CHAPTER 3: USING A HOPFIELD NEURAL NETWORK

- Understanding the Hopfield Neural Network
- Recognizing Patterns
- Using Autoassociation
- Constructing a Neural Network Application

Neural networks have long been the mainstay of artificial intelligence (AI) programming. As programmers, we can create programs that do fairly amazing things. However, ordinary programs can only automate repetitive tasks, such as balancing checkbooks or calculating the value of an investment portfolio. While a program can easily maintain a large collection of images, it cannot tell us what is illustrated in any of those images. Programs are inherently unintelligent and uncreative.

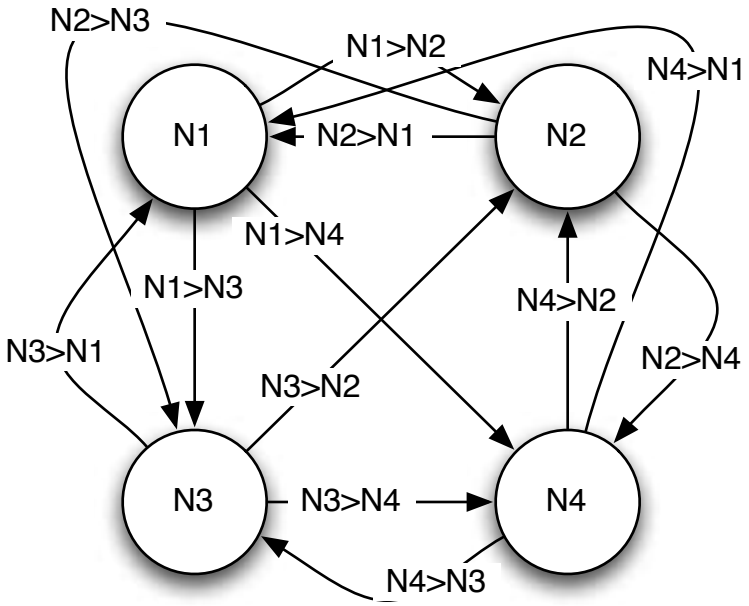
Neural networks attempt to give computer programs human-like intelligence. They are usually designed and trained to recognize specific patterns in data. This chapter will teach you the basic layout of a neural network by introducing the Hopfield neural network.

The Hopfield Neural Network

The Hopfield neural network is perhaps the simplest type of neural network. The Hopfield neural network is a fully connected single layer, autoassociative network. This means it has a single layer in which each neuron is connected to every other neuron. Autoassociative means that if the neural network recognizes a pattern, it will return that pattern.

In this chapter we will examine a Hopfield neural network with just four neurons. It is small enough to be easily understood, yet it can recognize a few patterns. A Hopfield network, with connections, is shown in Figure 3.1.

Figure 3.1: A Hopfield neural network with 12 connections.



We will build an example program that creates the Hopfield network shown in Figure 3.1. Since every neuron in a Hopfield neural network is connected to every other neuron, you might assume a four-neuron network contains 42 or 16 connections. However, 16 connections would require that every neuron be connected to itself, as well as to every other neuron. This is not the case in a Hopfield neural network, so the actual number of connections is 12.

As we develop an example neural network program, we will store the connections in a matrix. Since each neuron in a Hopfield neural network is by definition connected to every other neuron, a two dimensional matrix works well. All neural network examples in this book will use some form of matrix to store their weights.

Table 3.1 shows the layout of the matrix.

Table 3.1: Connections in a Hopfield Neural Network

	Neuron 1 (N1)	Neuron 2 (N2)	Neuron 3 (N3)	Neuron 4 (N4)
Neuron 1(N1)	(n/a)	N2->N1	N3->N1	N4->N1
Neuron 2(N2)	N1->N2	(n/a)	N3->N2	N3->N2
Neuron 3(N3)	N1->N3	N2->N3	(n/a)	N4->N3
Neuron 4(N4)	N1->N4	N2->N4	N3->N4	(n/a)

This matrix is called the weight matrix, and contains the weights associated with each connection. This matrix is the "memory" of the neural network, and will allow the neural network to recall certain patterns when they are presented. Many of the neural networks in this book will also contain threshold values, in addition to the weights. However, the Hopfield neural network contains only weights.

For example, the values shown in Table 3.2 show the correct weight values to use to enable a network to recall the patterns `0101` and `1010`. The method used to create a network with the values contained in Table 3.2 will be covered shortly. First, you will be shown how the values in Table 3.2 are used by the network to recall `0101` and `1010`.

Table 3.2: Weights Used to Recall 0101 and 1010

	Neuron 1 (N1)	Neuron 2 (N2)	Neuron 3 (N3)	Neuron 4 (N4)
Neuron 1 (N1)	0	-1	1	-1
Neuron 2 (N2)	-1	0	-1	1
Neuron 3 (N3)	1	-1	0	-1
Neuron 4 (N4)	-1	1	-1	0

Recalling Patterns

You will now be shown exactly how a neural network is used to recall patterns. We will begin by presenting `0101` to the Hopfield network. To do this, we present each input neuron, which in this case are also the output neurons, with the pattern. Each neuron will activate based upon the input pattern. For example, when Neuron 1 is presented with `0101`, its activation will result in the sum of all weights that have a 1 in the input pattern. For example, we can see from Table 3.2 that Neuron 1 has connections to the other neurons with the following weights:

0 -1 1 -1

We must now compare those weights with the input pattern of **0101**:

0 1 0 1

0 -1 1 -1

We will sum only the weights corresponding to the positions that contain a 1 in the input pattern. Therefore, the activation of the first neuron is –1 + –1, or –2. The results of the activation of each neuron are shown below.

```
N1 = -1 + -1 = -2
N2 = 0 + 1 = 1
N3 = -1 + -1 = -2
N4 = 1 + 0 = 1
```

Therefore, the output neurons, which are also the input neurons, will report the above activation results. The final output vector will then be –2, 1, –2, 1. These values are meaningless without an activation function. We said earlier that a threshold establishes when a neuron will fire. A threshold is a type of activation function. An activation function determines the range of values that will cause the neuron, in this case the output neuron, to fire. A threshold is a simple activation function that fires when the input is above a certain value.

