CHAPTER 12: OCR WITH THE SELF-ORGANIZING MAP

- What is OCR?
- Cropping an Image
- Downsampling an Image
- Training the Neural Network to Recognize Characters
- Recalling Characters
- A "Commercial-Grade" OCR Application

In the previous chapter, you learned how to construct a self-organizing map (SOM). You learned that an SOM can be used to classify samples into several groups. In this chapter, we will examine a specific SOM application; we will apply an SOM to Optical Character Recognition (OCR).

OCR programs are capable of reading printed text. This may be text scanned from a document or handwritten text drawn on a hand-held device, such as a personal digital assistant (PDA). OCR programs are used widely in many industries. One of the largest users of OCR systems is the United States Postal Service.

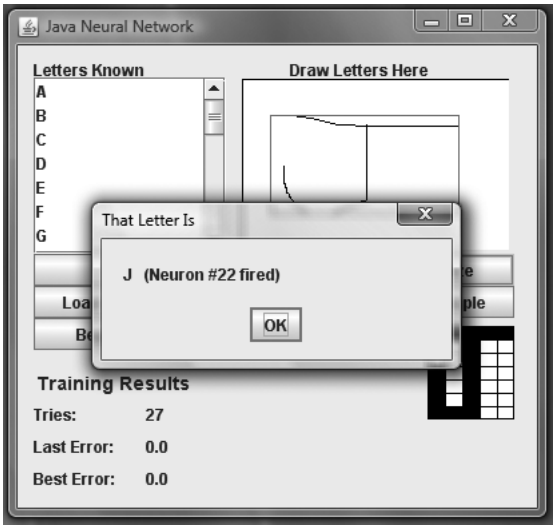
In the 1970s and 1980s, the US Postal Service had many letter-sorting machines (LSMs). These machines were manned by human clerks, who keyed the zip codes of 60 letters per minute. Human letter-sorters have now been replaced by computerized letter-sorting machines. These new letter-sorting machines rely on OCR technology; they scan incoming letters, read their zip codes, and route them to their correct destinations.

In this chapter, a program will be presented to demonstrate how a self-organizing map can be trained to recognize human handwriting. We will not create a program that can scan pages of text; rather, this program will read individual characters as they are drawn by the user. This function is similar to the handwriting recognition techniques employed by many PDA's.

The OCR Application

When the OCR application is launched, it displays a simple GUI interface. Through this interface, the user can both train and use the neural network. The GUI interface is shown in Figure 12.1.

Figure 12.1: The OCR application.



The program is not immediately ready to recognize letters upon startup. It must first be trained using letters that have actually been drawn. Training files are stored in the same directory as the OCR application. The name of the training sample is "sample.dat".

If you downloaded the "sample.dat" file from Heaton Research, you will see it contains handwriting samples I produced. If you use this file to train the program and then attempt to recognize your own handwriting, you may not experience the results you would achieve with a training file based on your own handwriting. Creating a sample based on your own handwriting will be covered in the next section. For now, we will focus on how the program recognizes handwriting using the sample file provided.

You should begin by clicking the "Load" button on the OCR application. The program will then attempt to load the training file. A small message box should be displayed indicating that the file was loaded successfully. Once the file has been loaded, the program will display all of the letters for which it will be trained. The training file provided only contains entries for the 26 capital letters of the Latin alphabet.

Now that the letters have been loaded, the neural network must be trained. By clicking the "Train" button, the application will begin the training process. The training process may take anywhere from a few seconds to several minutes, depending on the speed of your computer. A small message box will be displayed once training is complete.

Using the Sample Program to Recognize Letters

Now that the training set has been loaded and the neural network has been trained, you are ready to recognize characters. The user interface makes this process very easy. You simply draw the character that you would like to have the program recognize in the large rectangular region containing the instruction "Draw Letters Here". Once you have drawn a letter, you can select several different options.

The letters that you draw are downsampled before they are recognized, meaning the image is mapped to a small grid that is five pixels wide and seven pixels high. The advantage of downsampling to such a size is twofold. First, the lower-resolution image requires fewer input neurons for processing than a full-sized image. Second, by downsampling everything to the same size, the size of a character is neutralized; it does not matter if you draw a large character or a small character. If you click the "Downsample" button, you can see the downsampled version of your letter. Clicking the "Clear" button will cause the drawing and downsampled regions to be cleared.

