Chapter 8

Self-Organizing Maps

- SOM Structure
- Training a SOM
- Neighborhood Functions
- SOM Error Calculation

So far this book has focused on the feedforward neural network. In this chapter we will look at a different type of neural network. This chapter will focus on the Self-Organizing Map (SOM). Though the SOM can be considered a type of feedforward neural network, it is used very differently. A SOM is used for unsupervised classification.

Self-Organizing maps start with random weights, just like the neural networks we've seen so far. However, the means by which these weights are organized to produce meaningful output is very different than a feedforward neural network. SOM's make use of unsupervised training.

Unsupervised training works very differently than the supervised training that we have been using thus far. Unsupervised training sets only specify the input to the neural network. There is no ideal output provided. Because of this the SOM learns to cluster, or map, your data into a specified number of classes.

The number of output neurons in the SOM defines how many output classes you would like to cluster the input data into. The number of input neurons defines the attributes about each of the training set items that you would like the neural network to cluster based on.

This is exactly the same as was true for feedforward neural networks. You must present data to the neural network through a fixed number of input neurons.

SOM's are only used for classification, which is they divide the input into classes. SOM's are not generally used for regression. Regression is where the neural network predicts one, or more, numeric values. With a SOM, you simply provide the total number of classes, and the SOM automatically your training data into these classes. Additionally, the SOM should be able to organize data it was never trained with into these classes.

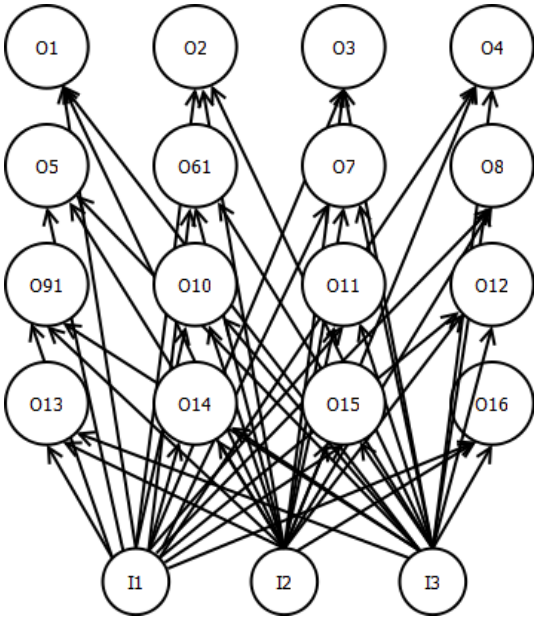
8.1 SOM Structure

The SOM has a structure somewhat similar to the neural networks that we have seen already in this book. However, there are some very important differences. Some of the differences between SOM's and traditional feedforward neural networks are shown here.

- SOM networks have no bias neurons
- SOM networks have no hidden layers
- SOM networks have no activation functions
- SOM networks must have more than one output neuron
- Output neurons in a 1d, 2d, 3d, etc lattice

Consider a SOM that has six input neurons and three output neurons. This SOM is designed to classify data into three groups. The incoming data items that will be classified each have six values. This SOM can be seen in Figure 8.1.

Figure 8.1: A 2D SOM with 16 Outputs and 3 Inputs



As you can see from the above diagram there are no hidden layers and there are no bias neurons. You will also notice that the output neurons are arranged in a lattice. In this case the output neurons are arranged in a 2D grid. This grid will become very important when the neural network is trained. Neurons near each other on the grid will be trained together. The grid does not need to be 2D, grids can be 1D, 3D or of an even higher number of dimensions. The topology of this grid is defined by the neighborhood function that is used to train the neural network. Neighborhood functions will be described later in this chapter.

A SOM can be thought of as reducing a high number of dimensions to a lower number of dimensions. The higher number of dimensions is provided by the input neurons. One dimension for each input neuron. The lower number of dimensions is the configuration of the output neuron lattice. In the case of Figure 8.1, we are reducing three dimensions to two.

In Chapter 1 we saw how a feedforward neural network calculates its output. The method by which a SOM calculates its output is very different. The output of a SOM is the "winning" output neuron for a given input. This winning neuron is also called the Best Matching Unit, or BMU. We will see how to calculate the BMU in the next section.

