Chapter 4: How a Machine Learns

- Understanding Layers
- Supervised Training
- Unsupervised Training
- Error Calculation
- Understanding Hebb's Rule and the Delta Rule

There are many different ways that a neural network can learn; however, every learning algorithm involves the modification of the weight matrix, which holds the weights for the connections between the neurons. In this chapter, we will examine some of the more popular methods used to adjust these weights. In chapter 5, "The Feedforward Backpropagation Neural Network," we will follow up this discussion with an introduction to the backpropagation method of training. Backpropagation is one of the most common neural network training methods used today.

Learning Methods

Training is a very important process for a neural network. There are two forms of training that can be employed, supervised and unsupervised. Supervised training involves providing the neural network with training sets and the anticipated output. In unsupervised training, the neural network is also provided with training sets, but not with anticipated outputs. In this book, we will examine both supervised and unsupervised training. This chapter will provide a brief introduction to each approach. They will then be covered in much greater detail in later chapters.

Unsupervised Training

What does it mean to train a neural network without supervision? As previously mentioned, the neural network is provided with training sets, which are collections of defined input values. The unsupervised neural network is not provided with anticipated outputs.

Unsupervised training is typically used to train classification neural networks. A classification neural network receives input patterns, which are presented to the input neurons. These input patterns are then processed, causing a single neuron on the output layer to fire. This firing neuron provides the classification for the pattern and identifies to which group the pattern belongs.

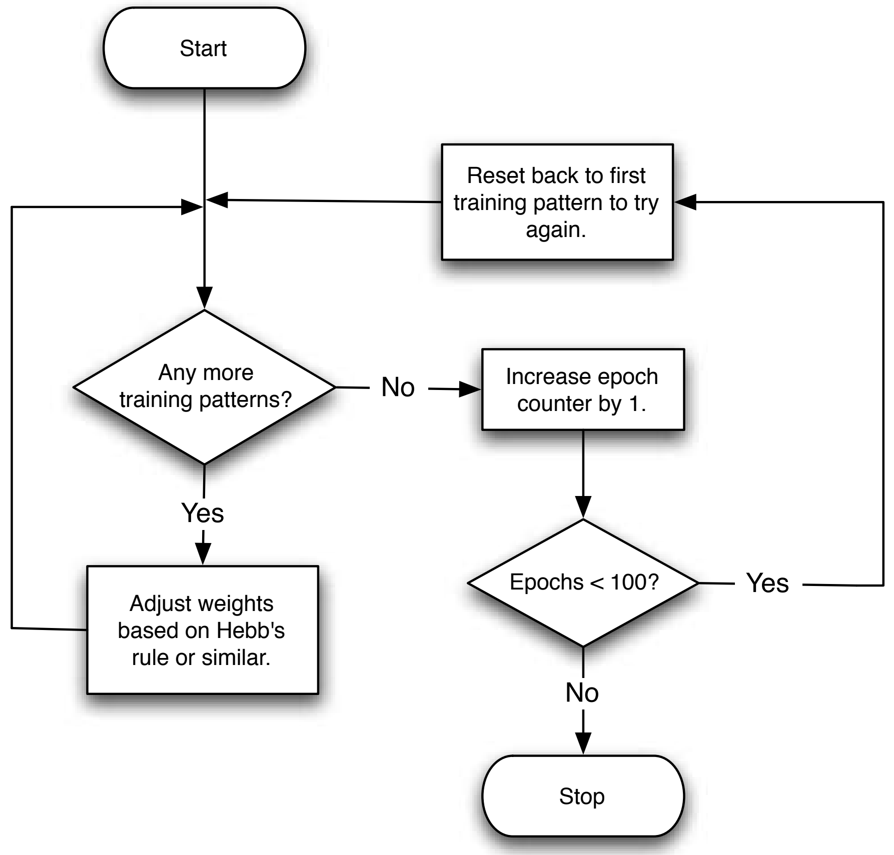
Another common application for unsupervised training is data mining. In this case, you have a large amount of data to be searched, but you may not know exactly what you are looking for. You want the neural network to classify this data into several groups. You do not want to dictate to the neural network ahead of time which input pattern should be classified into which group. As the neural network trains, the input patterns fall into groups with other inputs having similar characteristics. This allows you to see which input patterns share similarities.