You will notice that a box is drawn around your letter when you display the downsampled version. This is a cropping box. The purpose of the cropping box is to remove any non-essential white space in the image. This also has the desirable effect of eliminating the need to have the program consider a letter's position. A letter can be drawn in the center of the drawing region, near the top, or in some other location, and the program will still recognize it.

When you are ready to recognize a letter, you should click the "Recognize" button. This will cause the application to downsample your letter and then attempt to recognize it using the self-organizing map. The exact process for downsampling an image will be discussed in the next section. The pattern is then presented to the self-organizing map and the winning neuron is selected.

If you recall from chapter 11, a self-organizing map has several output neurons. One output neuron is selected as the winner for each input pattern. The self-organizing map used by this sample program has 26 output neurons to match the 26 letters in the sample set. The program will respond to the letter you enter by telling you both which neuron fired, and which letter it believes you have drawn. In matching my own handwriting, I have found this program generally achieves a success rate of approximately 80–90%. If you are having trouble getting the program to recognize your letters, ensure that you are writing clear capital letters. You may also try training the neural network to recognize your own handwriting, as covered in the next section.

Training the Sample Program to Recognize Letters

You may find that the program does not recognize your handwriting as well as you think it should. This may be because the program was trained using my handwriting, which may not be representative of the handwriting of the entire population. (My grade school teachers would surely argue that this is indeed the case.) In this section, I will explain how you can train the network using your own handwriting.

There are two approaches from which you can choose: you can start from a blank training set and enter all 26 letters yourself, or you can start with my training set and replace individual letters. The latter is a good approach if the network is recognizing most of your characters, and failing on only a small set. In this case, you can just retrain the neural network for the letters that the program is failing to recognize.

To delete a letter that the training set already has listed, you should select that letter and press the "Delete" button on the OCR application. Note that this is the GUI's "Delete" button and not the delete button on your computer's keyboard.

To add a new letter to the training set, you should draw your letter in the drawing input area. Once your letter is drawn, click the "Add" button. You will be prompted for the actual letter that you just drew. The character you type in response to this prompt will be displayed to you when the OCR application recognizes the letter.

Once your training set is complete you should save it. This is accomplished by clicking the application's "Save" button. This will save the training set to the file "sample.dat". If you already have a file named "sample.dat", it will be overwritten; therefore, it is important to make a copy of your previous training file if you would like to keep it. If you exit the OCR application without saving your training data, it will be lost. When you launch the OCR application again, you can click "Load" to retrieve the data you stored in the "sample.dat" file.

In the previous two sections you learned how to use the OCR application. As you have seen, the program is adept at recognizing characters that you have entered and demonstrates a good use of the self-organizing map.

Implementing the OCR Program

We will now see how the OCR program was implemented. There are several classes that make up the OCR application. The purpose of each class in the application is summarized in Table 12.1.

Table 12.1: Classes for the OCR Application

Class	Purpose
Entry	The drawing area through which the user inputs letters.
OCR	The main framework; this class starts the OCR application.
Sample	Used to display a downsampled image.
SampleData	Used to hold a downsampled image.

We will now examine each section of the program. We will begin by examining how a user draws an image.

Drawing Images

Though not directly related to neural networks, the process by which a user is able to draw characters is an important part of the OCR application. We will examine the process in this section. The code for this process is contained in the Sample.java file, and can be seen in Listing 12.1.

