CHAPTER 11: UNDERSTANDING THE SELF-ORGANIZING MAP

- What is a Self-Organizing Map?
- How is a Self-Organizing Map Used to Classify Patterns?
- Training a Self-Organizing Map
- Dealing with Neurons that do not Learn to Classify

In chapter 5, you learned about the feedforward backpropagation neural network. While the feedforward architecture is commonly used for neural networks, it is not the only option available. In this chapter, we will examine another architecture commonly used for neural networks, the self-organizing map (SOM).

The self-organizing map, sometimes called a Kohonen neural network, is named after its creator, Tuevo Kohonen. The self-organizing map differs from the feedforward backpropagation neural network in several important ways. In this chapter, we will examine the self-organizing map and see how it is implemented. Chapter 12 will continue by presenting a practical application of the self-organizing map, optical character recognition.

Introducing the Self-Organizing Map

The self-organizing map differs considerably from the feedforward backpropagation neural network in both how it is trained and how it recalls a pattern. The self-organizing map does not use an activation function or a threshold value.

In addition, output from the self-organizing map is not composed of output from several neurons; rather, when a pattern is presented to a self-organizing map, one of the output neurons is selected as the "winner." This "winning" neuron provides the output from the self-organizing map. Often, a "winning" neuron represents a group in the data that is presented to the self-organizing map. For example, in chapter 12 we will examine an OCR program that uses 26 output neurons that map input patterns to the 26 letters of the Latin alphabet.

The most significant difference between the self-organizing map and the feedforward backpropagation neural network is that the self-organizing map trains in an unsupervised mode. This means that the self-organizing map is presented with data, but the correct output that corresponds to the input data is not specified.

It is also important to understand the limitations of the self-organizing map. You will recall from earlier discussions that neural networks without hidden layers can only be applied to certain problems. This is the case with the self-organizing map. Self-organizing maps are used because they are relatively simple networks to construct and can be trained very rapidly.

How a Self-Organizing Map Recognizes a Pattern

We will now examine how the self-organizing map recognizes a pattern. We will begin by examining the structure of the self-organizing map. You will then be instructed on how to train the self-organizing map to properly recognize the patterns you desire.

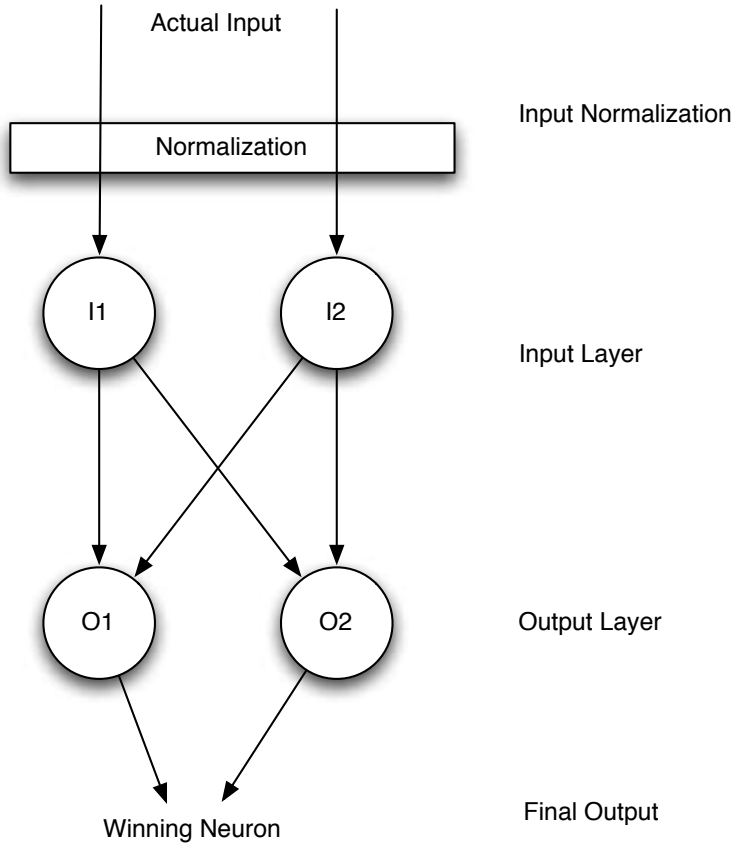
The Structure of the Self-Organizing Map

The self-organizing map works differently than the feedforward neural network that we learned about in chapter 5, "Feedforward Backpropagation Neural Networks." The self-organizing map only contains an input neuron layer and an output neuron layer. There is no hidden layer in a self-organizing map.

The input to a self-organizing map is submitted to the neural network via the input neurons. The input neurons receive floating point numbers that make up the input pattern to the network. A self-organizing map requires that the inputs be normalized to fall between -1 and 1. Presenting an input pattern to the network will cause a reaction from the output neurons.

The output of a self-organizing map is very different from the output of a feedforward neural network. Recall from chapter 5 that if we have a neural network with five output neurons, we will receive an output that consists of five values. As noted earlier, this is not the case with the self-organizing map. In a self-organizing map, only one of the output neurons actually produces a value. Additionally, this single value is either `true` or `false`. Therefore, the output from the self-organizing map is usually the index of the neuron that fired (e.g. Neuron #5). The structure of a typical self-organizing map is shown in Figure 11.1.

Figure 11.1: A self-organizing map.



Now that you understand the structure of the self-organizing map, we will examine how the network processes information by considering a very simple self-organizing map. This network will have only two input neurons and two output neurons. The input to be given to the two input neurons is shown in Table 11.1.

Table 11.1: Sample Inputs to a Self-Organizing Map

Input Neuron 1 (I1)	0.5
Input Neuron 2 (I2)	0.75

We must also know the connection weights between the neurons. The connection weights are given in Table 11.2.