Unsupervised training is also a very common training technique for self-organizing maps (SOM), also called Kohonen neural networks. In chapter 11, we will discuss how to construct an SOM and introduce the general process for training them without supervision.

In chapter 12, “OCR and the Self-Organizing Map,” you will be shown a practical application of an SOM. The example program presented in chapter 12, which is designed to read handwriting, learns through the use of an unsupervised training method. The input patterns presented to the SOM are dot images of handwritten characters and there are 26 output neurons, which correspond to the 26 letters of the English alphabet. As the SOM is trained, the weights are adjusted so input patterns can then be classified into these 26 groups. As will be demonstrated in chapter 12, this technique results in a relatively effective method for character recognition.

As you can see, unsupervised training can be applied to a number of situations. It will be covered in much greater detail in chapters 11 and 12. Figure 4.1 illustrates the flow of information through an unsupervised training algorithm.

Figure 4.1: Unsupervised training.


Start
Reset back to first training pattern to try again.
Any more training patterns?
No
Increase epoch counter by 1.
Yes
Adjust weights based on Hebb's rule or similar.
Epochs < 100?
Yes
No
Stop

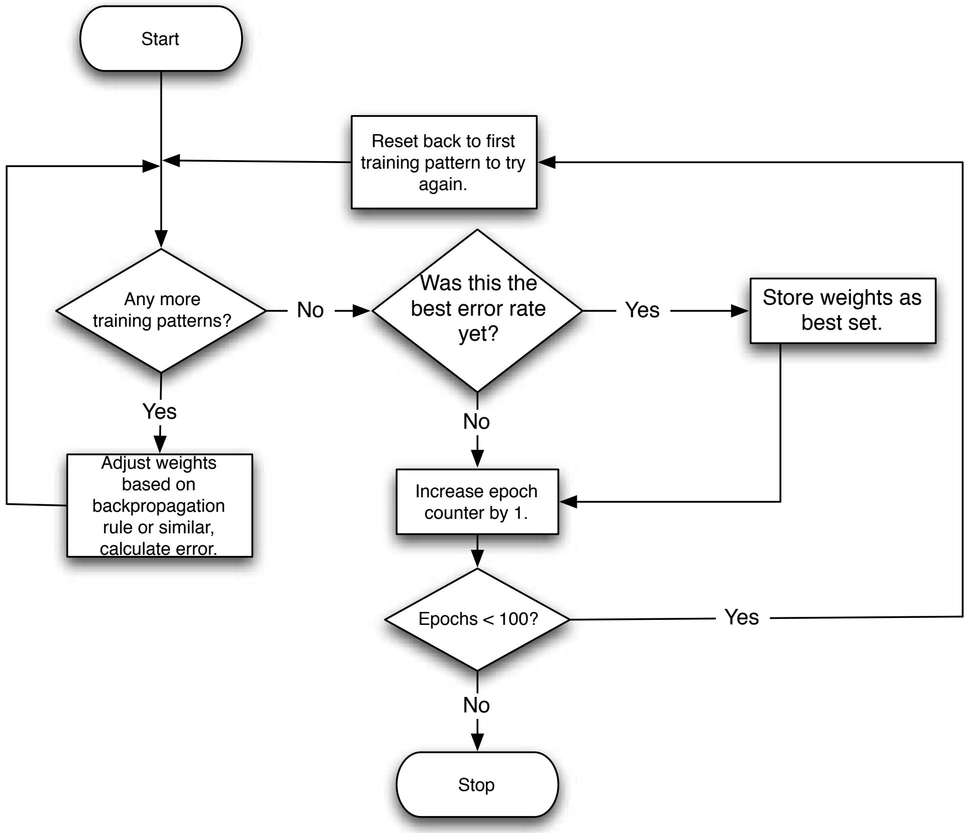
Supervised Training

The supervised training method is similar to the unsupervised training method, in that training sets are provided. Just as with unsupervised training, these training sets specify input signals to the neural network. The primary difference between supervised and unsupervised training is that in supervised training the expected outputs are provided. This allows the neural network to adjust the values in the weight matrix based on the differences between the anticipated output and the actual output.

There are several popular supervised training algorithms. One of the most common is the backpropagation algorithm. Backpropagation will be discussed in the next chapter. It is also possible to use simulated annealing or genetic algorithms to implement supervised training. Simulated annealing and genetic algorithms will be discussed in chapters 6, "Training using a Genetic Algorithm," and chapter 7, "Training using Simulated Annealing." We will now discuss how errors are calculated for both supervised and unsupervised training algorithms.