Listing 12.1: Drawing Images (Sample.java)

```
package com.heatonresearch.book.introneuralnet.ch12.ocr;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;
/**
 * Chapter 12: OCR and the Self Organizing Map
 *
 * Sample: GUI element that displays sampled data.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class Sample extends JPanel {

      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
2250441617163548592L;
      /**
       * The image data.
       */
      SampleData data;
```

```
/**
 * The constructor.
 *
 * @param width
 *             The width of the downsampled image
 * @param height
 *             The height of the downsampled image
 */
Sample(final int width, final int height) {
      this.data = new SampleData(' ', width, height);
}

/**
 * The image data object.
 *
 * @return The image data object.
 */
SampleData getData() {
      return this.data;
}

/**
 * @param g
 *             Display the downsampled image.
 */
@Override
public void paint(final Graphics g) {
      if (this.data == null) {
            return;
      }

      int x, y;
      final int vcell = getHeight() / this.data.getHeight();
      final int hcell = getWidth() / this.data.getWidth();

      g.setColor(Color.white);
      g.fillRect(0, 0, getWidth(), getHeight());

      g.setColor(Color.black);
      for (y = 0; y < this.data.getHeight(); y++) {
            g.drawLine(0, y * vcell, getWidth(), y * vcell);
      }
      for (x = 0; x < this.data.getWidth(); x++) {
            g.drawLine(x * hcell, 0, x * hcell,
            getHeight());
      }
```

```
            for (y = 0; y < this.data.getHeight(); y++) {
                  for (x = 0; x < this.data.getWidth(); x++) {
                        if (this.data.getData(x, y)) {
                              g.fillRect(x * hcell, y
                  * vcell, hcell, vcell);
                        }
                  }
            }

            g.setColor(Color.black);
            g.drawRect(0, 0, getWidth() - 1, getHeight() - 1);

      }

      /**
       * Assign a new image data object.
       *
       * @param data
       *            The image data object.
       */

      void setData(final SampleData data) {
            this.data = data;
      }

}
```

The **Sample** class defines a number of properties. The variables are described in Table 12.2.

Table 12.2: Variables for the OCR Application

Variable	Purpose
downSampleBottom	The bottom of the cropping region; used during downsampling.
downSampleLeft	The left side of the cropping region; used during downsampling.
downSampleRight	The right side of the cropping region; used during downsampling.
downSampleTop	The top side of the cropping region; used during downsampling.
entryGraphics	A graphics object that allows the user to draw on the image that corresponds to the drawing area.
entryImage	The image that holds the character that the user is drawing.
lastX	The last x coordinate at which the user was drawing.
lastY	The last y coordinate at which the user was drawing.
pixelMap	Numeric pixel map that will actually be downsampled. This is taken directly from the entryImage.
ratioX	The downsample ratio for the x dimension.
ratioY	The downsample ratio for the y dimension.
sample	The object that will contain the downsampled image.

Most of the actual drawing is handled by the processMouseMotionEvent. A line is drawn when a user drags the mouse from the last mouse position to the current mouse position. The mouse moves faster than the program can accept new values; thus, by drawing the line, we cover missed pixels as best we can. The line is drawn to the off-screen image, and then updated to the user's screen. This is accomplished with the following lines of code.

```
entryGraphics.setColor(Color.black);
entryGraphics.drawLine(lastX,lastY,e.getX(),e.getY());
getGraphics().drawImage(entryImage,0,0,this);
lastX = e.getX();
lastY = e.getY();
```

This method is called repeatedly as the program runs, and whatever the user is drawing is saved to the off-screen image. In the next section, you will learn how to downsample an image. You will then see that the off-screen image from this section is accessed as an array of integers, which allows the image data to be directly manipulated.

Downsampling the Image

Every time a letter is drawn for either training or recognition, it must be downsampled. In this section, we will examine the process by which downsampling is achieved. However, before we discuss the downsampling process, we should discuss how downsampled images are stored.

Storing Downsampled Images

Downsampled images are stored in the **SampleData** class. The **SampleData** class is shown in Listing 12.2.

