your overall neural network architecture. Though these layers do not directly interact with the external environment, they have a tremendous influence on the final output. Both the number of hidden layers and the number of neurons in each of these hidden layers must be carefully considered.

Using too few neurons in the hidden layers will result in something called underfitting. Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set.

Using too many neurons in the hidden layers can result in several problems. First, too many neurons in the hidden layers may result in overfitting. Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers. A second problem can occur even when the training data is sufficient. An inordinately large number of neurons in the hidden layers can in-

crease the time it takes to train the network. The amount of training time can increase to the point that it is impossible to adequately train the neural network. Obviously, some compromise must be reached between too many and too few neurons in the hidden layers.

There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers, such as the following:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

These three rules provide a starting point for you to consider. Ultimately, the selection of an architecture for your neural network will come down to trial and error. But what exactly is meant by trial and error? You do not want to start throwing random numbers of layers and neurons at your network. To do so would be very time consuming. Chapter 8, "Pruning a Neural Network" will explore various ways to determine an optimal structure for a neural network.

Examining the Feedforward Process

Earlier in this chapter you saw how to present data to a neural network and train that neural network. The remainder of this chapter will focus on how these operations were performed. The neural network classes presented earlier are not overly complex. We will begin by exploring how they calculate the output of the neural network. This is called the feedforward process.

Calculating the Output Mathematically

Equation 5.4 describes how the output of a single neuron can be calculated.

Equation 5.4: Feedforward Calculations

$$output = (\sum_{i=0}^{n-1} x_i w_i) + w_n$$

The above equation takes input values named **x**, and multiplies them by the weight **w**. As you will recall from chapter 2, the last value in the weight matrix is the threshold. This threshold is **wn**.

To perform the above operation with matrix mathematics, the input is used to populate a matrix and a row is added, the elements of which are all ones. This value will be multiplied against the threshold value. For example, if the input were 0.1, 0.2, and 0.3, the following input matrix would be produced.

Equation 5.5: An Input Matrix

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 & 1 \end{bmatrix}$$

The dot product would then be taken between the input matrix and the weight matrix. This number would then be fed to the activation function to produce the output from the neuron.

Calculating the Output for a Feedforward Neural Network

To obtain the output from the neural network, the **computeOutputs** function should be called. This function will call the layers of the neural network and determine the output. The signature for the **computeOutputs** function is shown here:

```
public double[] computeOutputs(final double input[])
```

The first thing that the **computeOutputs** function does is to check and see if the **input** array is of the correct size. The size of the **input** array must match the number of input neurons in the input layer.

```
if (input.length != this.inputLayer.getNeuronCount()) {
  throw new NeuralNetworkError(
  "Size mismatch: Can't compute outputs for input size="
  + input.length
  + " for input layer size="
  + this.inputLayer.getNeuronCount());
}
```

Next, each of the **FeedforwardLayer** objects will be looped across.

```
for (final FeedforwardLayer layer : this.layers) {
```

The input layer will receive the input that was provided.

```
  if (layer.isInput()) {
    layer.computeOutputs(input);
```

Each of the hidden layers will receive the output from the previous layer as input.

```
  } else if (layer.isHidden()) {
    layer.computeOutputs(null);
  }
}
```

Finally, the output layer's result is returned.

```
return (this.outputLayer.getFire());
```

The **computeOutputs** method for the neural network obtains its output by calculating the output for each layer. The next section will explain how the output for a layer is calculated.

Calculating the Output for a Layer

The **FeedforwardLayer** class contains a **computeOutputs** method as well. This method is used to calculate the output for each of the layers. The signature for this method is shown here:

```
public double[] computeOutputs(final double pattern[])
```

First, the **computeOutputs** method checks to see whether or not a pattern was presented to it.

```
  if (pattern != null) {
```

If a **pattern** was presented, then the pattern should be copied to the **fire** instance variable.

```
  for (int i = 0; i < getNeuronCount(); i++) {
    setFire(i, pattern[i]);
  }
}
```

If a **pattern** was not presented, then the **fire** instance variable will have already been set by the previous layer.

```
    final Matrix inputMatrix =
       createInputMatrix(this.fire);
```

Next, an output value must be calculated to become the input value for each of the neurons in the next layer.

```
    for (i = 0; i < this.next.getNeuronCount(); i++) {
```

To do this, we obtain a column matrix for each of the neurons in the next layer.

```
      final Matrix col = this.matrix.getCol(i);
```

The dot product between the weight matrix column and input is then determined. This will be the value that will be presented to the next layer.

```
      final double sum = MatrixMath.dotProduct(col, inputMatrix);
```

The output is passed to the **activationFunction** and then stored in the fire instance variable of the next layer.

```
      this.next.setFire(i,
```

```
        this.activationFunction.activationFunction(sum));
    }
```

The **fire** instance variable is returned as the output for this layer.

```
return this.fire;
```

This process continues with each layer. The output from the output layer is the output from the neural network.