Table 11.2: Connection Weights in the Sample Self-Organizing Map

I1 -> O1	0.1
I2 -> O1	0.2
I1 -> O2	0.3
I2 -> O2	0.4

Using these values, we will now examine which neuron will win and produce output. We will begin by normalizing the input.

Normalizing the Input

The self-organizing map requires that its input be normalized. Thus, some texts refer to the normalization as a third layer. However, in this book, the self-organizing map is considered to be a two-layer network, because there are only two actual neuron layers at work.

The self-organizing map places strict limitations on the input it receives. Input to the self-organizing map must be between the values of –1 and 1. In addition, each of the input neurons must use the full range. If one or more of the input neurons were to only accept the numbers between 0 and 1, the performance of the neural network would suffer.

Input for a self-organizing map is generally normalized using one of two common methods, multiplicative normalization and z-axis normalization.

Multiplicative normalization is the simpler of the two methods, however z-axis normalization can sometimes provide a better scaling factor. The algorithms for these two methods will be discussed in the next two sections. We will begin with multiplicative normalization.

Multiplicative Normalization

To perform multiplicative normalization, we must first calculate the vector length of the input data, or vector. This is done by summing the squares of the input vector and then taking the square root of this number, as shown in Equation 11.1.

Equation 11.1: Multiplicative Normalization

$$f = \frac{1}{\sqrt{\sum_{i=0}^{n-1} x_i^2}}$$

The above equation produces the normalization factor that each input is multiplied by to properly scale them. Using the sample data provided in Tables 11.1 and 11.2, the normalization factor is calculated as follows:

```
1.0 / Math.sqrt( (0.5 * 0.5) + (0.75 * 0.75) )
```

This produces a normalization factor of 1.1094.

Z-Axis Normalization

Unlike the multiplicative algorithm for normalization, the z-axis normalization algorithm does not depend upon the actual data itself; instead the raw data is multiplied by a constant. To calculate the normalization factor using z-axis normalization, we use Equation 11.2.

Equation 11.2: Z-Axis Normalization

$$f = \frac{1}{\sqrt{n}}$$

As can be seen in the above equation, the normalization factor is only dependent upon the size of the input, denoted by the variable **n**. This preserves absolute magnitude information. However, we do not want to disregard the actual inputs completely; thus, a synthetic input is created, based on the input values. The synthetic input is calculated using Equation 11.3.

Equation 11.3: Synthetic Input

$$s = f\sqrt{n - l^2}$$

The variable **n** represents the input size. The variable **f** is the normalization factor. The variable **l** is the vector length. The synthetic input will be added to the input vector that was presented to the neural network.

You might be wondering when you should use the multiplicative algorithm and when you should use the z-axis algorithm. In general, you will want to use the z-axis algorithm, since the z-axis algorithm preserves absolute magnitude. However, if most of the training values are near zero, the z-axis algorithm may not be the best choice. This is because the synthetic component of the input will dominate the other near-zero values.

Calculating Each Neuron's Output

To calculate the output, the input vector and neuron connection weights must both be considered. First, the dot product of the input neurons and their connection weights must be calculated. To calculate the dot product between two vectors, you must multiply each of the elements in the two vectors as shown in Equation 11.4.

Equation 11.4: Calculating the SOM Output

$$[0.5 \quad 0.75] * [0.1 \quad 0.2] = (0.5 * 0.75) + (0.1 * 0.2) = 0.395$$

As you can see from the above calculation, the dot product is 0.395. This calculation will have to be performed for each of the output neurons. In this example, we will only examine the calculations for the first output neuron. The calculations necessary for the second output neuron are carried out in the same way.

The output must now be normalized by multiplying it by the normalization factor that was determined in the previous step. You must multiply the dot product of 0.395 by the normalization factor of 1.1094. The result is an output of 0.438213. Now that the output has been calculated and normalized, it must be mapped to a bipolar number.

Mapping to a Bipolar Ranged Number

As you may recall from chapter 2, a bipolar number is an alternate way of representing binary numbers. In the bipolar system, the binary zero maps to –1 and the binary one remains a 1. Because the input to the neural network has been normalized to this range, we must perform a similar normalization on the output of the neurons. To make this mapping, we multiply by two and subtract one. For the output of 0.438213, the result is a final output of –0.123574.

The value –0.123574 is the output of the first neuron. This value will be compared with the outputs of the other neuron. By comparing these values we can determine a "winning" neuron.

Choosing the Winner

We have seen how to calculate the value for the first output neuron. If we are to determine a winning neuron, we must also calculate the value for the second output neuron. We will now quickly review the process to calculate the second neuron.

The same normalization factor is used to calculate the second output neuron as was used to calculate the first output neuron. As you recall from the previous section, the normalization factor is 1.1094. If we apply the dot product for the weights of the second output neuron and the input vector, we get a value of 0.45. This value is multiplied by the normalization factor of 1.1094, resulting in a value of 0.0465948. We can now calculate the final output for neuron 2 by converting the output of 0.0465948 to bipolar, which yields –0.9068104.

As you can see, we now have an output value for each of the neurons. The first neuron has an output value of –0.123574 and the second neuron has an output value of –0.9068104. To choose the winning neuron, we select the neuron that produces the largest output value. In this case, the winning neuron is the second output neuron with an output of –0.9068104, which beats the first neuron's output of –0.123574.

You have now seen how the output of the self-organizing map was derived. As you can see, the weights between the input and output neurons determine this output. In the next section we will see how these weights can be adjusted to produce output that is more suitable for the desired task. The training process modifies these weights and will be described in the next section.

How a Self-Organizing Map Learns

In this section, you will learn how to train a self-organizing map. There are several steps involved in the training process. Overall, the process for training a self-organizing map involves stepping through several epochs until the error of the self-organizing map is below an acceptable level. In this section, you will learn how to calculate the error rate for a self-organizing map and how to adjust the weights for each epoch. You will also learn how to determine when no additional epochs are necessary to further train the neural network.