Listing 12.2: Downsampled Image Data (SampleData.java)

```
package com.heatonresearch.book.introneuralnet.ch12.ocr;


/**
 * Chapter 12: OCR and the Self Organizing Map
 *
 * SampleData: Holds sampled data that will be used to train
 * the neural network.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class SampleData implements Comparable<SampleData>,
      Cloneable {

      /**
       * The downsampled data as a grid of booleans.
       */
      protected boolean grid[][];

      /**
       * The letter.
       */
      protected char letter;

      /**
       * The constructor
       *
       * @param letter
       *            What letter this is
       * @param width
       *            The width
       * @param height
```

```
     *            The height
     */
    public SampleData(final char letter, final int width,
          final int height) {
          this.grid = new boolean[width][height];
          this.letter = letter;
    }

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

    /**
     * Create a copy of this sample
     *
     * @return A copy of this sample
     */
    @Override
    public Object clone()

    {

          final SampleData obj = new SampleData(
    this.letter, getWidth(),
                      getHeight());
          for (int y = 0; y < getHeight(); y++) {
                for (int x = 0; x < getWidth(); x++) {
                      obj.setData(x, y, getData(x, y));
                }
          }
          return obj;
    }

    /**
     * Compare this sample to another, used for sorting.
     *
     * @param o
     *            The object being compared against.
     * @return Same as String.compareTo
     */
```

```
public int compareTo(final SampleData o) {
      final SampleData obj = o;
      if (this.getLetter() > obj.getLetter()) {
            return 1;
      } else {
            return -1;
      }
}

/**
 * Get a pixel from the sample.
 *
 * @param x
 *            The x coordinate
 * @param y
 *            The y coordinate
 * @return The requested pixel
 */
public boolean getData(final int x, final int y) {
      return this.grid[x][y];
}

/**
 * Get the height of the down sampled image.
 *
 * @return The height of the downsampled image.
 */
public int getHeight() {
      return this.grid[0].length;
}

/**
 * Get the letter that this sample represents.
 *
 * @return The letter that this sample represents.
 */
public char getLetter() {
      return this.letter;
}

/**
 * Get the width of the downsampled image.
 *
 * @return The width of the downsampled image
 */
```

```
	public int getWidth() {
		return this.grid.length;
	}

	/**
	 * Set one pixel of sample data.
	 *
	 * @param x
	 *            The x coordinate
	 * @param y
	 *            The y coordinate
	 * @param v
	 *            The value to set
	 */
	public void setData(final int x, final int y, final boolean v)
{
		this.grid[x][y] = v;
	}

	/**
	 * Set the letter that this sample represents.
	 *
	 * @param letter
	 *            The letter that this sample represents.
	 */
	public void setLetter(final char letter) {
		this.letter = letter;
	}

	/**
	 * Convert this sample to a string.
	 *
	 * @return Just returns the letter that this sample is
	 * assigned to.
	 */
	@Override
	public String toString() {
		return "" + this.letter;
	}

}
```

As you can see, this class represents a 5 X 7 grid. All downsampled images are stored in this class. The **SampleData** class includes methods for setting and accessing the data associated with the downsampled grid. The **SampleData** class also contains a method named clone that is used to create exact duplicates of an image.

Negating Size and Position

As mentioned earlier, all images are downsampled before being used. This facilitates the processing of images by the neural network, since size and position do not have to be considered. This is particularly important, since the drawing area is large enough to allow a user to draw letters of different sizes. Downsampling results in images of consistent size. In this section, I will explain how this is done.

When you draw an image, the first thing the program does is draw a box around the boundaries of your letter. This allows the program to eliminate all of the white space. The process is performed inside the **downsample** method of the **Entry** class. As you draw a character, the character is also drawn on the **entryImage** instance variable of the **Entry** object. In order to crop this image, and eventually downsample it, we must grab its bit pattern. This is done using the **PixelGrabber** class, as shown here:

```
final int w = this.entryImage.getWidth(this);
final int h = this.entryImage.getHeight(this);

final PixelGrabber grabber = new PixelGrabber(entryImage,
                                              0,0,w,h,true);
grabber.grabPixels();
pixelMap = (int[])grabber.getPixels();
```

After the program processes this code, the **pixelMap** variable, which is an array of **int** datatypes, contains the bit pattern of the image. The next step is to crop the image and remove any white space. Cropping is accomplished by dragging four imaginary lines, one from the top, one from the left, one from the bottom, and one from the right side of the image. These lines stop as soon as they encounter a pixel containing part of the image. The four lines then snap to the outer edges of the image. The **hLineClear** and **vLineClear** methods both accept a parameter that indicates the line to scan, and return **true** if that line is clear. The program works by calling **hLineClear** and **vLineClear** until they cross the outer edges of the image. The horizontal line method (**hLineClear**) is shown here:

```
protected boolean hLineClear(int y)
{
  final int w = this.entryImage.getWidth(this);
  for ( int i=0;i<w;i++ ) {
    if ( this.pixelMap[(y*w)+i] !=-1 )
      return false;
  }
  return true;
}
```