The activation function used for a Hopfield network is any value greater than zero, so the following neurons will fire. This establishes the threshold for the network.

```
N1 activation result is -2; will not fire (0)
N2 activation result is 1; will fire (1)
N3 activation result is -2; will not fire(0)
N4 activation result is 1; will fire (1)
```

As you can see, we assign a binary value of 1 to all neurons that fired, and a binary value of 0 to all neurons that did not fire. The final binary output from the Hopfield network will be **0101**. This is the same as the input pattern. An autoassociative neural network, such as a Hopfield network, will echo a pattern back if the pattern is recognized. The pattern was successfully recognized. Now that you have seen how a connection weight matrix can cause a neural network to recall certain patterns, you will be shown how the connection weight matrix was derived.

Deriving the Weight Matrix

You are probably wondering how the weight matrix shown in Table 3.2 was derived. This section will explain how to create a weight matrix that can recall any number of patterns. First you should start with a blank connection weight matrix, as described in Equation 3.1.

Equation 3.1: A Blank Matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We will first train this neural network to accept the value **0101**. To do this, we must first calculate a matrix just for **0101**, which is called **0101**'s contribution matrix. The contribution matrix will then be added to the connection weight matrix. As additional contribution matrixes are added to the connection weight matrix, the connection weight is said to learn each of the new patterns.

We begin by calculating the contribution matrix of **0101**. There are three steps involved in this process. First, we must calculate the bipolar values of **0101**. Bipolar representation simply means that we are representing a binary string with –1's and 1's, rather than 0's and 1's. Next, we transpose and multiply the bipolar equivalent of **0101** by itself. Finally, we set all the values from the northwest diagonal to zero, because neurons do not have connections to themselves in a Hopfield network. Let's take the steps one at a time and see how this is done. We will start with the bipolar conversion.

Step 1: Convert 0101 to its bipolar equivalent.

We convert the input, because the binary representation has one minor flaw. Zero is NOT the inverse of 1. Rather –1 is the mathematical inverse of 1. Equation 3.2 can be used to convert the input string from binary to bipolar.

Equation 3.2: Binary to Bipolar

$$f(x) = 2x - 1$$

Conversely, Equation 3.3 can be used to convert from bipolar to binary.

Equation 3.3: Bipolar to Binary

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

To convert 0101 to its bipolar equivalent, we convert all of the zeros to –1's, as follows:

0 = –1

1 = 1

0 = –1

1 = 1

The final result is the array –1, 1, –1, 1. This array will be used in step 2 to begin building the contribution matrix for 0101.

Step 2: Multiply –1, 1, –1, 1 by its transposition.

For this step, we will consider –1, 1, –1, 1 to be a matrix:

Equation 3.4: Input Matrix

$$\begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

Taking the transposition of this matrix we have:

Equation 3.5: Input Matrix Transposed

$$\begin{bmatrix} -1 & 1 & -1 & 1 \end{bmatrix}$$

We must now multiply these two matrixes. Matrix multiplication was covered in chapter 2. It is a relatively easy procedure in which the rows and columns are multiplied against each other, resulting in the following:

-1 X -1 = 1 1 X -1 = -1 -1 X -1 = 1 1 X -1 = -1

-1 X 1 = -1 1 X 1 = 1 -1 X 1 = -1 1 X 1 = 1

-1 X -1 = 1 1 X -1 = -1 -1 X -1 = 1 1 X -1 = -1

-1 X 1 = -1 1 X 1 = 1 -1 X 1 = -1 1 X 1 = 1

Condensed, the above results in the following matrix:

Equation 3.6: Resulting Matrix

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Now that we have successfully multiplied the matrix by its inverse, we are ready for step 3.

Step 3: Set the northwest diagonal to zero.

Mathematically speaking, we are now going to subtract the identity matrix from the matrix we derived in step 2. The net result is that the cells in the northwest diagonal get set to zero. The real reason we do this is that neurons do not have connections to themselves in Hopfield networks. Thus, positions `[0][0]`, `[1][1]`, `[2][2]`, and `[3][3]` in our two dimensional array, or matrix, get set to zero. This results in the final contribution matrix for the bit pattern `0101`.

Equation 3.7: Contribution Matrix

$$\begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

This contribution matrix can now be added to the connection weight matrix. If we only want this network to recognize **0101**, then this contribution matrix becomes our connection weight matrix. If we also want to recognize **1001**, then we would calculate both contribution matrixes and add the results to create the connection weight matrix.

If this process seems a bit confusing, you might try looking at the next section in which we actually develop a program that builds connection weight matrixes. The process is explained using Java terminology.

Before we end the discussion of determining the weight matrix, one small side effect should be mentioned. We went through several steps to determine the correct weight matrix for **0101**. Any time you create a Hopfield network that recognizes a binary pattern, the network also recognizes the inverse of that bit pattern. You can get the inverse of a bit pattern by flipping all 0's to 1's and 1's to 0's. The inverse of **0101** is **1010**. As a result, the connection weight matrix we just calculated would also recognize **1010**.

Creating a Java Hopfield Neural Network

The **Hopfield** neural network is implemented using two classes. The first class, called **HopfieldNetwork** is the main class that performs training and pattern recognition. This class relies on the **Matrix** and **MatrixMath** classes, introduced in chapter 2, to work with the neural network's weight matrix. The second class, called **HopfieldException**, is an exception that is raised when an error occurs while processing the Hopfield network. This is usually triggered as a result of bad input.