Figure 4.2 illustrates the flow of information through a supervised training algorithm.

Figure 4.2: Supervised training.


Start
Reset back to first training pattern to try again.
Any more training patterns?
No
Was this the best error rate yet?
Yes
Store weights as best set.
Yes
No
Adjust weights based on backpropagation rule or similar, calculate error.
Increase epoch counter by 1.
Epochs < 100?
Yes
No
Stop

Error Calculation

Error calculation is an important aspect of any neural network. Whether the neural network is supervised or unsupervised, an error rate must be calculated. The goal of virtually all training algorithms is to minimize the rate of error. In this section, we will examine how the rate of error is calculated for a supervised neural network. We will also discuss how the rate of error is determined for an unsupervised training algorithm. We will begin this section by examining two error calculation steps used for supervised training.

Error Calculation and Supervised Training

There are two values that must be considered in determining the rate of error for supervised training. First, we must calculate the error for each element of the training set as it is processed. Second, we must calculate the average of the errors for all of the elements of the training set across each sample. For example, consider the XOR logical operator from chapter 1 that has only four items in its training set. Refer to Table 1.3 to review the XOR logical operator.

In chapter 1, we intuitively chose values for the weight matrix. This is fine for a simple neural network, such as the one that implements the XOR operator. However, this is not practical for more complex neural networks. Typically, the process involves creating a random weight matrix and then testing each row in the training set. An output error is then calculated for each element of the training set. Finally, after all of the elements of the training set have been processed, the root mean square (RMS) error is determined for all of them.

Output Error

The output error is simply an error calculation that is performed to determine how different a neural network's output is from the ideal output. This value is rarely used for any purpose other than as a steppingstone in the calculation of the root mean square (RMS) error for the entire training set. Once all of the elements of a training set have been run through the network, the RMS error can be calculated. This error acts as the global rate of error for the entire neural network.

We will create a generic error calculation class that will be used in all of the neural networks in this book. This class is named **`ErrorCalculation`**. This class works by calculating the output error for each member of the training set. This error is allowed to grow until all of the elements of the training set have been presented. Then, the RMS error is calculated. The calculation of the RMS error is covered in the next section. The **`ErrorCalculation`** class is shown in Listing 4.1.

Listing 4.1: The ErrorCalculation Class (ErrorCalculation.java)

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


/**
 * ErrorCalculation: An implementation of root mean square (RMS)
 * error calculation.  This class is used by nearly every neural
 * network in this book to calculate error.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class ErrorCalculation {
      private double globalError;
      private int setSize;

      /**
       * Returns the root mean square error for a complete
       * training set.
       *
       * @param len
       *           The length of a complete training set.
       * @return The current error for the neural network.
       */
      public double calculateRMS() {
            final double err = Math.sqrt(this.globalError /
                  (this.setSize));
            return err;

      }

      /**
       * Reset the error accumulation to zero.
       */
      public void reset() {
            this.globalError = 0;
            this.setSize = 0;
      }

      /**
       * Called to update for each number that should be checked.
       * @param actual The actual number.
       * @param ideal The ideal number.
       */
      public void updateError(final double actual[],
            final double ideal[]) {
```

```
            for (int i = 0; i < actual.length; i++) {
                  final double delta = ideal[i] - actual[i];
                  this.globalError += delta * delta;
                  this.setSize += ideal.length;
            }
      }

}
```

First, we will see how to use the **ErrorCalculation** class. When an error is to be calculated, there will usually be two arrays, the **ideal** array and the **actual** array. The **ideal** array contains the values that we hope the neural network will produce. An **ideal** array is shown here:

```
double ideal[][] = {
  {1,2,3,4},
  {5,6,7,8},
  {9,10,11,12},
  {13,14,15,16} };
```

This **ideal** array contains four sets, which are the rows of the array. Each row contains four numbers. The neural network that this array would be used to train would, therefore, have four output neurons. The columns correspond to the output neurons. The rows are the individual elements of the training set.

The actual output of the neural network is stored in an **actual** array. If we were training a neural network, we would have an array of input values that would result in the actual output values. A hypothetical **actual** array is provided here for use with the above **ideal** array.