Examining the Backpropagation Process

You have now seen how to calculate the output for a feedforward neural network. You have seen both the mathematical equations and the Java implementation. As we examined how to calculate the final values for the network, we used the connection weights and threshold values to determine the final result. You may be wondering how these values were determined.

The values contained in the weight and threshold matrix were determined using the backpropagation algorithm. This is a very useful algorithm for training neural networks. The backpropagation algorithm works by running the neural network just as we did in our recognition example, as shown in the previous section. The main difference in the backpropagation algorithm is that we present the neural network with training data. As each item of training data is presented to the neural network, the error is calculated between the actual output of the neural network and the output that was expected (and specified in the training set). The weights and threshold are then modified, so there is a greater chance of the network returning the correct result when the network is next presented with the same input.

Backpropagation is a very common method for training multilayered feedforward networks. Backpropagation can be used with any feedforward network that uses a activation function that is differentiable. It is this derivative function that we will use during training. It is not necessary that you understand calculus or how to take the derivative of an equation to work with the material in this chapter. If you are using one of the common activation functions, you can simply get the activation function derivative from a chart.

To train the neural network, a method must be determined to calculate the error. As the neural network is trained, the network is presented with samples from the training set. The result obtained from the neural network is then compared with the anticipated result that is part of the training set. The degree to which the output from the neural network differs from this anticipated output is the error.

To train the neural network, we must try to minimize this error. To minimize the error, the neuron connection weights and thresholds must be modified. We must define a function that will calculate the rate of error of the neural network. This error function must be mathematically differentiable. Because the network uses a differentiable

activation function, the activations of the output neurons can be thought of as differentiable functions of the input, weights, and thresholds. If the error function is also a differentiable function, such as the sum of square error function, the error function itself is a differentiable function of these weights. This allows us to evaluate the derivative of the error using the weights. Then, using these derivatives, we find weights and thresholds that will minimize the error function.

There are several ways to find weights that will minimize the error function. The most popular approach is to use the gradient descent method. The algorithm that evaluates the derivative of the error function is known as backpropagation, because it propagates the errors backward through the network.

The Train Interface

This book will cover three different training methods that can be used for feedforward neural networks. This chapter presents the backpropagation method. Chapter 6 will discuss using a genetic algorithm for training. Chapter 7 will discuss using simulated annealing for a neural network. All three of these training methods implement the Train interface. The **`Train`** interface is shown in Listing 5.5.

Listing 5.5: The Train Interface (Train.java)

```
package com.heatonresearch.book.introneuralnet.neural.feedforward.
train;

import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;

/**
 * Train: Interface for all feedforward neural network training
 * methods.  There are currently three training methods define:
 *
 * Backpropagation
 * Genetic Algorithms
 * Simulated Annealing
 *
 * @author Jeff Heaton
 * @version 2.1
 */

public interface Train {

      /**
       * Get the current error percent from the training.
       * @return The current error.
       */
      public double getError();
```

```
        /**
         * Get the current best network from the training.
         * @return The best network.
         */
        public FeedforwardNetwork getNetwork();

        /**
         * Perform one iteration of training.
         */
        public void iteration();
}
```

There are three different methods that must be implemented to use the **Train** interface. The **getError** method will return the current error level, calculated using root mean square (RMS) for the current neural network. The **getNetwork** method returns a trained neural network that achieves the error level reported by the **getError** method. Finally, the **iteration** method is called to perform one training iteration. Generally, the **iteration** method is called repeatedly until the **getError** method returns an acceptable error.

The Backpropagation Java Classes

The backpropagation algorithm is implemented in two classes. The **Backpropagation** class, which implements the **Train** interface, is the main training class used. Internally, the **Backpropagation** class uses the **BackpropagationLayer** class to hold information that the backpropagation algorithm needs for each layer of the neural network.

The two main methods that will be called by methods using the **Backpropagation** class are the **iteration** and **getError** functions. The **iteration** function has the following signature:

```
public void iteration() {
```

The iteration function begins by looping through all of the training sets that were provided.