The HopfieldNetwork Class

The **HopfieldNetwork** class is shown in Listing 3.1.

Listing 3.1: The Hopfield Neural Network (HopfieldNetwork.java)

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

import com.heatonresearch.book.introneuralnet.neural.
exception.NeuralNetworkError;
import com.heatonresearch.book.introneuralnet.neural.matrix.
BiPolarUtil;
import com.heatonresearch.book.introneuralnet.neural.matrix.
Matrix;
import com.heatonresearch.book.introneuralnet.neural.matrix.
MatrixMath;

/**
 * HopfieldNetwork: This class implements a Hopfield neural network.
 * A Hopfield neural network is fully connected and consists of a
```

```
 * single layer.  Hopfield neural networks are usually used for
 * pattern recognition.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class HopfieldNetwork {

      /**
       * The weight matrix for this neural network. A Hopfield
       * neural network is a
       * single layer, fully connected neural network.
       *
       * The inputs and outputs to/from a Hopfield neural
       * network are always  boolean values.
       */
      private Matrix weightMatrix;

      public HopfieldNetwork(final int size) {
            this.weightMatrix = new Matrix(size, size);

      }

      /**
       * Get the weight matrix for this neural network.
       *
       * @return
       */
      public Matrix getMatrix() {
            return this.weightMatrix;
      }

      /**
       * Get the size of this neural network.
       *
       * @return
       */
      public int getSize() {
            return this.weightMatrix.getRows();
      }

      /**
       * Present a pattern to the neural network and receive
       * the result.
       *
       * @param pattern
```

```
         *        The pattern to be presented to the neural network.
         * @return The output from the neural network.
         * @throws HopfieldException
         *             The pattern caused a matrix math error.
         */
        public boolean[] present(final boolean[] pattern) {

              final boolean output[] = new boolean[pattern.length];

              // convert the input pattern into a matrix with a
              // single row.
              // also convert the boolean values to
              // bipolar(-1=false, 1=true)
              final Matrix inputMatrix =
Matrix.createRowMatrix(BiPolarUtil
                          .bipolar2double(pattern));

              // Process each value in the pattern
              for (int col = 0; col < pattern.length; col++) {
                    Matrix columnMatrix =
this.weightMatrix.getCol(col);
                    columnMatrix =
MatrixMath.transpose(columnMatrix);

// The output for this input element is the dot product of the
// input matrix and one column from the weight matrix.
                    final double dotProduct = MatrixMath.
dotProduct(inputMatrix,
                                columnMatrix);

// Convert the dot product to either true or false.
                    if (dotProduct > 0) {
                          output[col] = true;
                    } else {
                          output[col] = false;
                    }
              }

              return output;
        }

        /**
         * Train the neural network for the specified pattern.
         * The neural network
         * can be trained for more than one pattern. To do this
         * simply call the
```

```
        * train method more than once.
        *
        * @param pattern
        *            The pattern to train on.
        * @throws HopfieldException
        *             The pattern size must match the size of
        * this neural network.
        */
       public void train(final boolean[] pattern) {
             if (pattern.length != this.weightMatrix.getRows()) {
                   throw new NeuralNetworkError(
"Can't train a pattern of size "
             + pattern.length + " on a hopfield network of size "
                               + this.weightMatrix.getRows());
             }

             // Create a row matrix from the input, convert
            // boolean to bipolar
             final Matrix m2 = Matrix.createRowMatrix(BiPolarUtil
                         .bipolar2double(pattern));
             // Transpose the matrix and multiply by the
             // original input matrix
             final Matrix m1 = MatrixMath.transpose(m2);
             final Matrix m3 = MatrixMath.multiply(m1, m2);

             // matrix 3 should be square by now, so create
             // an identity
             // matrix of the same size.
             final Matrix identity = MatrixMath.identity(
                   m3.getRows());

             // subtract the identity matrix
             final Matrix m4 = MatrixMath.subtract(m3, identity);

             // now add the calculated matrix, for this pattern,
             // to the
             // existing weight matrix.
             this.weightMatrix =
                   MatrixMath.add(this.weightMatrix, m4);

       }
}
```

To make use of the Hopfield neural network, you should instantiate an instance of this class. The constructor takes one integer parameter that specifies the size of the neural network. Once the **HopfieldNetwork** class has been instantiated, you can call the provided methods to use the neural network. These methods are summarized in Table 3.3.

Table 3.3: Summary of HopfieldNetwork Methods

Method Name	Purpose
getMatrix	Accesses the neural network's weight matrix.
getSize	Gets the size of the neural network.
present	Presents a pattern to the neural network.
train	Trains the neural network on a pattern.

The **getMatrix** and **getSize** methods are both very simple accessor methods and will not be covered further. Most of the work is done with the **present** and **train** methods. These methods will be covered in the next two sections.

Recalling Patterns with the Java Hopfield Network

To recall a pattern with the **HopfieldNetwork** class, the **present** method should be used. The signature for this method is shown here:

```
public boolean[] present(final boolean[] pattern)
      throws HopfieldException
```

The procedure for training a Hopfield neural network was already discussed earlier in this chapter. The earlier discussion explained how to recall a pattern mathematically. Now we will see how to implement Hopfield pattern recollection in Java.

First, an array is created to hold the output from the neural network. This array is the same length as the input array.

```
final boolean output[] = new boolean[pattern.length];
```

Next, a **Matrix** is created to hold the bipolar form of the input array. The **bipolar2double** method of the **BiPolarUtil** class is used to convert the **boolean** array. The one-dimensional array is converted into a "row matrix." A "row matrix" is a **Matrix** that consists of a single row.

```
final Matrix inputMatrix = Matrix.createRowMatrix(BiPolarUtil
                         .bipolar2double(pattern));
```

We must now loop through each item in the input array. Each item in the input array will be applied to the weight matrix to produce an output.