8.1.1 Best Matching Unit

The best matching unit is the output neuron whose weights most closely matches the input being provided to the neural network. Consider the SOM shown in Figure 8.1. There are 16 output neurons and 3 input neurons. This means that each of the 16 output neurons has 3 weights, one from each of the input neurons. The input to the SOM would also be three numbers, as there are three input neurons. To determine the BMU we determine the output neuron whose three weights most closely match the three input values fed into the SOM.

It is easy enough to calculate the BMU. To do this, we loop over every output neuron and calculate the Euclidean distance between the output neuron's weights and the input values. Whatever output neuron has the lowest Euclidean distance is the BMU. The Euclidean distance is simply the distance between two multi-dimensional points. If you are dealing with two dimensions, the Euclidean distance is simply the length of a line drawn between the two points.

The Euclidean distance is used often in Machine Learning. It is a quick way to compare two arrays of numbers that have the same number of elements. Consider three arrays, named array **a**, array **b** and array **c**. The Euclidean distance between array **a** and array **b** is 10. The Euclidean distance between array **a** and array **c** is 20. In this case, the contents of array a more closely match array b than they do array c.

We will now look at how to actually calculate the Euclidean distance. Equation 8.1 shows this formula.

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \quad (8.1)$$

The above equation shows us the Euclidean distance **d** between two arrays **p** and **q**. The above equation also states that **d(p,q)** is the same as **d(q,p)**. This simply states that the distance is the same no matter what end you start at. Calculation of the Euclidean distance is no more than summing the squares of the difference of each array element. Finally, the square root of this sum is taken. This square root is the Euclidean distance. The output neuron with the lowest Euclidean distance is the BMU.

8.2 Training a SOM

In the previous chapters we learned several methods for training a feedforward neural network. We learned about such techniques as backpropagation, RPROP and LMA. These are all supervised training methods. Supervised training methods work by adjusting the weights of a neural network to produce the correct output for a given input.

A supervised training method will not work for a SOM. SOM networks require unsupervised training. In this section we will learn to train a SOM with an unsupervised method. The training technique generally used for SOM networks is summarized in Equation 8.2.

$$W_v(t+1) = W_v(t) + \theta(v,t)\alpha(t)(D(t) - W_v(t)) \tag{8.2}$$

The above equation shows how the weights of a SOM neural network are updated as training progresses. The current training iteration is noted by the letter **t**, and the next training iteration is noted by **t+1**. Equation 8.2 allows us to see how weight **v** is adjusted for the next training iteration.

The variable **v** denotes that we are performing the same operation on every weight. The variable **W** represents the weights. The symbol **theta** is a special function, called a neighborhood function. The variable **D** represents the current training input to the SOM.

The symbol alpha denotes a learning rate. The learning rate changes for each iteration. This is why Equation 8.2 shows the learning rate with the symbol t, as the learning rate is attached to the iteration. The learning rate for a SOM is said to be monotonically decreasing. Monotonically decreasing simply means that the learning rate only falls, and never increases.

8.2.1 SOM Training Example

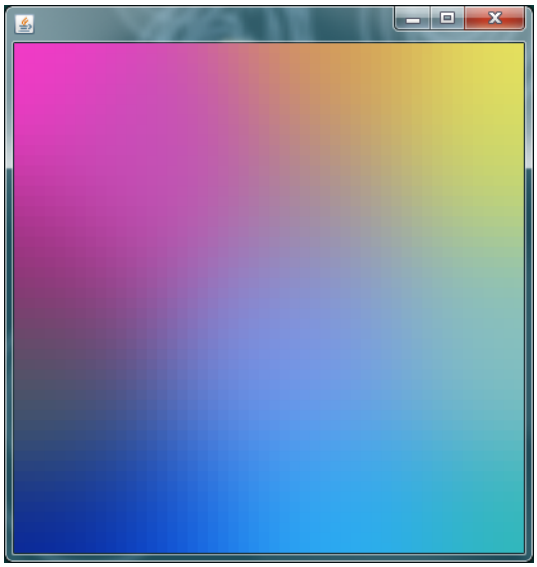
We will now see how to actually train a SOM. We will apply the equation presented in the previous section. However, we will approach it more from an algorithm perspective so we can see the actual learning strategy behind the equation. To see the SOM in action we need a very simple example. We will use a SOM very similar to the one that we saw in Figure 8.1. This SOM will attempt to match colors. However, instead of the 4x4 lattice we saw in Figure 8.1, we will have a lattice of 50x50. This results in a total of 2,500 output neurons.