As you can see, the horizontal line method accepts a **y** coordinate that specifies the horizontal line to be checked. The program then loops through each **x** coordinate on this row, checking to see if there are any pixel values. The value of -1 indicates white and is ignored. The **findBounds** method uses **hLineClear** and **vLineClear** to calculate the four edges. The beginning of this method is shown here:

```
protected void findBounds(int w,int h)
{
  // top line
  for ( int y=0;y<h;y++ ) {
    if ( !hLineClear(y) ) {
      this.downSampleTop=y;
      break;
    }

  }
  // bottom line
  for ( int y=h-1;y>=0;y-- ) {
    if ( !hLineClear(y) ) {
      this.downSampleBottom=y;
      break;
    }
  }
```

To calculate the top line of the cropping rectangle, the program starts at 0 and continues to the bottom of the image. The first non-clear line encountered is established as the top of the cropping rectangle. The same process, is carried out in reverse to determine the bottom of the image. The processes to determine the left and right boundaries are carried out in the same way.

Performing the Downsampling

Once the cropping has taken place, the image must be downsampled. This involves reducing the image to a 5 X 7 resolution. To understand how to reduce an image to 5 X 7, begin by thinking of an imaginary grid being drawn on top of the high-resolution image. The grid divides the image into regions, five across and seven down. If any pixel in a region is filled, then the corresponding pixel in the 5 X 7 downsampled image is also filled in. Most of the work done by this process is accomplished inside the **downSampleRegion** method, as shown here:

```
protected boolean downSampleRegion(final int x, final int y) {
  final int w = this.entryImage.getWidth(this);
  final int startX = (int) (this.downSampleLeft + (x
    * this.ratioX));
  final int startY = (int) (this.downSampleTop + (y *
    this.ratioY));
  final int endX = (int) (startX + this.ratioX);
  final int endY = (int) (startY + this.ratioY);
```

```
    for (int yy = startY; yy <= endY; yy++) {
      for (int xx = startX; xx <= endX; xx++) {
        final int loc = xx + (yy * w);

        if (this.pixelMap[loc] != -1) {
          return true;
        }
      }
    }

    return false;
  }
```

The **downSampleRegion** method accepts the number of the region to be calculated. The starting and ending **x** and **y** coordinates are then calculated. The **downSampleLeft** method is used to calculate the first **x** coordinate for the region specified. This is the left side of the cropping rectangle. Then **x** is multiplied by **ratioX**, which is the ratio used to indicate the number of pixels that make up each region. It allows us to determine where to place **startX**. The starting **y** position, **startY**, is calculated in the same way. Next, the program loops through every **x** and **y** in the specified region. If even one pixel in the region is filled, the method returns **true**. The **downSampleRegion** method is called for each region in the image. The final result is a reduced copy of the image, stored in the **SampleData** class. The class is a wrapper class that contains a 5 X 7 array of Boolean values. It is this structure that forms the input for both training and character recognition.

Using the Self-Organizing Map

The downsampled character pattern that is drawn by the user is now fed to the input neurons of the self-organizing map. There is one input neuron for every pixel in the downsampled image. Because the downsampled image is a 5 X 7 grid, there are 35 input neurons.

The neural network communicates which letter it thinks the user drew through the output neurons. The number of output neurons always matches the number of unique letter samples provided. Since 26 letters were provided in the sample, there are 26 output neurons. If this program were modified to support multiple samples per individual letter, there would still be 26 output neurons.

In addition to input and output neurons, there are also connections between the individual neurons. These connections are not all equal. Each connection is assigned a weight. The weights are ultimately the only factors that determine what the network will output for a given input pattern. In order to determine the total number of con-

nections, you must multiply the number of input neurons by the number of output neurons. A neural network with 26 output neurons and 35 input neurons will have a total of 910 connection weights. The training process is dedicated to finding the correct values for these weights.