```
for (int col = 0; col < pattern.length; col++) {
```

Each column in the weight matrix represents the weights associated with the connections between the neuron and the other neurons. Therefore, we must extract the column that corresponds to each of the input array values.

```
  Matrix columnMatrix = this.weightMatrix.getCol(col);
```

We must now determine the dot product of that column and the input array. However, before that can be done we must transpose the column from the weight matrix. This will properly orient the values in the column so the dot product computation can be performed.

```
  columnMatrix = MatrixMath.transpose(columnMatrix);
```

Next, the dot product is calculated.

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

If the dot product is above zero, then the output will be **`true`**; otherwise the output will be **`false`**.

```
    if (dotProduct > 0) {
      output[col] = true;
    } else {
      output[col] = false;
    }
  }
```

Zero is the threshold. A value above the threshold will cause the output neuron to fire; at or below the threshold and the output neuron will not fire. It is important to note that since the Hopfield network has a single layer, the input and output neurons are the same. Thresholds will be expanded upon in the next chapter when we deal with the feedforward neural network.

Finally, the **`output`** array is returned.

```
return output;
```

The next section will explain how the Java Hopfield network is trained.

Training the Hopfield Network

The **`train`** method is used to train instances of the **`HopfieldNetwork`** class. The signature for this method is shown here:

```
public void train(final boolean[] pattern) throws HopfieldException
```

The length of the pattern must be the same as the size of the neural network. If it is not, then an exception is thrown.

```
if (pattern.length != this.weightMatrix.getRows()) {
  throw new HopfieldException("Can't train a pattern of size "
  + pattern.length + " on a Hopfield network of size "
  + this.weightMatrix.getRows());
}
```

If the pattern is the proper length, a matrix is created that contains a single row. This row will contain the bipolar representation of the input pattern.

```
final Matrix m2 = Matrix.createRowMatrix(BiPolarUtil
                                .bipolar2double(pattern));
```

The row matrix is then transposed into a column matrix.

```
final Matrix m1 = MatrixMath.transpose(m2);
```

Finally, the row is multiplied by the column.

```
final Matrix m3 = MatrixMath.multiply(m1, m2);
```

Multiplying the row by the column results in a square matrix. This matrix will have a diagonal of ones running from its northwest corner to its southwest corner. The ones must be converted to zeros. To do this, an identity matrix of the same size is created.

```
final Matrix identity = MatrixMath.identity(m3.getRows());
```

The identity matrix is nothing more than a matrix containing a diagonal of ones, which can be subtracted from the pervious matrix to set the diagonal to zero.

```
final Matrix m4 = MatrixMath.subtract(m3, identity);
```

The new matrix is now added to the old weight matrix.

```
this.weightMatrix = MatrixMath.add(this.weightMatrix, m4);
```

This produces a weight matrix that will likely recognize the new pattern, as well as the old patterns.

Simple Hopfield Example

Now you will see how to make use of the **`HopfieldNetwork`** class that was created in the last section. The first example implements a simple console application that demonstrates basic pattern recognition. The second example graphically displays the weight matrix using a Java applet. Finally, the third example uses a Java applet to illustrate how a Hopfield neural network can be used to recognize a grid pattern.

The first example, which is a simple console application, is shown in Listing 3.2.

Listing 3.2: Simple Console Example (ConsoleHopfield.java)

```
package com.heatonresearch.book.introneuralnet.ch3.console;

import com.heatonresearch.book.introneuralnet.neural.hopfield.Hop-
fieldNetwork;

/**
 * Chapter 3: Using a Hopfield Neural Network
 *
 * ConsoleHopfield: Simple console application that shows how to
 * use a Hopfield Neural Network.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class ConsoleHopfield {

      /**
       * Convert a boolean array to the form [T,T,F,F]
       *
       * @param b
       *            A boolen array.
       * @return The boolen array in string form.
       */
      public static String formatBoolean(final boolean b[]) {
            final StringBuilder result = new StringBuilder();
            result.append('[');
            for (int i = 0; i < b.length; i++) {
                  if (b[i]) {
                        result.append("T");
                  } else {
                        result.append("F");
                  }
                  if (i != b.length - 1) {
                        result.append(",");
                  }
            }
            result.append(']');
            return (result.toString());
      }
```

```
      /**
       * A simple main method to test the Hopfield neural network.
       *
       * @param args
       *            Not used.
       */
      public static void main(final String args[]) {

            // Create the neural network.
            final HopfieldNetwork network = new HopfieldNetwork(4);
            // This pattern will be trained
            final boolean[] pattern1 = { true, true, false, false
};
            // This pattern will be presented
            final boolean[] pattern2 = { true, false, false,
                  false };
            boolean[] result;

            // train the neural network with pattern1
            System.out.println("Training Hopfield network with: "
                        + formatBoolean(pattern1));
            network.train(pattern1);
            // present pattern1 and see it recognized
            result = network.present(pattern1);
            System.out.println("Presenting pattern:"
                  + formatBoolean(pattern1)
                        + ", and got " + formatBoolean(result));
            // Present pattern2, which is similar to pattern 1.
            // Pattern 1 should be recalled.
            result = network.present(pattern2);
            System.out.println("Presenting pattern:" +
formatBoolean(pattern2)
                        + ", and got " + formatBoolean(result));

      }

}
```

There are two methods provided in Listing 3.2. The first method, named **formatBoolean**, is used to format Boolean arrays as follows:

```
[T,T,F,F]
```

This method allows the program to easily display both input and output for the neural network. The **formatBoolean** function is relatively simple. It loops through each element in the array and displays either a **T** or an **F** depending upon whether the array element is **true** or **false**. This can be seen in Listing 3.2.

The second method, **main**, is used to set up the Hopfield network and use it. First, a new **HopfieldNetwork** is created with four neurons.