The training process for the self-organizing map is competitive. For each training set, one neuron will "win." This winning neuron will have its weight adjusted so that it will react even more strongly to the input the next time it sees it. As different neurons win for different patterns, their ability to recognize that particular pattern will increase.

We will first examine the overall process of training the self-organizing map. The self-organizing map is trained by repeating epochs until one of two things happens: If the calculated error is below an acceptable level, the training process is complete. If, on the other hand, the error rate has changed by only a very small amount, this individual cycle will be aborted without any additional epochs taking place. If it is determined that the cycle is to be aborted, the weights will be initialized with random values and a new training cycle will begin.

Learning Rate

The learning rate is a variable that is used by the learning algorithm to adjust the weights of the neurons. The learning rate must be a positive number less than 1, and is typically 0.4 or 0.5. In the following sections, the learning rate will be specified by the symbol alpha.

Generally, setting the learning rate to a higher value will cause training to progress more quickly. However, the network may fail to converge if the learning rate is set to a number that is too high. This is because the oscillations of the weight vectors will be too great for the classification patterns to ever emerge.

Another technique is to start with a relatively high learning rate and decrease this rate as training progresses. This allows rapid initial training of the neural network that is then "fine tuned" as training progresses.

Adjusting Weights

The memory of the self-organizing map is stored inside the weighted connections between the input layer and the output layer. The weights are adjusted in each epoch. An epoch occurs when training data is presented to the self-organizing map and the weights are adjusted based on the results of this data. The adjustments to the weights should produce a network that will yield more favorable results the next time the same training data is presented. Epochs continue as more and more data is presented to the network and the weights are adjusted.

Eventually, the return on these weight adjustments will diminish to the point that it is no longer valuable to continue with this particular set of weights. When this happens, the entire weight matrix is reset to new random values; thus beginning a new cycle. The final weight matrix that will be used will be the best weight matrix from each of the cycles. We will now examine how weights are transformed.

The original method for calculating changes to weights, which was proposed by Kohonen, is often called the additive method. This method uses the following equation:

Equation 11.5: Adjusting the SOM Weights (Additive)

$$w^{t+1} = \frac{w^t + \alpha x}{length(w^t + \alpha x)}$$

The variable **x** is the training vector that was presented to the network. Variable **wt** is the weight of the winning neuron, and the result of the equation is the new weight. The double vertical bars represent the vector length. This equation will be implemented as a method in the self-organizing map example presented later in this chapter.

This additive method generally works well for most self-organizing maps; however, in cases for which the additive method shows excessive instability and fails to converge, an alternate method can be used. This method is called the subtractive method. The subtractive method uses the following equations:

Equation 11.6: Adjusting the SOM Weight (Subtractive)

$$e = x - w^t$$

$$w^{t+1} = w^t + \alpha e$$

These two equations describe the basic transformation that will occur on the weights of the network. In the next section, you will see how these equations are implemented as a Java program, and their use will be demonstrated.

Calculating the Error

Before we can understand how to calculate the error for the neural network, we must first define "error." A self-organizing map is trained in an unsupervised fashion, so the definition of error is somewhat different than the definition with which we are familiar.

In the previous chapter, supervised training involved calculating the error. The error was the difference between the anticipated output of the neural network and the actual output of the neural network. In this chapter, we are examining unsupervised training. In unsupervised training, there is no anticipated output; thus, you may be wondering exactly how we can calculate an error. The answer is that the error we are calculating is not a true error, or at least not an error in the normal sense of the word.

The purpose of the self-organizing map is to classify input into several sets. The error for the self-organizing map, therefore, provides a measure of how well the network is classifying input into output groups. The error itself is not used to modify the weights, as is the case in the backpropagation algorithm. There is no one official way to calculate the error for a self-organizing map, so we will examine two different methods in the following section as we explore how to implement a Java training method.

Implementing the Self-Organizing Map

Now that you have an understanding of how the self-organizing map functions, we will implement one using Java. In this section, we will see how several classes can be used together to create a self-organizing map. Following this section, you will be shown an example of how to use the self-organizing map classes to create a simple self-organizing map. Finally, in chapter 12, you will be shown how to construct a more complex application, based on the self-organizing map, that can recognize handwriting.

First, you must understand the structure of the self-organizing map classes that we are constructing. The classes used to implement the self-organizing map are summarized in Table 11.3.

Table 11.3: Classes Used to Implement the Self-organizing Map

Class	Purpose
NormalizeInput	Normalizes the input for the self-organizing map. This class implements the normalization method discussed earlier in this chapter.
SelfOrganizingMap	This is the main class that implements the self-organizing map.
TrainSelfOrganizingMap	Used to train the self-organizing map.

Now that you are familiar with the overall structure of the self-organizing map classes, we will examine each individual class. You will see how these classes work together to provide self-organizing map functionality. We will begin by examining how the training set is constructed using the **`NormalizeInput`** class.