The recognition process begins when the user draws a character and then clicks the "Recognize" button. First, the letter is downsampled to a 5 X 7 image. This downsampled image must be copied from its 2-dimensional array to an array of doubles that will be fed to the input neurons, as seen here:

```
this entry.downSample();

final double input[] = new double[5*7];
int idx=0;
final SampleData ds = this.sample.getData();
for ( int y=0;y<ds.getHeight();y++ ) {
   for ( int x=0;x<ds.getWidth();x++ ) {
      input[idx++] = ds.getData(x,y)?.5:-.5;
   }
}
```

The above code does the conversion. Neurons require floating point input; therefore, the program uses the value of 0.5 to represent a black pixel and -0.5 to represent a white pixel. The 5 X 7 array of 35 values is fed to the input neurons. This is accomplished by passing the input array to the neural network's winner method. This method will identify which of the 35 neurons won, and will store this information in the best integer.

```
final int best = net.winner ( input , normfac , synth ) ;
final char map[] = mapNeurons();

JOptionPane.showMessageDialog(this,
   "  " + map[best] + "   (Neuron #"
   + best + " fired)","That Letter Is",
   JOptionPane.PLAIN_MESSAGE);
```

Knowing the winning neuron is not very helpful, because it does not show you which letter was actually recognized. To determine which neuron is associated with each letter, the network must be fed each letter to see which neuron wins. For example, if you were to feed the training image for "J" into the neural network, and neuron #4 was returned as the winner, you would know that neuron #4 is the neuron that was trained to recognize J's pattern. This process is accomplished by calling the **mapNeurons** method. The **mapNeurons** method returns an array of characters. The index of each array element corresponds to the neuron number that recognizes the particular character.

Training the Neural Network

Learning is the process of selecting a neuron weight matrix that will correctly recognize input patterns. A self-organizing map learns by constantly evaluating and optimizing its weight matrix. To do this, a starting weight matrix must be established. This is accomplished by selecting random numbers. This weight matrix will likely do a poor job of recognizing letters, but it will provide a starting point.

Once the initial random weight matrix is created, the training can begin. First, the weight matrix is evaluated to determine its current error level. The error is determined by evaluating how well the training inputs (the letters you created) map to the output neurons. The error is calculated by the **`evaluateErrors`** method of the **`KohonenNetwork`** class. When the error level is low, say below 10%, the process is complete.

The training process begins when the user clicks the "Begin Training" button. This begins the training and the number of input and output neurons are calculated. First, the number of input neurons is determined from the size of the downsampled image. Since the height is seven and the width is five for this example, the number of input neurons is 35. The number of output neurons matches the number of characters the program has been given.

This part of the program can be modified if you want to train it with more than one sample per letter. For example, if you want to use 4 samples per letter, you will have to make sure that the output neuron count remains 26, even though 104 input samples will be provided for training—4 for each of the 26 letters.

The training runs in a background thread as a Java **`run`** method. The signature for the run method is shown here:

```
public void run()
```

First, we calculate the number of input neurons needed. This is the product of the downsampled image's height and width.

```
try {
  final int inputNeuron = OCR.DOWNSAMPLE_HEIGHT
        * OCR.DOWNSAMPLE_WIDTH;
  final int outputNeuron = this.letterListModel.size();
```

Next, the training set is allocated. This is a 2-dimensional array with rows equal to the number of training elements, which in this example is the 26 letters of the alphabet. The number of columns is equal to the number of input neurons.

```
  final double set[][] = new double[this.letterListModel.size()]
[inputNeuron];
```

We then loop through all of the letter samples.

```
  for (int t = 0; t < this.letterListModel.size(); t++) {
      int idx = 0;
```

A letter's sample data is obtained.

```
      final SampleData ds = (SampleData) this.letterListModel
            .getElementAt(t);
```

The data is transferred into the network and the Boolean **true** and **false** are transformed to 0.5 and -0.5.

```
      for (int y = 0; y < ds.getHeight(); y++) {
        for (int x = 0; x < ds.getWidth(); x++) {
            set[t][idx++] = ds.getData(x, y) ? .5 : -.5;
        }
      }
  }
```

The training data has been created, so the new **SelfOrganizingMap** object is now created. This object will use multiplicative normalization, which was discussed in chapter 11.

```
  this.net = new SelfOrganizingMap(inputNeuron, outputNeuron,
        NormalizationType.MULTIPLICATIVE);
```

A **TrainSelfOrganizingMap** object is now created to train the self-organizing map just created. This trainer uses the subtractive training method, which was also discussed in chapter 11.

```
  final TrainSelfOrganizingMap train = new TrainSelfOrganizingMap(
        this.net, set,LearningMethod.SUBTRACTIVE,0.5);
```