```
final HopfieldNetwork network = new HopfieldNetwork(4);
```

Next, an input pattern named **pattern1** is created. This is the pattern that the Hopfield network will be trained on. Since there are four neurons in the network, there must also be four values in the training pattern.

```
final boolean[] pattern1 = { true, true, false, false };
```

A second input pattern, named **pattern2**, is then created. This pattern is slightly different than **pattern1**. This pattern will allow the network to be tested to see if it still recognizes **pattern1**, even though this pattern is slightly different.

```
final boolean[] pattern2 = { true, false, false, false };
```

A Boolean array named **result** is created to hold the results of presenting patterns to the network.

```
boolean[] result;
```

The user is then informed that we are training the network with **pattern1**. The **formatBoolean** method is used to display **pattern1**.

```
System.out.println("Training Hopfield network with: "
+ formatBoolean(pattern1));
```

The network is called and trained with **pattern1**.

```
network.train(pattern1);
```

Now, **pattern1** is presented to the network to see if it will be recognized. We tell the user that we are doing this and display the **result**.

```
result = network.present(pattern1);
System.out.println("Presenting pattern:" + formatBoolean(pattern1)
+ " and got " + formatBoolean(result));
```

Next, **pattern2**, which is similar to **pattern1**, is presented. The same values should be recalled for **pattern2** as were recalled for **pattern1**.

```
result = network.present(pattern2);
System.out.println("Presenting pattern:" + formatBoolean(pattern2)
+ " and got " + formatBoolean(result));
```

The results are displayed to the user. The output from this program is as follows:

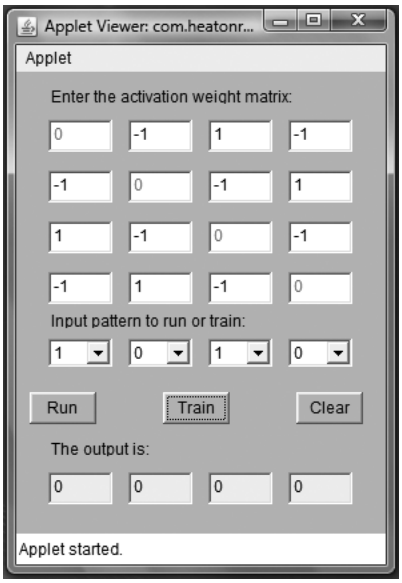
```
Training Hopfield network with: [T,T,F,F]
Presenting pattern:[T,T,F,F] and got [T,T,F,F]
Presenting pattern:[T,F,F,F] and got [T,T,F,F]
```

This program shows how to instantiate and use a Hopfield neural network without any bells or whistles. The next program is an applet that will allow you to see the weight matrix as the network is trained.

Visualizing the Weight Matrix

This second example is essentially the same as the first; however, this example uses an applet. Therefore, it has a GUI that allows the user to interact with it. The user interface for this program can be seen in Figure 3.2.

Figure 3.2: A Hopfield Applet



The 4x4 grid is the weight matrix. The four "0/1" fields allow you to input four-part patterns into the neural network. This pattern can then be presented to the network to be recognized or to be used for training.

The Hopfield applet is shown in Listing 3.3.

Listing 3.3: Visual Hopfield Weight Matrix Applet (HopfieldApplet.java)

```
package com.heatonresearch.book.introneuralnet.ch3.hopfield;

import java.applet.Applet;
import java.awt.Color;
import java.awt.event.FocusEvent;

import com.heatonresearch.book.introneuralnet.neural.hopfield.
```

```
      HopfieldNetwork;

/**
 * Chapter 3: Using a Hopfield Neural Network
 *
 * HopfieldApplet: Applet that allows you to work with a Hopfield
 * network.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class HopfieldApplet extends Applet implements
            java.awt.event.ActionListener,
            java.awt.event.FocusListener {
      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
6244453822526901486L;
      HopfieldNetwork network = new HopfieldNetwork(4);

      java.awt.Label label1 = new java.awt.Label();
      java.awt.TextField matrix[][] = new java.awt.
TextField[4][4];

      java.awt.Choice input[] = new java.awt.Choice[4];
      java.awt.Label label2 = new java.awt.Label();
      java.awt.TextField output[] = new java.awt.TextField[4];
      java.awt.Label label3 = new java.awt.Label();
      java.awt.Button go = new java.awt.Button();
      java.awt.Button train = new java.awt.Button();
      java.awt.Button clear = new java.awt.Button();

      /**
       * Called when the user clicks one of the buttons.
       *
       * @param event
       *            The event.
       */
      public void actionPerformed(
final java.awt.event.ActionEvent event) {
            final Object object = event.getSource();
            if (object == this.go) {
                  runNetwork();
            } else if (object == this.clear) {
                  clear();
            } else if (object == this.train) {
```

```
                train();
        }
    }

    /**
     * Clear the neural network.
     */
    void clear() {
        this.network.getMatrix().clear();
        this.setMatrixValues();
    }

    /**
     * Collect the matrix values from the applet and place
     * inside the weight
     * matrix for the neural network.
     */
    private void collectMatrixValues() {
        for (int row = 0; row < 4; row++) {
            for (int col = 0; col < 4; col++) {
                final String str =
                        this.matrix[row][col].getText();
                int value = 0;

                try {
                        value = Integer.parseInt(str);
                } catch (final NumberFormatException e) {
                        // let the value default to zero,
                        // which it already is by this point.
                }

                // do not allow neurons to self-connect
                if (row == col) {
                        this.network.getMatrix().set(
                        row, col, 0);
                } else {
                        this.network.getMatrix().set(
                        row, col, value);
                }

            }
        }
    }

    public void focusGained(final FocusEvent e) {
        // don't care
```

```
}

public void focusLost(final FocusEvent e) {
      this.collectMatrixValues();
      this.setMatrixValues();
}