```
double actual[][] = {
  {1,2,3,5},
  {5,6,7,8},
  {9,10,11,12},
  {13,14,15,16} };
```

As you can see, the **actual** array is fairly close to the **ideal** array. An **ErrorCalculation** object is now instantiated named **error**.

```
ErrorCalculation error = new ErrorCalculation();
```

The output for each element in the training set must now be compared to the ideal output. We loop through all four rows in the arrays.

```
for(int i=0;i<ideal.length;i++) {
  error.updateError(actual[i], ideal[i]);
}
```

The corresponding rows of the **actual** array and the **ideal** array are presented to the **error** object and the **error object is updated.**

Finally, the RMS error is calculated and printed.

```
System.out.println( error.calculateRMS(ideal.length, ideal[0].
length));
```

The **error** object can be reused for another error calculation, if needed. Simply call the **reset** method and the error object is ready to be used again.

This **CalculateError** class will be used frequently in this book. Any time the RMS error is needed for a neural network, this class will be used. The next section will describe how the RMS error is calculated.

Root Mean Square (RMS) Error

The RMS method is used to calculate the rate of error for a training set based on predefined ideal results. The RMS method is effective in calculating the rate of error regardless of whether the actual results are above or below the ideal results. To calculate the RMS for a series of **n** values of **x**, consider Equation 4.1.

Equation 4.1: Root Mean Square Error (RMS)

$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$

The values of **x** are squared and their sum is divided by **n**. Squaring the values eliminates the issue associated with some values being above the ideal values and others below, since computing the square of a value always results in a positive number.

To apply RMS to the output of a neural network, consider Equation 4.2.

Equation 4.2: RMS for a Neural Network

$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (actual_i - ideal_i)^2}$$

To calculate the RMS for the arrays in the previous section, you would calculate the difference between the actual results and the ideal results, as shown in the above equation. The square of each of these would then be calculated and the results would be summed. The sum would then be divided by the number of elements, and the square root of the result of that computation would provide the rate of error.

To implement this in Java, the **updateError** method is called to compare the output produced for each element of the training set with the ideal output values for the neural network. The signature for the **updateError** method is shown here:

```
public void updateError(double actual[],double ideal[]) {
```

First, we loop through all of the elements in the **actual** array.

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

We determine the difference, or **delta**, between the actual and the ideal values. It does not matter if this is a negative number; that will be handled in the next step.

```
    double delta = ideal[i] - actual[i];
```

We then add the square of each **delta** to the **globalError** variable. The **setSize** variable is used to track how many elements have been processed.

```
    globalError += delta * delta;
    setSize += ideal.length;
  }
}
```

Finally, once all of the elements in the training set have been cycled through the **updateError** method, the **calculateRMS** method is called to calculate the RMS error. The signature for the **calculateRMS** method is shown here.

```
public double calculateRMS() {
```

We calculate the error as the square root of the **globalError** divided by the **setSize**, and return the error.

```
  double err = Math.sqrt(globalError / ((double)setSize));
  return err;
}
```

Once the **CalculateError** object has been used to calculate the rate of error, it must be reset before another training set can be processed. Otherwise, the **globalError** variable would continue to grow, rather than start from zero. To reset the **CalculateError** class, the **reset** method should be called.

Error Calculation and Unsupervised Training

We have discussed how errors are calculated for supervised training, we must now discuss how they are calculated for unsupervised training. This may not be immediately obvious. How can an error be calculated when no ideal outputs are provided? The exact procedure by which this is done will be covered in chapter 11, "Using a Self-Organizing Map." For now, we will simply highlight the most important details of the process.

Most unsupervised neural networks are designed to classify input data. The input data is classified based on one of the output neurons. The degree to which each output neuron fires for the input data is studied in order to produce an error for unsupervised training. Ideally, we would like a single neuron to fire at a high level for each member of the training set. If this is not the case, we adjust the weights to the neuron with the greatest number of firings, that is, the winning neuron consolidates its win. This training method causes more and more neurons to fire for the different elements in the training set.

Training Algorithms

Training occurs as the neuron connection weights are modified to produce more desirable results. There are several ways that training can take place. In the following sections we will discuss two simple methods for training the connection weights of a neural network. In chapter 5, we will examine backpropagation, which is a much more complex training algorithm.

Neuron connection weights are not modified in a single pass. The process by which neuron weights are modified occurs over multiple iterations. The neural network is presented with training data and the results are then observed. Neural network learning occurs when these results change the connection weights. The exact process by which this happens is determined by the learning algorithm used.

Learning algorithms, which are commonly called learning rules, are almost always expressed as functions. A learning function provides guidance on how a weight between two neurons should be changed. Consider a weight matrix containing the weights for the connections between four neurons, such as we saw in chapter 3, "Using a Hopfield Neural Network." This is expressed as an array of doubles.