The SOM Normalization Class

The **`NormalizeInput`** class receives all of the information that it will need from its constructor. The signature for the constructor is shown here:

```
public NormalizeInput(final double input[],
      final NormalizationType type)
```

The constructor begins by storing the type of normalization requested. This can be either multiplicative or z-axis normalization.

```
this.type = type;
```

Next, the normalization factor and synthetic input values are calculated. The **calculateFactors** method is explained in the next section.

```
calculateFactors(input);
```

Finally, the input matrix is created. This includes the synthetic input. The **createInputMatrix** method is described in a section to follow.

```
this.inputMatrix = this.createInputMatrix(input, this.synth);
```

Calculating the Factors

The **calculateFactors** method calculates both the normalization factor and the synthetic input value. The signature for the **calculateFactors** method is shown here:

```
protected void calculateFactors(final double input[]) {
```

First, the input array is converted to a column matrix.

```
final Matrix inputMatrix = Matrix.createColumnMatrix(input);
```

The vector length is then calculated for the **inputMatrix** variable.

```
double len = MatrixMath.vectorLength(inputMatrix);
```

The length of the vector is evaluated, to ensure it has not become too small.

```
len = Math.max(len, SelfOrganizingMap.VERYSMALL);
```

The number of inputs is determined and this value is stored in the **numInputs** variable.

```
final int numInputs = input.length;
```

Next, the type of normalization to be performed is determined.

```
if (this.type == NormalizationType.MULTIPLICATIVE) {
```

If the type of normalization is multiplicative, then the reciprocal of the vector length is used as the normalization factor.

```
  this.normfac = 1.0 / len;
```

Because the normalization method is additive, no synthetic input is needed. We simply set the synthetic input variable to zero, so that it does not have any influence.

```
  this.synth = 0.0;
} else {
```

If z-axis normalization is being used, then the normalization factor is computed as the reciprocal of the square root of the number of inputs.

```
  this.normfac = 1.0 / Math.sqrt(numInputs);
```

Now we must determine the synthetic input.

```
  final double d = numInputs - Math.pow(len,2);
```

If the synthetic input is calculated to be greater than zero, then we multiply it by the normalization factor.

```
  if (d > 0.0) {
    this.synth = Math.sqrt(d) * this.normfac;
  } else {
    this.synth = 0;
  }
}
```

If the synthetic input is less than zero, then we set the synthetic input variable to zero.

Creating the Input Matrix

Now that the normalization factor and synthetic input have been determined, the input matrix can be created. The input matrix is created by the **`createInputMatrix`** method. The signature for the **`createInputMatrix`** method is shown here:

```
protected Matrix createInputMatrix(final double pattern[],
                   final double extra)
```

First, a matrix is created that has one row and columns equal to one more than the length of the pattern. The extra column will hold the synthetic input.

```
final Matrix result = new Matrix(1, pattern.length + 1);
```

Next, all of the values from the pattern are inserted.

```
for (int i = 0; i < pattern.length; i++) {
  result.set(0, i, pattern[i]);
}
```

Finally, the synthetic input is added and the **`result`** variable is returned.

```
result.set(0, pattern.length, extra);

return result;
```

The input matrix is now ready for use.

The SOM Implementation Class

The self-organizing map is implemented in the class **TrainSelfOrganizingMap**. A pattern is presented to the SOM using a method named **winner**. The signature for this method is shown here:

```
public int winner(final double input[]) {
```

This method does little more than create a normalized matrix and send it on to a more advanced version of the **winner** method that accepts a normalized matrix.

```
final NormalizeInput normalizedInput = new NormalizeInput(input,
this.normalizationType);
```

The winning neuron is returned from this more advanced **winner** method.

```
return winner(normalizedInput);
```

This **winner** method accepts a **NormizedInput** object. The signature for this method is shown here:

```
public int winner(final NormalizedInput input)
```

First, a local variable is created to hold the winning neuron. This variable is named **win**.

```
int win = 0;
```

As we progress, we keep track of the output neuron with the greatest output. One of the output neurons will win. We set the **biggest** variable to a very small number before we begin.

```
double biggest = Double.MIN_VALUE;
```

Then we loop over the output neurons.

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

We obtain the row from the weight matrix that corresponds to this output neuron.

```
  final Matrix optr = this.outputWeights.getRow(i);
```

We then obtain the dot product between this row and the input pattern. This is the output for this neuron.

```
  this.output[i] = MatrixMath
    .dotProduct(input.getInputMatrix(), optr)
      * input.getNormfac();
```

The output from the neuron is mapped to a number between –1 and 1.

```
  this.output[i] = (this.output[i]+1.0)/2.0;
```

The number is evaluated to see if this is the biggest output so far.

```
  if (this.output[i] > biggest) {
    biggest = this.output[i];
    win = i;
  }
```

If the output is above one or below zero, it is adjusted as necessary.

```
  if( this.output[i] <0 ) {
    this.output[i]=0;
  }

  if( this.output[i]>1 ) {
    this.output[i]=1;
  }
}
```

The winning neuron is then returned.

```
return win;
```

As you can see, the output is calculated very differently than the output of the feedforward networks seen earlier in this book. The self-organizing map is a competitive neural network; thus, the output from this neural network is the winning neuron.

The SOM Training Class

The self-organizing map is trained using different techniques than those used with the feedforward neural networks demonstrated thus far. The training is performed by a class named **`TrainSelfOrganizingMap`**. This class is implemented like the other training methods; it goes through a series of iterations until the error is sufficiently small. The training iteration is discussed in the next section.

Training Iteration

To perform one **`iteration`** of training, the **`iteration`** method of the **`TrainSelfOrganizingMap`** class is called. The signature for the **`iteration`** method is shown here:

```
public void iteration() throws RuntimeException
```

First, **`evaluateErrors`** is called to determine the current error level. The **`totalError`** variable, which was just calculated, is saved to the **`globalError`** variable.

```
evaluateErrors();
this.totalError = this.globalError;
```

The current error is evaluated to see if it is better than the best error encountered thus far. If so, the weights are copied over the previous best weight matrix.

```
if (this.totalError < this.bestError) {
  this.bestError = this.totalError;
  copyWeights(this.som, this.bestnet);
}
```

The number of neurons that have won, since the last time the errors were calculated, is determined.

```
int winners = 0;
for (int i = 0; i < this.won.length; i++) {
  if (this.won[i] != 0) {
      winners++;
  }
}
```

If there have been too few winners, one is forced.

```
if ((winners < this.outputNeuronCount) && (winners < this.train.
length)) {
  forceWin();
  return;
}
```

The weights are adjusted based on the training from this iteration.

```
adjustWeights();
```

The learning rate is gradually decreased down to 0.01.

```
if (this.learnRate > 0.01) {
  this.learnRate *= this.reduction;
}
```

The weights are then copied from the best network.

```
copyWeights(this.som, this.bestnet);
```

Finally, these weights are normalized.

```
for (int i = 0; i < this.outputNeuronCount; i++) {
  normalizeWeight(this.som.getOutputWeights(), i);
}
```

This completes one training iteration.