/**
 * Setup the applet.
 */
@Override
public void init() {
      setLayout(null);
      setBackground(Color.lightGray);
      this.label1.setText(
            "Enter the activation weight matrix:");
      add(this.label1);
      this.label1.setBounds(24, 12, 192, 12);
      this.label2.setText(
            "Input pattern to run or train:");
      add(this.label2);
      this.label2.setBounds(24, 180, 192, 12);

      for (int row = 0; row < 4; row++) {
            for (int col = 0; col < 4; col++) {
                  this.matrix[row][col] =
                        new java.awt.TextField();
                  add(this.matrix[row][col]);
                  this.matrix[row][col].setBounds(24
                        + (col * 60),
                              36 + (row * 38), 48, 24);
                  this.matrix[row][col].setText("0");
                  this.matrix[row][col]
                        .addFocusListener(this);

                  if (row == col) {
                        this.matrix[row][col]
                        .setEnabled(false);
                  }

            }
      }

      for (int i = 0; i < 4; i++) {
            this.output[i] = new java.awt.TextField();
            this.output[i].setEditable(false);
```

```
                this.output[i].setText("0");
                this.output[i].setEnabled(true);
                add(this.output[i]);
                this.output[i].setBounds(24 + (i * 60),
                      300, 48, 24);

                this.input[i] = new java.awt.Choice();
                this.input[i].add("0");
                this.input[i].add("1");
                this.input[i].select(0);
                add(this.input[i]);
                this.input[i].setBounds(24 + (i * 60),
                      200, 48, 24);
          }

          this.label3.setText("The output is:");
          add(this.label3);
          this.label3.setBounds(24, 276, 192, 12);

          this.go.setLabel("Run");
          add(this.go);
          this.go.setBackground(java.awt.Color.lightGray);
          this.go.setBounds(10, 240, 50, 24);

          this.train.setLabel("Train");
          add(this.train);
          this.train.setBackground(java.awt.Color.lightGray);
          this.train.setBounds(110, 240, 50, 24);

          this.clear.setLabel("Clear");
          add(this.clear);
          this.clear.setBackground(java.awt.Color.lightGray);
          this.clear.setBounds(210, 240, 50, 24);

          this.go.addActionListener(this);
          this.clear.addActionListener(this);
          this.train.addActionListener(this);
     }

     /**
      * Collect the input, present it to the neural network,
      * then display the results.
      */
     void runNetwork() {

          final boolean pattern[] = new boolean[4];
```

```
            // Read the input into a boolean array.
            for (int row = 0; row < 4; row++) {
                  final int i = this.input[row].getSelectedIndex();
                  if (i == 0) {
                        pattern[row] = false;
                  } else {
                        pattern[row] = true;
                  }
            }

            // Present the input to the neural network.
            final boolean result[] = this.network.present(pattern);

            // Display the result.
            for (int row = 0; row < 4; row++) {
                  if (result[row]) {
                        this.output[row].setText("1");
                  } else {
                        this.output[row].setText("0");
                  }

// If the result is different than the input, show in yellow.
                  if (result[row] == pattern[row]) {
                        this.output[row].setBackground(
                              java.awt.Color.white);
                  } else {
                        this.output[row].setBackground(
                              java.awt.Color.yellow);
                  }
            }

      }

      /**
       * Set the matrix values on the applet from the matrix
       * values stored in the neural network.
       */
      private void setMatrixValues() {

            for (int row = 0; row < 4; row++) {
                  for (int col = 0; col < 4; col++) {
                        this.matrix[row][col].setText(""
                                    + (int) this.network.
                        getMatrix().get(row, col));
                  }
```

```
            }
      }

      /**
       * Called when the train button is clicked. Train for
       * the current pattern.
       */
      void train() {
            final boolean[] booleanInput = new boolean[4];

            // Collect the input pattern.
            for (int x = 0; x < 4; x++) {
                  booleanInput[x] = (this.input[x].
                  getSelectedIndex() != 0);
            }

            // Train the input pattern.
            this.network.train(booleanInput);
            this.setMatrixValues();

      }

}
```

This applet can be run online from most browsers by accessing the following URL:

http://www.heatonresearch.com/articles/61/page1.html

To use the applet, follow these steps:

1. Notice the activation weight matrix is empty (contains all zeros). This neural network has no knowledge. Let's teach it to recognize the pattern 1001. Enter 1001 under the "Input pattern to run or train:" prompt. Click the "Train" button. Notice the weight matrix adjusts to absorb the new knowledge.

2. Now test it. Enter the pattern 1001 under the "Input pattern to run or train:" prompt. (It should still be there from your training.) Now click "Run." The output will be "1001." This is an autoassociative network, therefore it echos the input if it recognizes it.

3. Let's test it some more. Enter the pattern 1000 and click "Run." The output will now be "1001." The neural network did not recognize "1000;" the closest thing it knew was "1001." It figured you made an error typing and attempted a correction!

4. Now, notice a side effect. Enter "0110," which is the binary inverse of what the network was trained with ("1001"). Hopfield networks ALWAYS get trained for a pattern's binary inverse. So, if you enter "0110," the network will recognize it.

5. Likewise, if you enter "0100," the neural network will output "0110" thinking that is what you meant.

6. One final test—let's try "1111," which is totally off base and not at all close to anything the neural network knows. The neural network responds with "0000." It did not try to correct you. It has no idea what you mean!

7. Play with it some more. It can be taught multiple patterns. As you train new patterns, it builds upon the matrix already in memory. Pressing "Clear," clears out the memory.

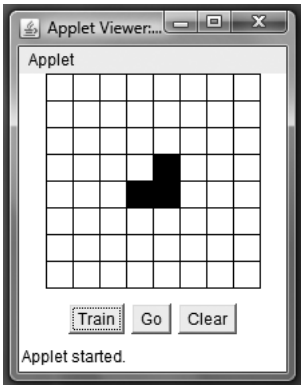
The Hopfield network works exactly the same way it did in the first example. Patterns are presented to a four-neuron Hopfield neural network for training and recognition. All of the extra code shown in Listing 3.3 simply connects the Hopfield network to a GUI using the Java Swing and Abstract Windowing Toolkit (AWT).

Hopfield Pattern Recognition Applet

Hopfield networks can be much larger than four neurons. For the third example, we will examine a 64-neuron Hopfield network. This network is connected to an 8x8 grid, which an applet allows you to draw upon. As you draw patterns, you can either train the network with them or present them for recognition.

The user interface for the applet can be seen in Figure 3.3.

Figure 3.3: A pattern recognition Hopfield applet.



The source code for this example is provided in Listing 3.4.

Listing 3.4: Hopfield Pattern Recognition (PatternApplet.java)