```
double weights[][] = new double[4][4];
```

This matrix is used to store the weights between four neurons. Since Java array indexes begin with zero, we shall refer to these neurons as neurons zero through three. Using the above array, the weight between neuron two and neuron three would be contained in the location **`weights[2][3]`**. Therefore, we would like a learning function that will return the new weight between neurons "i" and "j," such that

```
weights[i][j] += learningRule(...)
```

The hypothetical method **`learningRule`** calculates the change (delta) that must occur between the two neurons in order for learning to take place. We never discard the previous weight value altogether; rather, we compute a delta value that is used to modify the original weight. It takes more than a single modification for the neural network to learn. Once the weights of the neural network have been modified, the network is again presented with the training data and the process continues. These iterations continue until the neural network's error rate has dropped to an acceptable level.

Another common input to the learning rule is the error. The error is the degree to which the actual output of the neural network differs from the anticipated output. If such an error is provided to the training function, then the method is called supervised training. In supervised training, the neural network is constantly adjusting the weights to attempt to better align the actual results with the anticipated outputs that were provided.

Conversely, if no error was provided to the training function, then we are using an unsupervised training algorithm. Recall, in unsupervised training, the neural network is not told what the "correct" output is. Unsupervised training leaves the neural network to determine this for itself. Often, unsupervised training is used to allow the neural network to group the input data. The programmer does not know ahead of time exactly what the groupings will be.

We will now examine two common training algorithms. The first, Hebb's rule, is used for unsupervised training and does not take into account network error. The second, the delta rule, is used with supervised training and adjusts the weights so that the input to the neural network will more accurately produce the anticipated output. We will begin with Hebb's Rule.

Hebb's Rule

One of the most common learning algorithms is called Hebb's Rule. This rule was developed by Donald Hebb to assist with unsupervised training. We previously examined a hypothetical learning rule defined by the following expression:

```
weights[i][j] += learningRule(...)
```

Rules for training neural networks are almost always represented as algebraic formulas. Hebb's rule is expressed in Equation 4.3.

Equation 4.3: Hebb's Rule

$$\Delta w_{ij} = \mu a_i j_i$$

The above equation calculates the needed change (delta) in the weight for the connection from neuron "i" to neuron "j." The Greek letter mu (μ) represents the learning rate. The activation of each neuron is given as **`ai`** and **`aj`**. This equation can easily be translated into the following Java method.

```
protected double learningRule(
  double rate, double input, double output)
{
  return rate*input*output;
}
```

We will now examine how this training algorithm actually works. To do this, we will consider a simple neural network with only two neurons. In this neural network, these two neurons make up both the input and output layer. There is no hidden layer. Table 4.1 summarizes some of the possible scenarios using Hebbian training. Assume that the learning rate is one.

Table 4.1: Using Hebb's Rule

Case	Neuron i Value	Neuron j Output	Hebb's Rule	Weight Delta
Case 1	+1	-1	1*1*-1	-1
Case 2	-1	+1	1*-1*1	-1
Case 3	+1	+1	1*1*1	+1

As you can see from the above table, if the activation of neuron "i" was +1 and the activation of neuron j was –1, the neuron connection weight between neuron "i" and neuron "j" would be decreased by one.

Hebb's rule is unsupervised, so we are not training the neural network for some ideal output. Rather, Hebb's rule works by reinforcing what the neural network already knows. This is sometimes summarized with the catchy phrase: "Neurons that fire together, wire together." That is, if the two neurons have similar activations, their weight is increased. If two neurons have dissimilar activations, their weight is decreased.

An example of Hebb's rule is shown in Listing 4.2.

Listing 4.2: Using Hebb's Rule (Hebb.java)

```
package com.heatonresearch.book.introneuralnet.ch4.hebb;

/**
 * Chapter 4: Machine Learning
 *
 * Hebb: Learn, using Hebb's rule.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class Hebb {