Evaluating Errors

Error evaluation is an important part of the training process. It involves determining how many neurons win for a given training pattern. The neuron that has the greatest response to a given training pattern is adjusted to strengthen that win. The errors are calculated using the **`evaluateErrors`** method. The signature for this method is shown here:

```
void evaluateErrors() throws RuntimeException {
```

First, the correction matrix is cleared. The correction matrix will hold the training corrections. These corrections will be applied to the actual neural network later when the **adjustWeights** method is called. The **adjustWeights** method is covered in the next section.

```
this.correc.clear();

for (int i = 0; i < this.won.length; i++) {
  this.won[i] = 0;
}

this.globalError = 0.0;
```

Next, we loop through all the training sets.

```
for (int tset = 0; tset < this.train.length; tset++) {
```

The input is normalized and presented to the neural network.

```
  final NormalizeInput input = new NormalizeInput(this.train[tset],
        this.som.getNormalizationType());
```

The variable **best** will hold the winning neuron.

```
  final int best = this.som.winner(input);
```

The number of times each neuron wins is recorded.

```
  this.won[best]++;
```

The weights for the winning neuron are then obtained.

```
  final Matrix wptr = this.som.getOutputWeights().getRow(best);
```

The length will be calculated and placed in the **length** variable.

```
  double length = 0.0;
  double diff;
```

We now loop over all of the input neurons.

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

The difference between the training set and the corresponding weight matrix entry are calculated.

```
      diff = this.train[tset][i] * input.getNormfac()
            - wptr.get(0, i);
```

The length is calculated by squaring the difference. A length is the square root of the sum of the squares.

```
    length += diff * diff;
```

What is done next depends upon the learning method.

```
    if (this.learnMethod == LearningMethod.SUBTRACTIVE) {
```

For the subtractive method, the difference is added to the winning neuron.

```
      this.correc.add(best, i, diff);
    } else {
```

For the additive method, a work matrix is used. The work matrix is a temporary matrix used to hold the values to be added. The work matrix is a single column with a number of rows equal to the input neurons. The training set multiplied by the corresponding weight value is added and scaled by the learning rate.

```
      this.work.set(0, i, this.learnRate * this.train[tset][i]
            * input.getNormfac() + wptr.get(0, i));
    }
}
```

Finally, the synthetic input is addressed. The difference between the synthetic input and the corresponding weight matrix value is determined.

```
diff = input.getSynth() - wptr.get(0, this.inputNeuronCount);
```

This difference is added to the length.

```
length += diff * diff;
```

If the learning method is additive, then the synthetic difference is simply added.

```
if (this.learnMethod ==LearningMethod.SUBTRACTIVE) {
    this.correc.add(best, this.inputNeuronCount, diff);
} else {
```

The work matrix value for this input neuron is set to the synthetic input added to the corresponding weight matrix value scaled by the learning rate.

```
    this.work
          .set(0, this.inputNeuronCount, this.learnRate
                * input.getSynth()
                + wptr.get(0, this.inputNeuronCount));
}
```

The calculated length of the differences is the error. We then determine if this beats the current error.

```
if (length > this.globalError) {
    this.globalError = length;
}
```

So far, the additive method has not been modifying the correction matrix; rather it has been modifying the work matrix. The correction matrix must now be updated with any changes to be reflected when the **adjustWeights** method is called.

```
  if (this.learnMethod == LearningMethod.ADDITIVE) {
      normalizeWeight(this.work, 0);
      for (int i = 0; i <= this.inputNeuronCount; i++) {
        this.correc.add(best, i, this.work.get(0, i)
              - wptr.get(0, i));
      }
  }

}
```

The error calculation is now complete.

```
this.globalError = Math.sqrt(this.globalError);
}
```

Finally, the length is determined by performing the square root of the error, which is the sum of the differences squared.

Force a Winner

There are times when certain neurons will not win for any training pattern. These neurons are dead weight and should be adjusted to win for a few patterns. The **forceWin** method is used to adjust these neurons. This method is called when too few neurons have been winning. The signature for the **forceWin** method is shown here.

```
void forceWin() throws RuntimeException
```

The variable **best** will hold the winning neuron for each training set iteration. The variable **which** will hold the training set that has the lowest winning neuron.

```
int best, which = 0;
```

First, the output weights and the last output from the self-organizing map to be trained are obtained.

```
final Matrix outputWeights = this.som.getOutputWeights();
```

We then loop over all training sets and see which output neuron has the smallest response. We initialize **dist** to a large value, and continue to lower it as we find increasingly small output values.

```
double dist = Double.MAX_VALUE;
```

Next, we loop over all of the training sets.

```
for (int tset = 0; tset < this.train.length; tset++) {
```

The winning neuron from each training set is obtained.

```
  best = this.som.winner(this.train[tset]);
final double output[] = this.som.getOutput();
```

The neuron is evaluated to see if it has a lower output than any previous neuron encountered. If so, this is the new lowest neuron.

```
  if (output[best] < dist) {
      dist = output[best];
      which = tset;
  }
}
```

We then reprocess the neuron with the lowest output. The training set is normalized so it can be presented to the neural network.

```
final NormalizeInput input = new NormalizeInput(this.train[which],
      this.som.getNormalizationType());
best = this.som.winner(input);
final double output[] = this.som.getOutput();
```

The neuron with the greatest output for the training set that produced the lowest winner is then identified.

```
dist = Double.MIN_VALUE;
int i = this.outputNeuronCount;
while ((i--) > 0) {
  if (this.won[i] != 0) {
      continue;
  }
```

If this neuron is lower, then it is chosen.

```
  if (output[i] > dist) {
      dist = output[i];
      which = i;
  }
}
```

The weights for the lowest neuron are adjusted so that the neuron will respond to this training pattern.

```
for (int j = 0; j < input.getInputMatrix().getCols(); j++) {
  outputWeights.set(which, j,
    input.getInputMatrix().get(0,j));
}

normalizeWeight(outputWeights, which);
```

Finally, the weights are normalized.