```
package com.heatonresearch.book.introneuralnet.ch3.pattern;

import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Graphics;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import com.heatonresearch.book.introneuralnet.neural.hopfield.Hop-
fieldNetwork;

/**
 * Chapter 3: Using a Hopfield Neural Network
 *
 * PatternApplet: An applet that displays an 8X8 grid that
 * presents patterns to a Hopfield
 * neural network.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class PatternApplet extends Applet implements
            MouseListener,
            ActionListener {
```

```
	/**
	 * Serial id for this class.
	 */
	private static final long serialVersionUID =
1882626251007826502L;
	public final int GRID_X = 8;
	public final int GRID_Y = 8;
	public final int CELL_WIDTH = 20;
	public final int CELL_HEIGHT = 20;
	public HopfieldNetwork hopfield;
	private boolean grid[];
	private int margin;
	private Panel buttonPanel;
	private Button buttonTrain;
	private Button buttonGo;
	private Button buttonClear;
	private Button buttonClearMatrix;

	public void actionPerformed(final ActionEvent e) {
		if (e.getSource() == this.buttonClear) {
			clear();
		} else if (e.getSource() == this.buttonClearMatrix) {
			clearMatrix();
		} else if (e.getSource() == this.buttonGo) {
			go();
		} else if (e.getSource() == this.buttonTrain) {
			train();
		}
	}

	/**
	 * Clear the grid.
	 */
	public void clear() {
		int index = 0;
		for (int y = 0; y < this.GRID_Y; y++) {
			for (int x = 0; x < this.GRID_X; x++) {
				this.grid[index++] = false;
			}
		}

		repaint();
	}
```

```
/**
 * Clear the weight matrix.
 */
private void clearMatrix() {
      this.hopfield.getMatrix().clear();
}

/**
 * Run the neural network.
 */
public void go() {

      this.grid = this.hopfield.present(this.grid);
      repaint();

}

@Override
public void init() {
      this.grid = new boolean[this.GRID_X * this.GRID_Y];
      this.addMouseListener(this);
      this.buttonTrain = new Button("Train");
      this.buttonGo = new Button("Go");
      this.buttonClear = new Button("Clear");
      this.buttonClearMatrix = new Button("Clear Matrix");
      this.setLayout(new BorderLayout());
      this.buttonPanel = new Panel();
      this.buttonPanel.add(this.buttonTrain);
      this.buttonPanel.add(this.buttonGo);
      this.buttonPanel.add(this.buttonClear);
      this.buttonPanel.add(this.buttonClearMatrix);
      this.add(this.buttonPanel, BorderLayout.SOUTH);

      this.buttonTrain.addActionListener(this);
      this.buttonGo.addActionListener(this);
      this.buttonClear.addActionListener(this);
      this.buttonClearMatrix.addActionListener(this);

      this.hopfield = new HopfieldNetwork(this.GRID_X *
      this.GRID_Y);
}

public void mouseClicked(final MouseEvent event) {
      // not used

}
```

```
public void mouseEntered(final MouseEvent e) {
      // not used

}

public void mouseExited(final MouseEvent e) {
      // not used

}

public void mousePressed(final MouseEvent e) {
      // not used

}

public void mouseReleased(final MouseEvent e) {
      final int x = ((e.getX() - this.margin) /
      this.CELL_WIDTH);
      final int y = e.getY() / this.CELL_HEIGHT;
      if (((x >= 0) && (x < this.GRID_X)) && ((y >= 0)
            && (y < this.GRID_Y))) {
            final int index = (y * this.GRID_X) + x;
            this.grid[index] = !this.grid[index];
      }
      repaint();

}

@Override
public void paint(final Graphics g) {
      this.margin = (this.getWidth() -
            (this.CELL_WIDTH * this.GRID_X)) / 2;
      int index = 0;
      for (int y = 0; y < this.GRID_Y; y++) {
            for (int x = 0; x < this.GRID_X; x++) {
                  if (this.grid[index++]) {
                        g.fillRect(this.margin + (x *
                        this.CELL_WIDTH), y
                        * this.CELL_HEIGHT, this.CELL_WIDTH,
                                    this.CELL_HEIGHT);
                  } else {
                        g.drawRect(this.margin + (x
                        * this.CELL_WIDTH), y
                        * this.CELL_HEIGHT, this.CELL_WIDTH,
                                    this.CELL_HEIGHT);
```

```
                        }
                    }
                }
        }

        /**
         * Train the neural network.
         */
        public void train() {

                this.hopfield.train(this.grid);

        }

}
```

This applet can be run online from most browsers by accessing the following URL:

http://www.heatonresearch.com/articles/61/page1.html

To make use of the applet, draw some sort of pattern on the grid and click "Train." Now draw another pattern and also click "Train." Finally, draw a pattern similar to one of the two previous patterns in the same location, and click "Go." The network will attempt to recognize what you drew. You can also click "Clear" to clear the grid. Clicking "Clear Matrix" will clear the training matrix.

The following sections explain how the main methods of this program were constructed.

Drawing the Grid

The grid is drawn using the applet's **`paint`** method. The signature for this method is shown here:

```
public void paint(final Graphics g)
```

The grid size is defined by several constants. Unless you change these, the grid itself will be 8x8 and the cells will be 20 pixels square. This means that the buttons will take up more space than the grid; therefore, it looks better if the grid is centered. A **`margin`** is calculated to center the grid.

The margin is the actual width minus the grid's width, divided by two.