      /**
       * Main method just instanciates a delta object and calls
       * run.
       *
       * @param args
       *            Not used
       */
      public static void main(final String args[]) {
            final Hebb delta = new Hebb();
            delta.run();

      }

      /**
       * Weight for neuron 1
       */
      double w1;

      /**
       * Weight for neuron 2
       */
      double w2;

      /**
       * Learning rate
       */
      double rate = 1.0;

      /**
       * Current epoch #
       */
      int epoch = 1;

      public Hebb() {
            this.w1 = 1;
            this.w2 = -1;
      }

      /**
```

```
	 * Process one epoch. Here we learn from all three
	 * training samples and then
	 * update the weights based on error.
	 */

	protected void epoch() {
		System.out.println("***Beginning Epoch #"
			+ this.epoch + "***");
		presentPattern(-1, -1);
		presentPattern(-1, 1);
		presentPattern(1, -1);
		presentPattern(1, 1);
		this.epoch++;
	}

	/**
	 * Present a pattern and learn from it.
	 *
	 * @param i1
	 *            Input to neuron 1
	 * @param i2
	 *            Input to neuron 2
	 * @param i3
	 *            Input to neuron 3
	 * @param anticipated
	 *            The anticipated output
	 */
	protected void presentPattern(final double i1,
		final double i2) {
		double result;
		double delta;

		// run the net as is on training data
		// and get the error
		System.out.print("Presented [" + i1 + "," + i2 + "]");
		result = recognize(i1, i2);
		System.out.print(" result=" + result);

		// adjust weight 1
		delta = trainingFunction(this.rate, i1, result);
		this.w1 += delta;
		System.out.print(",delta w1=" + delta);

		// adjust weight 2
		delta = trainingFunction(this.rate, i2, result);
		this.w2 += delta;
```

```
			System.out.println(",delta w2=" + delta);

		}

		/**
		 * @param i1
		 *            Input to neuron 1
		 * @param i2
		 *            Input to neuron 2
		 * @param i3
		 *            Input to neuron 3
		 * @return the output from the neural network
		 */
		protected double recognize(final double i1, final double i2) {
			final double a = (this.w1 * i1) + (this.w2 * i2);
			return (a * .5);
		}

		/**
		 * This method loops through 10 epochs.
		 */
		public void run() {
			for (int i = 0; i < 5; i++) {
				epoch();
			}
		}

		/**
		 * The learningFunction implements Hebb's rule. This
		 * method will return
		 * the weight adjustment for the specified input neuron.
		 *
		 * @param rate
		 *            The learning rate
		 * @param input
		 *            The input neuron we're processing
		 * @param error
		 *            The error between the actual output and
		 * anticipated output.
		 * @return The amount to adjust the weight by.
		 */
		protected double trainingFunction(final double rate,
			final double input,
				final double output) {
			return rate * input * output;
		}
}
```

The Hebb's rule example uses two input neurons and one output neuron. As a result, there are a total of two weights, one weight for each of the connections between the input neurons and the output neuron. The first weight, which is the weight between neuron one and the output neuron is initialized to one. The second weight, which is the weight between neuron two and the output neuron, is initialized to two.

Consider the output from the first epoch.

```
***Beginning Epoch #1***
Presented [-1.0,-1.0] result=0.0,delta w1=-0.0,delta w2=-0.0
Presented [-1.0,1.0] result=-1.0,delta w1=1.0,delta w2=-1.0
Presented [1.0,-1.0] result=2.0,delta w1=2.0,delta w2=-2.0
Presented [1.0,1.0] result=0.0,delta w1=0.0,delta w2=0.0
```

The above output shows how the three-neuron network responded to four different input patterns. The middle two input patterns returned the strongest results. The second pattern was strong in the negative direction, the third pattern was strong in the positive direction. Hebb's rule tends to strengthen the output in the direction it already has a tendency towards.

The above output also shows the calculated deltas for weight one (w1) and weight two (w2). The first and fourth patterns both produced outputs of zero, so neither will produce a delta for the weight, other than zero. However, the negative output for pattern two will produce a weight delta of –1, and the positive result of pattern three will produce a weight delta of 2. These delta weights will be applied and will strengthen the negative or positive tendencies of the respective neurons.

It is also important to note that this example is always applying the deltas as the patterns are presented. This is why the third pattern will always have a stronger output than the second pattern; the delta for the second pattern has already been applied by the time the third pattern is presented.

The second epoch continues in much the same way as the first.