Adjust Weights

After the errors have been calculated, the **adjustWeights** method is called. The **adjustWeights** method applies the correction matrix to the actual matrix. The signature for the **adjustWeights** method is shown here.

```
protected double adjustWeights()
```

Set the **result** to zero. The result is the adjusted error value.

```
double result = 0.0;
```

We loop over all of the output neurons.

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

If this output neuron has never won, then continue; there is nothing to be done. The **forceWin** method will likely assist this neuron later.

```
  if (this.won[i] == 0) {
      continue;
  }
```

The reciprocal of the number of times this output neuron has won is calculated.

```
  double f = 1.0 / this.won[i];
```

If using the subtractive method, then this reciprocal is scaled by the learning rate.

```
  if (this.learnMethod == LearningMethod.SUBTRACTIVE) {
      f *= this.learnRate;
  }
```

The vector length of the input weights multiplied by their correction matrix values is calculated.

```
  double length = 0.0;

  for (int j = 0; j <= this.inputNeuronCount; j++) {
      final double corr = f * this.correc.get(i, j);
      this.somLayer.getMatrix().add(i, j, corr);
      length += corr * corr;
  }
```

The longest length is recorded.

```
  if (length > result) {
      result = length;
  }
}
```

Using the learning rate, the correction is then scaled.

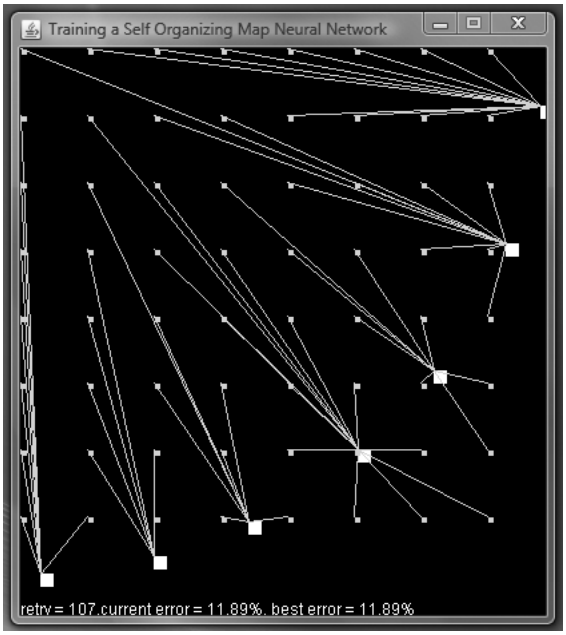
```
result = Math.sqrt(result) / this.learnRate;
return result;
```

Finally, the error value is returned.

Using the Self-organizing Map

We will now examine a simple program that trains a self-organizing map. As the network is trained, you will be shown a graphical display of the weights. The output from this program is shown in Figure 11.2.

Figure 11.2: Training a self-organizing map.



This program contains two input neurons and seven output neurons. Each of the seven output neurons are plotted as white squares. The x-dimension shows the weights between them and the first input neuron and the y-dimension shows the weights between them and the second input neuron. You will see the boxes move as training progresses.

You will also see lines from select points on the grid drawn to each of the squares. These identify which output neuron is winning for the x and y coordinates of that point. Points with similar x and y coordinates are shown as being recognized by the same output neuron.

We will now examine the program, as shown in Listing 11.1.