The number of training **tries** is tracked.

```
  int tries = 1;
```

The number of tries and the error information are updated on the window for each training iteration.

```
  do {
      train.iteration();
      update(tries++, train.getTotalError(), train.getBestEr-
ror());
  } while ((train.getTotalError() > MAX_ERROR) && !this.halt);

this.halt = true;
  update(tries, train.getTotalError(), train.getBestError());

} catch (final Exception e) {
```

When the training error has reached an acceptable level, the final training numbers are displayed.

```
  e.printStackTrace();
  JOptionPane.showMessageDialog(this, "Error: " + e, "Training",
        JOptionPane.ERROR_MESSAGE);
}
}
```

The neural network is now ready to use.

Beyond This Example

The program presented here is only capable of recognizing individual letters, one at a time. In addition, the sample data provided only includes support for the uppercase letters of the Latin alphabet. There is nothing in this program that would prevent you from using both upper and lowercase characters, as well as digits. If you train the program for two sets of 26 letters each and 10 digits, the program will require 62 training sets.

You can quickly run into problems with such a scenario. The program will have a very hard time differentiating between a lowercase "o", an uppercase "O, and the digit zero (0). The problem of discerning between them cannot be handled by the neural network. Instead, you will have to examine the context in which the letters and digits appear.

Many layers of complexity will be added if the program is expanded to process an entire page of writing at one time. Even if the page is only text, it will be necessary for the program to determine where each line begins and ends. Additionally, spaces between letters will need to be located so that the individual characters can be fed to the self-organizing map for processing.

If the image being scanned is not pure text, then the job becomes even more complex. It will be necessary for the program to scan around the borders of the text and graphics. Some lines may be in different fonts, and thus be of different sizes. All of these issues will need to be considered to extend this program to a commercial grade OCR application.

Another limitation of this sample program is that only one drawing can be defined per character. You might want to use three different handwriting samples for a letter, rather than just one. The underlying neural network classes will easily support this feature. This change can be implemented by adding a few more classes to the user interface. To do so, you will have to modify the program to accept more training data than the number of output neurons.

As you can see, there are many considerations that will have to be made to expand this application into a commercial-grade application. In addition, you will not be able to use just a single neural network. It is likely several different types of neural networks will be required to accomplish the tasks mentioned.

Chapter Summary

This chapter presented a practical application of the self-organizing map. You were introduced to the concept of OCR and the uses of this technology. The example presented mimics the OCR capabilities of a PDA. Characters are input when a user draws on a high-resolution box. Unfortunately, this resolution is too high to be directly presented to the neural network. To resolve this problem, we use the techniques of cropping and downsampling to transform the image into a second image that has a much lower resolution.

Once the image has been entered, it must be cropped. Cropping is the process by which extra white space is removed. The program automatically calculates the size of any white space around the image. A rectangle is then plotted around the boundary between the image and the white space. Using cropping has the effect of removing position dependence. It does not matter where the letter is drawn, since cropping eliminates all non-essential areas.

Once the image has been cropped, it must be downsampled. Downsampling is the process by which a high-resolution image is transformed into a lower resolution image. A high-resolution image is downsampled by breaking it up into a number of regions that are equal to the number of pixels in the downsampled image. Each pixel in the downsampled image is assigned the average color of the corresponding region in the high-resolution image. The resulting downsampled image is then fed to either the training or recollection process of the neural network.

The next chapter will present another application of neural networks, bots. Bots are computer programs that can access web sites and perform automated tasks. In doing so, a bot may encounter a wide range of data. Additionally, this data may not be uniformly formatted. A neural network is ideal for use in understanding this data.

Vocabulary

Downsample

Optical character recognition (OCR)

Questions for Review

1. Describe how an image is downsampled.

2. Why is downsampling necessary?

3. How do the dimensions of the downsampled image relate to the input neurons of the self-organizing map?

4. What would be the advantages and disadvantages of using a larger number of input neurons for the self-organizing map used for OCR in this chapter?

5. The OCR application in this chapter drew four imaginary lines over the drawn character. These lines formed a square that fit exactly around the character the user drew. What is the purpose of the imaginary lines?