```
***Beginning Epoch #2***
Presented [-1.0,-1.0] result=0.0,delta w1=-0.0,delta w2=-0.0
Presented [-1.0,1.0] result=-4.0,delta w1=4.0,delta w2=-4.0
Presented [1.0,-1.0] result=8.0,delta w1=8.0,delta w2=-8.0
Presented [1.0,1.0] result=0.0,delta w1=0.0,delta w2=0.0
```

However, the deltas from the previous epoch have already been applied. New weight deltas are calculated that further enhance the positive or negative tendencies of the neurons.

Delta Rule

The delta rule is also known as the least mean squared error rule (LMS). Using this rule, the actual output of a neural network is compared against the anticipated output. Because the anticipated output is specified, using the delta rule is considered supervised training. Algebraically, the delta rule is written as follows in Equation 4.4.

Equation 4.4: The Delta Rule

$$\Delta w_{ij} = 2 \mu x_i (ideal - actual)_j$$

The above equation calculates the needed change (delta) in weights for the connection from neuron "i" to neuron "j." The Greek letter mu (µ) represents the learning rate. The variable **`ideal`** represents the desired output of the "j" neuron. The variable **`actual`** represents the actual output of the "j" neuron. As a result, **`(ideal-actual)`** is the error. This equation can easily be translated into the following Java method.

```
protected double trainingFunction(
    double rate, double input, double ideal, double actual)
{
  return rate*input*(ideal-actual);
}
```

We will now examine how the delta training algorithm actually works. To see this, we will look at the example program shown in Listing 4.3.

Listing 4.3: Using the Delta Rule (Delta.java)

```
package com.heatonresearch.book.introneuralnet.ch4.delta;

/**
 * Chapter 4: Machine Learning
 *
 * Delta: Learn, using the delta rule.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class Delta {

      /**
       * Main method just instanciates a delta object and
       * calls run.
       *
```

```
	 * @param args
	 *            Not used
	 */
	public static void main(final String args[]) {
		final Delta delta = new Delta();
		delta.run();

	}

	/**
	 * Weight for neuron 1
	 */
	double w1;

	/**
	 * Weight for neuron 2
	 */
	double w2;

	/**
	 * Weight for neuron 3
	 */
	double w3;

	/**
	 * Learning rate
	 */
	double rate = 0.5;

	/**
	 * Current epoch #
	 */
	int epoch = 1;

	/**
	 * Process one epoch. Here we learn from all three training
	 * samples and then
	 * update the weights based on error.
	 */

	protected void epoch() {
		System.out.println("***Beginning Epoch #"
				+ this.epoch + "***");
		presentPattern(0, 0, 1, 0);
		presentPattern(0, 1, 1, 0);
		presentPattern(1, 0, 1, 0);
		presentPattern(1, 1, 1, 1);
		this.epoch++;
	}

	/**
	 * This method will calculate the error between the
	 * anticipated output and
	 * the actual output.
	 *
	 * @param actual
	 *            The actual output from the neural network.
	 * @param anticipated
	 *            The anticipated neuron output.
	 * @return The error.
	 */
	protected double getError(final double actual,
		final double anticipated) {
		return (anticipated - actual);
	}

	/**
	 * Present a pattern and learn from it.
	 *
	 * @param i1
	 *            Input to neuron 1
	 * @param i2
	 *            Input to neuron 2
	 * @param i3
	 *            Input to neuron 3
	 * @param anticipated
	 *            The anticipated output
	 */
	protected void presentPattern(final double i1,
		final double i2,
			final double i3, final double anticipated) {
		double error;
		double actual;
		double delta;

		// run the net as is on training data
		// and get the error
		System.out.print("Presented [" + i1 + "," + i2 + ","
			+ i3 + "]");
		actual = recognize(i1, i2, i3);
		error = getError(actual, anticipated);
		System.out.print(" anticipated=" + anticipated);
		System.out.print(" actual=" + actual);
		System.out.println(" error=" + error);

		// adjust weight 1
		delta = trainingFunction(this.rate, i1, error);
		this.w1 += delta;

		// adjust weight 2
		delta = trainingFunction(this.rate, i2, error);
		this.w2 += delta;

		// adjust weight 3
		delta = trainingFunction(this.rate, i3, error);
		this.w3 += delta;
	}