```
this.margin = (this.getWidth() -
      (this.CELL_WIDTH * this.GRID_X)) / 2;
```

Next, the **index** variable is created. This variable holds the current position in the **grid boolean** array. Even though the grid appears visually as an 8x8 matrix, it must be presented to the neural network as a flat 64-part Boolean pattern.

```
int index = 0;
```

Two loops are then established to loop through the **x** and **y** coordinates of the grid.

```
for (int y = 0; y < this.GRID_Y; y++) {
  for (int x = 0; x < this.GRID_X; x++) {
```

If the grid element for this cell is **true**, then draw a rectangle and fill it in.

```
    if (this.grid[index++]) {
      g.fillRect(this.margin + (x * this.CELL_WIDTH), y
      * this.CELL_HEIGHT, this.CELL_WIDTH,
      this.CELL_HEIGHT);
```

If the grid element for this cell is **false**, then draw an empty rectangle.

```
    } else {
      g.drawRect(this.margin + (x * this.CELL_WIDTH), y
      * this.CELL_HEIGHT, this.CELL_WIDTH,
      this.CELL_HEIGHT);
    }
  }
}
```

The **paint** method is called by Java whenever the grid needs to be redrawn. Additionally, other methods will force the **paint** method to be called using the **repaint** method.

Toggling Grid Positions

The **mouseReleased** method is called by Java whenever the mouse has been released. Of course, the mouse must have been pressed first. The signature for the **mouseReleased** method is shown here:

```
public void mouseReleased(final MouseEvent e)
```

First, the **x** and **y** coordinates are calculated in terms of the grid position, and not the pixel coordinates that are passed to the **mouseReleased** event.

```
final int x = ((e.getX() - this.margin) / this.CELL_WIDTH);
final int y = e.getY() / this.CELL_HEIGHT;
```

These coordinates must fall on the grid. If the click was outside of the grid, then the click will be ignored.

```
if (((x >= 0) && (x < this.GRID_X)) && ((y >= 0) && (y < this.
GRID_Y))) {
```

The **index** into the one-dimensional grid array is calculated, then it is toggled using the Boolean **!** operator.

```
  final int index = (y * this.GRID_X) + x;
  this.grid[index] = !this.grid[index];
}
```

Finally, the applet repaints the grid.

```
repaint();
```

Thus, the grid can be drawn by tracking when the mouse is released.

Training and Presenting Patterns

The "Train" and "Go" buttons allow you to train and recognize patterns respectively. The **train** method is called whenever the "Train" button is clicked. The **train** method's signature is shown here:

```
public void train()
```

The **train** method of the neural network is called with the **grid** variable.

```
            try {
                  this.hopfield.train(this.grid);
```

It is highly unlikely that a **HopfieldException** will be thrown, since the applet is designed to send grids of the correct size. However, since the **HopfieldException** is a checked exception, it must be handled.

```
            } catch (final HopfieldException e) {
                  e.printStackTrace();
            }
      }
```

Minimal error handling is performed for the exception. The stack is dumped to the console.

The **go** method is called whenever the "Go" button is clicked. The signature for the **go** method is shown here:

```
public void go()
```

The grid is presented to the neural network for processing. The results from the **present** method are copied to the **grid** variable. Calling **repaint** displays the results from the network recognition.

```
            try {
                  this.grid = this.hopfield.present(this.grid);
                  repaint();
```

Again, it is highly unlikely that a **`HopfieldException`** will be thrown, since the applet is designed to send grids of the correct size, but it must be handled.

```
            } catch (final HopfieldException e) {
                  e.printStackTrace();
            }
      }
```

And again, minimal error handling is performed for the exception. The stack is dumped to the console.

Chapter Summary

Neural networks are one of the most commonly used concepts in Artificial Intelligence. Neural networks are particularly useful for recognizing patterns. They are able to recognize something even when it is distorted.

A neural network may have input, output, and hidden layers. The input and output layers are the only required layers. The input and output layers may be the same neurons. Neural networks are typically presented input patterns that will produce some output pattern.

If a neural network mimics the input pattern it was presented with, then that network is said to be autoassociative. For example, if a neural network is presented with the pattern "0110," and the output is also "0110," then that network is said to be autoassociative.

A neural network calculates its output based on the input pattern and the neural network's internal connection weight matrix. The values for the connection weights will determine the output from the neural network.

A Hopfield neural network is a fully connected autoassociative neural network. This means that each neuron is connected to every other neuron in the neural network. A Hopfield neural network can be trained to recognize certain patterns. Training a Hopfield neural network involves performing some basic matrix manipulations on the input pattern that is to be recognized.

This chapter explained how to construct a simple Hopfield neural network. This network was trained using a simple training algorithm. Training algorithms allow weights to be adjusted to produce the desired outputs. There are many advanced training algorithms. Chapter 4 will introduce more complex training algorithms.

Vocabulary

Activation Function

Autoassociation

Bipolar

Hopfield neural network

Single layer neural network

Questions for Review

1. A typical Hopfield neural network contains six neurons. How many connections will this produce?

2. Convert 1 binary to bipolar.

3. Convert -1 bipolar to binary.

4. Consider a four-neuron Hopfield neural network with the following weight matrix.

$$\begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix}$$

What output will an input of 1101 produce?

5. Consider a four-neuron Hopfield network. Produce a weight matrix that will recognize the pattern 1100.