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

Each training set is presented to the neural network and the outputs are calculated.

```
  this.network.computeOutputs(this.input[j]);
```

Once the outputs have been calculated, the training can begin. This is a two-step process. First, the error is calculated by comparing the output to the ideal values.

```
  calcError(this.ideal[j]);
```

Once all the sets have been processed, then the network learns from these errors.

```
learn();
```

Finally, the new global error, that is, the error across all training sets, is calculated.

```
this.error = this.network.calculateError(this.input, this.ideal);
```

It is this error that will be returned when a call is made to **`getError`**.

Calculating the Error for Backpropagation

The first step in backpropagation error calculation is to call the **`calcError`** method of the **`BackPropagation`** object. The signature for the **`calcError`** method is shown here:

```
public void calcError(final double ideal[])
```

First, we verify that the size of the **`ideal`** array corresponds to the number of output neurons. Since the **`ideal`** array specifies ideal values for the output neurons, it must match the size of the output layer.

```
if (ideal.length !=
   this.network.getOutputLayer().getNeuronCount()) {

   throw new NeuralNetworkError(
     "Size mismatch: Can't calcError for ideal input size="
     + ideal.length + " for output layer size="
     + this.network.getOutputLayer().getNeuronCount());
}
```

The **`BackPropagation`** object contains a **`BackpropagationLayer`** object for each of the layers in the neural network. These objects should be cleared. The following code zeros out any previous errors from the **`BackpropagationLayer`** objects.

```
for (final FeedforwardLayer layer :this.network.getLayers()) {
  getBackpropagationLayer(layer).clearError();
}
```

As its name implies, the backpropagation propagates backwards through the neural network.

```
for (int i = this.network.getLayers().size() - 1; i >= 0; i--) {
```

Obtain each layer of the neural network.

```
  final FeedforwardLayer layer = this.network.getLayers().get(i);
```

Now call either of two overloaded versions of the **calcError** method. If it is the output layer, then pass in the **ideal** for comparison. If it is not the output layer, then the **ideal** values are not needed.

```
  if (layer.isOutput()) {
    getBackpropagationLayer(layer).calcError(ideal);
  } else {
  getBackpropagationLayer(layer).calcError();
 }
}
```

The **BackpropagationLayer** class has two versions of the **calcError** method. The signature for the version that operates on the output layer is shown below:

```
public void calcError(final double ideal[]) {
```

First, calculate the error share for each neuron. Loop across all of the output neurons.

```
for (int i = 0; i < this.layer.getNeuronCount(); i++) {
```

Next, set the error for this neuron. The error is simply the difference between the ideal output and the actual output.

```
  setError(i, ideal[i] - this.layer.getFire(i));
```

Calculate the delta for this neuron. The delta is the error multiplied by the derivative of the activation function. Bound the number, so that it does not become extremely small or large.

```
  setErrorDelta(i, BoundNumbers.bound(calculateDelta(i)));
}
```

These error deltas will be used during the learning process that is covered in the next section.

All other layers in the neural network will have their errors calculated by the **calcError** method that does not require an **ideal** array.

```
public void calcError() {
```

First, obtain the next layer. Since we are propagating backwards, this will be the layer that was just processed.

```
final BackpropagationLayer next =
  this.backpropagation.getBackpropagationLayer(
  this.layer.getNext());
```

Loop through every matrix value for connections between this layer and the next.

```
for (int i = 0; i < this.layer.getNext().getNeuronCount(); i++) {
  for (int j = 0; j < this.layer.getNeuronCount(); j++) {
```

The error calculation methods are called for each training set, therefore, it is necessary to accumulate the matrix deltas before the errors are cleared out. Determine this layer's contribution to the error by looking at the next layer's delta, and compare it to the outputs from this layer. Since we are propagating backwards, the next layer is actually the layer we just processed.

```
    accumulateMatrixDelta(j, i, next.getErrorDelta(i)
* this.layer.getFire(j));
```

Calculate, and add to, this layer's error by multiplying the matrix value by its delta.

```
setError(j, getError(j) + this.layer.getMatrix().get(j, i)

* next.getErrorDelta(i));
}
```

Also, accumulate deltas that affect the threshold.

```
  accumulateBiasDelta(i, next.getErrorDelta(i));
}
```

For the hidden layers, calculate the delta using the derivative of the activation function.

```
if (this.layer.isHidden()) {
  // hidden layer deltas
  for (int i = 0; i < this.layer.getNeuronCount(); i++) {
    setErrorDelta(i, BoundNumbers.bound(calculateDelta(i)));
  }
}
```

Once all of the errors have been calculated, the **`learn`** method can be used to apply the deltas to the weight matrix and teach the neural network to better recognize the input pattern.