	/**
	 * @param i1
	 *            Input to neuron 1
	 * @param i2
	 *            Input to neuron 2
	 * @param i3
	 *            Input to neuron 3
	 * @return the output from the neural network
	 */
	protected double recognize(final double i1, final double i2,
		final double i3) {
		final double a = (this.w1 * i1) + (this.w2 * i2)
			+ (this.w3 * i3);
		return (a * .5);
	}

	/**
	 * This method loops through 100 epochs.
	 */
	public void run() {
		for (int i = 0; i < 100; i++) {
			epoch();
		}
	}

	/**
	 * The learningFunction implements the delta rule. This
	 * method will return
	 * the weight adjustment for the specified input neuron.
	 *
	 * @param rate
	 *            The learning rate
	 * @param input
	 *            The input neuron we're processing
	 * @param error
	 *            The error between the actual output and
	 * anticipated output.
	 * @return The amount to adjust the weight by.
	 */
	protected double trainingFunction(final double rate,
			final double input,
				final double error) {
		return rate * input * error;
	}
}
```

This program will train for 100 iterations. It is designed to teach the neural network to recognize three patterns. These patterns are summarized as follows:

```
For 001 output 0
For 011 output 0
For 101 output 0
For 111 output 1
```

For each epoch, you will be shown the actual and anticipated results. By epoch 100, the network will be trained. The output from epoch 100 is shown here:

```
***Beginning Epoch #100***
Presented [0.0,0.0,1.0] anticipated=0.0 actu-
al=-0.33333333131711973 error=0.33333333131711973
Presented [0.0,1.0,1.0] anticipated=0.0 actual=0.333333333558949
error=-0.333333333558949
Presented [1.0,0.0,1.0] anticipated=0.0 actual=0.33333333370649876
error=-0.33333333370649876
Presented [1.0,1.0,1.0] anticipated=1.0 actual=0.6666666655103011
error=0.33333333448969893
```

As you can see from the above display, there are only two possible outputs 0.333 and 0.666. The output of 0.333 corresponds to 0 and the output of 0.666 corresponds to 1. A neural network will never produce the exact output desired, but through rounding it gets pretty close. While the delta rule is efficient at adjusting weights, it is not the most commonly used.

In the next chapter we will examine the feedforward backpropagation network, which is one of the most commonly used neural networks. Backpropagation is a more advanced form of the delta rule.

Chapter Summary

The rate of error for a neural network is a very important statistic, which is used as a part of the training process. This chapter showed you how to calculate the output error for an individual training set element, as well as how to calculate the RMS error for the entire training set.

Training occurs when the weights of the synapse are modified to produce a more suitable output. Unsupervised training occurs when the neural network is left to determine the correct responses. Supervised training occurs when the neural network is provided with training data and anticipated outputs. Hebb's rule can be used for unsupervised training. The delta rule is used for supervised training.

In this chapter we learned how a machine learns through the modification of the weights associated with the connections between neurons. This chapter introduced the basic concepts of how a machine learns. Backpropagation is a more advanced form of the delta rule, which was introduced in this chapter. In the next chapter we will explore backpropagation and see how the neural network class implements it.

Vocabulary

Classification

Delta Rule

Epoch

Hebb's Rule

Iteration

Kohonen Neural Network

Learning Rate

Root Mean Square (RMS)

Self-Organizing Map (SOM)

Supervised Training

Unsupervised Training

Questions for Review

1. Explain the difference between supervised and unsupervised training.

2. What is the primary difference between the delta rule and Hebb's rule?

3. Consider the following four results; calculate the RMS error.

```
Training Set #1, Expected = 5, Actual = 5
Training Set #2, Expected = 2, Actual = 3
Training Set #3, Expected = 6, Actual = 5
Training Set #4, Expected = 8, Actual = 4
Training Set #5, Expected = 1, Actual = 2
```

4. Use Hebb's rule to calculate the weight adjustment.

```
Two Neurons: N1 and N2
N1 to N2 Weight: 3
N1 Activation: 2
N2 Activation: 6
```

Calculate the weight delta.

5. Use the delta rule to calculate the weight adjustment.

```
Two neurons: N1 and N2
N1 to N2 Weight: 3
N1 Activation: 2
N2 Activation: 6, Expected: 5
```

Calculate the weight delta.