Listing 11.1: The SOM Training Example (TestSOM.java)

```
package com.heatonresearch.book.introneuralnet.ch11.som;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.text.NumberFormat;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import com.heatonresearch.book.introneuralnet.neural.matrix.
Matrix;
import com.heatonresearch.book.introneuralnet.neural.som.
SelfOrganizingMap;
import com.heatonresearch.book.introneuralnet.neural.som.
TrainSelfOrganizingMap;
import com.heatonresearch.book.introneuralnet.neural.som.
NormalizeInput.NormalizationType;
import com.heatonresearch.book.introneuralnet.neural.som.
TrainSelfOrganizingMap.LearningMethod;

/**
 * Chapter 11: Using a Self Organizing Map
 *
 * TestSOM: Very simple example to test the SOM and show how it
 * works.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class TestSOM extends JFrame implements Runnable {

      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
2772365196327194581L;

      /**
       * How many input neurons to use.
       */
```

```
public static final int INPUT_COUNT = 2;

/**
 * How many output neurons to use.
 */
public static final int OUTPUT_COUNT = 7;

/**
 * How many random samples to generate.
 */
public static final int SAMPLE_COUNT = 100;

/**
 * Startup the program.
 *
 * @param args
 *            Not used.
 */
public static void main(final String args[]) {
      final TestSOM app = new TestSOM();
      app.setVisible(true);
      final Thread t = new Thread(app);
      t.setPriority(Thread.MIN_PRIORITY);
      t.start();

}

/**
 * The unit length in pixels, which is the max of the
 * height and width of
 * the window.
 */
protected int unitLength;

/**
 * How many retries so far.
 */
protected int retry = 1;

/**
 * The current error percent.
 */
protected double totalError = 0;
/**
 * The best error percent.
 */
```

```
	protected double bestError = 0;

	/**
	 * The neural network.
	 */

	protected SelfOrganizingMap net;
	protected double input[][];

	/**
	 * The offscreen image. Used to prevent flicker.
	 */
	protected Image offScreen;

	/**
	 * The constructor sets up the position and size of the
	 * window.
	 */
	TestSOM() {
		setTitle(
"Training a Self Organizing Map Neural Network");
		setSize(400, 450);
		final Toolkit toolkit = Toolkit.getDefaultToolkit();
		final Dimension d = toolkit.getScreenSize();
		setLocation((int) (d.width -
		this.getSize().getWidth()) / 2,
				(int) (d.height -
		this.getSize().getHeight()) / 2);
		setDefaultCloseOperation(
		WindowConstants.DISPOSE_ON_CLOSE);
		setResizable(false);
	}

	/**
	 * Display the progress of the neural network.
	 *
	 * @param g
	 *            A graphics object.
	 */
	@Override
	public void paint(Graphics g) {
		if (this.net == null) {
			return;
		}
		if (this.offScreen == null) {
			this.offScreen = this.createImage((int)
```

```
      getBounds().getWidth(),
                  (int) getBounds().getHeight());
}
g = this.offScreen.getGraphics();
final int width = getContentPane().getWidth();
final int height = getContentPane().getHeight();
this.unitLength = Math.min(width, height);
g.setColor(Color.black);
g.fillRect(0, 0, width, height);

// plot the weights of the output neurons
g.setColor(Color.white);
final Matrix outputWeights =
      this.net.getOutputWeights();

for (int y = 0; y < outputWeights.getRows(); y++) {

      g.fillRect((int) (outputWeights.get(y, 0) *
      this.unitLength),
                  (int) (outputWeights.get(y, 1)
* this.unitLength), 10, 10);

}

// plot a grid of samples to test the net with
g.setColor(Color.green);
for (int y = 0; y < this.unitLength; y += 50) {
      for (int x = 0; x < this.unitLength; x += 50) {
            g.fillOval(x, y, 5, 5);
            final double d[] = new double[2];
            d[0] = x;
            d[1] = y;

            final int c = this.net.winner(d);

            final int x2 = (int) (
      outputWeights.get(c, 0) * this.unitLength);
            final int y2 = (int) (
      outputWeights.get(c, 1) * this.unitLength);

            g.drawLine(x, y, x2, y2);
      }

}

// display the status info
```

```
        g.setColor(Color.white);
        final NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);
        g.drawString("retry = " + this.retry
              + ",current error = "
                    + nf.format(this.totalError * 100)
              + "%, best error = "
                    + nf.format(this.bestError * 100)
              + "%", 0,
                    (int) getContentPane().getBounds().
              getHeight());
        getContentPane().getGraphics().drawImage(
              this.offScreen, 0, 0, this);

  }

  /**
   * Called to run the background thread. The
   * background thread sets up the
   * neural network and training data and begins training
   * the network.
   */
  public void run() {
        // build the training set
        this.input = new double[SAMPLE_COUNT][INPUT_COUNT];

        for (int i = 0; i < SAMPLE_COUNT; i++) {
              for (int j = 0; j < INPUT_COUNT; j++) {
                    this.input[i][j] = Math.random();
              }
        }

        // build and train the neural network
        this.net = new SelfOrganizingMap(INPUT_COUNT,
              OUTPUT_COUNT,
                    NormalizationType.MULTIPLICATIVE);
        final TrainSelfOrganizingMap train =
              new TrainSelfOrganizingMap(
                    this.net, this.input,
              LearningMethod.SUBTRACTIVE,0.5);
        train.initialize();
        double lastError = Double.MAX_VALUE;
        int errorCount = 0;

        while (errorCount < 10) {
```

```
                    train.iteration();
                    this.retry++;
                    this.totalError = train.getTotalError();
                    this.bestError = train.getBestError();
                    paint(getGraphics());

                    if (this.bestError < lastError) {
                          lastError = this.bestError;
                          errorCount = 0;
                    } else {
                          errorCount++;
                    }
              }
        }

}
```

There are several constants that govern the way the SOM training example works. These constants are summarized in Table 11.4.

Table 11.4: TestSOM Constants

Constant	Value	Purpose
INPUT_COUNT	2	How many input neurons to use.
OUTPUT_COUNT	7	How many output neurons to use.
SAMPLE_COUNT	100	How many random samples to generate.

There are two major components to this program. The first is the **`run`** method, which implements the background thread. The background thread processes the training of the SOM. The second is the **`paint`** method, which graphically displays the progress being made by the training process. These two methods will be discussed in the next two sections.

Background Thread

A background thread is used to process the SOM while the application runs. This allows you to see the training progress graphically. The background thread is handled by the **`run`** method. The signature for the **`run`** method is shown here:

```
void run()
```

First, the training set is created using random numbers. These are random points on a grid to which the program will train.

```
// build the training set
this.input = new double[SAMPLE_COUNT][INPUT_COUNT];


for (int i = 0; i < SAMPLE_COUNT; i++) {
  for (int j = 0; j < INPUT_COUNT; j++) {
    this.input[i][j] = Math.random();
  }
}
```

The neural network is then created.

```
// build and train the neural network
this.net = new SelfOrganizingMap(INPUT_COUNT, OUTPUT_
COUNT,NormalizationType.MULTIPLICATIVE);
```

A training class is then created to train the SOM.

```
final TrainSelfOrganizingMap train = new TrainSelfOrganizingMap(
  this.net, this.input);
```

The trainer is initialized.

```
train.initialize();
```

The **lastError** variable is initialized to a very high value and the **errorCount** is set to zero.

```
double lastError = Double.MAX_VALUE;
int errorCount = 0;
```

We then loop until the error has not improved for ten iterations.

```
while (errorCount < 10) {
```

One training iteration is processed and the best error is maintained.

```
  train.iteration();
  this.retry++;
  this.totalError = train.getTotalError();
  this.bestError = train.getBestError();
```

The window is updated with the current grid using the **paint** method described in the next section.

```
  paint(getGraphics());
```

The best error is evaluated to determine if there has been an improvement. If there was no improvement, then **errorCount** is increased by one.

```
  if (this.bestError < lastError) {
    lastError = this.bestError;
      errorCount = 0;
```

```
  } else {
    errorCount++;
  }
}
```

The looping continues until there has been no improvement in the error level for ten iterations.