Backpropagation Learning

The learning process is relatively simple. All of the desired changes were already calculated during the error calculation. It is now simply a matter of applying these changes. The values of the learning rate and momentum parameters will affect how these changes are applied.

The **learn** method in the **BackPropagation** object is called to begin the learning process.

```
public void learn()
```

Loop across all of the layers. The order is not important. During error calculation, the results from one layer depended upon another. As a result, it was very important to ensure that the error propagated backwards. However, during the learning process, values are simply applied to the neural network layers one at a time.

```
for (final FeedforwardLayer layer : this.network.getLayers()) {
```

Calling the **learn** method of each of the **BackpropagationLayer** objects causes the calculated changes to be applied.

```
  getBackpropagationLayer(layer).learn(this.learnRate,
      this.momentum);
}
```

The **learn** method provided in the **BackpropagationLayer** class is used to perform the actual modifications. The signature for the **learn** method is shown here:

```
public void learn(final double learnRate, final double momentum)
```

The **learn** layer makes sure there is a matrix. If there is no matrix, then there is nothing to train.

```
  if (this.layer.hasMatrix()) {
```

A matrix is then made that contains the accumulated matrix delta values that are scaled by the **learnRate**. The learning rate can be thought of as a percent. A value of one means that the deltas will be applied with no scaling.

```
    final Matrix m1 = MatrixMath.multiply(this.accMatrixDelta,
                        learnRate);
```

The previous deltas are stored in the **matrixDelta** variable. The learning from the previous iteration is applied to the current iteration scaled by the **momentum** variable. Some variants of **Backpropagation** use no momentum. To specify no momentum, use a **momentum** value of zero.

```
    final Matrix m2 = MatrixMath.multiply(this.matrixDelta,
      momentum);
```

Add the two together and store the result in the **matrixDelta** variable. This will be used with the momentum for the next training iteration.

```
    this.matrixDelta = MatrixMath.add(m1, m2);
```

Add the calculated values to the current matrix. This modifies the matrix and causes learning.

```
this.layer.setMatrix(MatrixMath.add(this.layer.getMatrix(),
    this.matrixDelta));
```

Clear the errors for the next learning iteration.

```
    this.accMatrixDelta.clear();
}
```

The **learn** method on the **BackpropagationLayer** class is called once per layer.

Chapter Summary

In this chapter, you learned how a feedforward backpropagation neural network functions. The feedforward backpropagation neural network is actually composed of two neural network algorithms. It is not necessary to always use feedforward and backpropagation together, but this is often the case. Other training methods will be introduced in coming chapters. The term "feedforward" refers to a method by which a neural network recognizes a pattern. The term "backpropagation" describes a process by which the neural network will be trained.

A feedforward neural network is a network in which neurons are only connected to the next layer. There are no connections between neurons in previous layers or between neurons and themselves. Additionally, neurons are not connected to neurons beyond the next layer. As a pattern is processed by a feedforward design, the thresholds and connection weights will be applied.

Neural networks can be trained using backpropagation. Backpropagation is a form of supervised training. The neural network is presented with the training data, and the results from the neural network are compared with the expected results. The difference between the actual results and the expected results is the error. Backpropagation is a method whereby the weights and input threshold of the neural network are altered in a way that causes this error to be reduced.

Backpropagation is not the only way to train a feedforward neural network. Simulated annealing and genetic algorithms are two other common methods. The next chapter will demonstrate how a genetic algorithm can be used to train a neural network.

Vocabulary

Activation Function

Backpropagation

Derivative

Feedforward

Hyperbolic Tangent Activation Function

Learning Rate

Linear Activation Function

Momentum

Overfitting

Sigmoid Activation Function

Underfitting

Questions for Review

1. What is an activation function? Explain when you might use the hyperbolic tangent activation function over the sigmoid activation function.

2. How can you determine if an activation function is compatible with the backpropagation training method?

3. Consider a neural network with one output neuron and three input neurons. The weights between the three input neurons and the one output neuron are 0.1, 0.2, and 0.3. The threshold is 0.5. What would be the output value for an input of 0.4, 0.5, and 0.6?

4. Explain the role of the learning rate in backpropagation.

5. Explain the role of the momentum in backpropagation.