Displaying the Progress

The current state of the neural network's weight matrix and training is displayed by calling the **`paint`** method. The signature for the **`paint`** method is shown here:

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

If there is no network defined, then there is nothing to draw, so we return.

```
if (this.net == null) {
  return;
}
```

To prevent screen flicker, this program uses an off-screen image to draw the grid. Once the grid is drawn, then the background image is copied to the window. The output is then displayed in one single pass.

The following lines of code check to see if the off-screen image has been created yet. If this image has not been created, then one is created now.

```
if (this.offScreen == null) {
    this.offScreen = this.createImage((int)
    getBounds().getWidth(),
    (int) getBounds().getHeight());
}
```

A graphics object is obtained with which the off-screen image will be drawn.

```
g = this.offScreen.getGraphics();
```

The dimensions of the window are determined.

```
final int width = getContentPane().getWidth();
final int height = getContentPane().getHeight();
```

The minimum of the window height and width is used as the size for a single unit. The entire window is set to black.

```
this.unitLength = Math.min(width, height);
g.setColor(Color.black);
g.fillRect(0, 0, width, height);
```

The output weights are obtained.

```
g.setColor(Color.white);
final Matrix outputWeights = this.net.getOutputWeights();
```

Then we loop through and display the output neurons. These will correspond to the random training data generated in the previous section.

```
for (int y = 0; y < outputWeights.getRows(); y++) {
```

Filled rectangles are drawn that correspond to all of the output neurons.

```
g.fillRect((int) (outputWeights.get(y, 0)
  * this.unitLength),
  (int) (outputWeights.get(y, 1)
  * this.unitLength), 10, 10);

}
```

A grid is then plotted of the samples with which to test the net. We then determine into which of the output neurons each training point is grouped.

```
g.setColor(Color.green);
for (int y = 0; y < this.unitLength; y += 50) {
  for (int x = 0; x < this.unitLength; x += 50) {
```

An oval is drawn at each of the training points.

```
    g.fillOval(x, y, 5, 5);
    final double d[] = new double[2];
```

The training point is presented to the SOM.

```
    d[0] = x;
    d[1] = y;
```

The winning neuron is then obtained.

```
    final int c = this.net.winner(d);
```

The weights for the winning neuron are determined.

```
    final int x2 = (int) (outputWeights.get(c, 0)
      * this.unitLength);
    final int y2 = (int) (outputWeights.get(c, 1)
      * this.unitLength);
```

A line is then drawn from the sample to the winning neuron.

```
    g.drawLine(x, y, x2, y2);
  }
}
```

As training progresses, status information is obtained and the output numbers are properly formatted.

```
// display the status info
g.setColor(Color.white);

final NumberFormat nf = NumberFormat.getInstance();
nf.setMaximumFractionDigits(2);
nf.setMinimumFractionDigits(2);
```

A text string is displayed.

```
g.drawString("retry = "
  + this.retry
  + ",current error = "
  + nf.format(this.totalError * 100)
  + "%, best error = "
  + nf.format(this.bestError * 100)
  + "%", 0,
  (int)getContentPane().getBounds().getHeight());
```

Now that the off screen image is ready, it is displayed to the window.

```
getContentPane().getGraphics().drawImage(this.offScreen, 0, 0,
this);
```

The training progress is now visible to the user.

Chapter Summary

In this chapter we learned about the self-organizing map. The self-organizing map differs from the feedforward backpropagation network in several ways. The self-organizing map uses unsupervised training. This means that it receives input data, but no anticipated output data. It then maps the training samples to each of its output neurons.

A self-organizing map contains only two layers. The network is presented with an input pattern that is passed to the input layer. This input pattern must be normalized to numbers between –1 and 1. The output from this neural network will be one single winning output neuron. The output neurons can be thought of as groups that the self-organizing map has classified.

To train the self-organizing map, we present it with the training elements and see which output neuron "wins." This winning neuron's weights are then modified so that it will respond even more strongly to the pattern that caused it to win the next time the pattern is encountered.

There may also be a case in which one or more neurons fail to ever win. Such neurons are dead weight on the neural network. We must identify such neurons and adjust them so they will recognize patterns that are already recognized by other more "overworked" neurons. This will allow the burden of recognition to fall more evenly over the output neurons.

This chapter presented only a simple example of the self-organizing map. In the next chapter we will apply the self-organizing map to a real-world application. We will see how to use the self-organizing map to recognize handwriting.

Vocabulary

Additive Weight Adjustment

Competitive Learning

Input Normalization

Multiplicative Normalization

Self-Organizing Map

Subtractive Weight Adjustment

Z-Axis Normalization

Questions for Review

1. How many hidden layers are normally used with a self-organizing map? What are their roles?

2. For what types of problems are self-organizing maps normally used?

3. How is competitive learning different from the learning presented earlier in this book, such as backpropagation or genetic algorithms?

4. What output does a self-organizing map produce, and what does this output represent?

5. What is the main advantage of z-axis normalization over multiplicative normalization? When might multiplicative normalization be more useful?