The SOM will use its three input neurons to match colors to the 2,500 output neurons. The three input neurons will contains the red, blue and green components of the color that

is currently being submitted to the SOM. For training, we will generate 15 random colors. The SOM will learn to cluster these colors.

This sort of training is demonstrated in one of the Encog examples. You can see the output from this example program in Figure 8.2.

Figure 8.2: Mapping Colors



As you can see from the above figure, similar colors are clustered together. Additionally, there are 2,500 output neurons, and only 15 colors that were trained with. This network could potentially recognize up to 2,500 colors. The fact that we trained with only 15 colors means we have quite a few unutilized output neurons. These output neurons will learn to recognize colors that are close to the 15 colors that we trained with.

What you are actually seeing in Figure 8.2 are the weights of SOM network that has been trained. As you can see, even though the SOM was only trained to recognize 15 colors, it is able to recognize quite a few colors. Any new color provided to the SOM will be mapped to one of the 2,500 colors seen in the above image. The SOM can be trained to recognize more classes than it its provided training data. This is defiantly the case in Figure 8.2. The unused output neurons will end up learning to recognize data that falls between elements of the smaller training set.

8.2.2 Training the SOM Example

We will now look at how the SOM network is trained for the colors example. To begin with all of the weights of the SOM network are randomized to values between -1 and +1. A training set is now generated for 15 random colors. Each of these 15 random colors will have three input values. Each of the three input values will be a random value between -1 and +1. For the red, green and blue values -1 represents that the color is totally off, and +1 represents that the color is totally on.

We will see how the SOM is trained for just one of these 15 colors. The same process would be used for the remaining 14 colors. Consider if we were training the SOM for the following random color.

```
-1,1,0.5
```

We will see how the SOM will be trained with this training elemtn.

8.2.3 BMU Calculation Example

The first step would be to compare this input against every output neuron in the SOM and find the Best Matching Unit (BMU). BMU was discussed earlier in this chapter. The BMU can be calculated by finding the smallest Euclidean distance in SOM. The random weights in the SOM are shown here.

```
Output Neuron 1: -0.2423, 0.4837, 0.8723
Output Neuron 2: -0.5437, -0.8734, 0.2234
Ě
Output Neuron 2500: -0.1287, 0.9872, -0.8723
```

Of course, we are skipping quite a few of the output neurons. Normally, you would calculate the Euclidean distance for all 2,500 output neurons. Just calculating the Euclidean distance for the above three neurons should give you an idea of how this is done. Using Equation 8.1 we calculate the Euclidean distance between the input and neuron one.

```
sqrt((-0.2423 - -1)^2 + (0.4837-1)^2 + (0.8723-0.5)^2) = 0.9895
```

A similar process can be used to calculate neuron two.

```
sqrt( (-0.5437 - -1)^2 +  (-0.8734-1)^2 + (0.2234-0.5)^2 ) = 1.947
```

Similarly, we can also calculate neuron 2,500.

```
sqrt((-0.1287 - -1)^2 +  (0.9872-1)^2 + (-0.8723-0.5)^2) = 1.6255
```

Now that we have calculated all of the Euclidean distances, we can determine the BMU. The BMU is neuron one. This is because the distance of 0.9895 is the lowest. Now that we have a BMU, we can update the weights.

8.2.4 Example Neighborhood Functions

We will now loop over every weight in the entire SOM and use Equation 8.2 to update them. The idea is that we will modify the BMU neuron to be more like the training input. However, in addition to modifying the BMU neuron to be more like the training input, we will also modify neurons in the neighborhood around the BMU to be more like the input as well. However, the further a neuron is from the BMU the less of an impact this weight change will have.

Determining the amount of change that will happen to a weight is the job of the neighborhood function. Any radial basis function (RBF) can be used as a neighborhood function. A radial basis function (RBF) is a real-valued function whose value depends only on the distance from the origin.

The Gaussian function is the most common choice for a neighborhood function. The Gaussian function is shown in Equation 8.3.

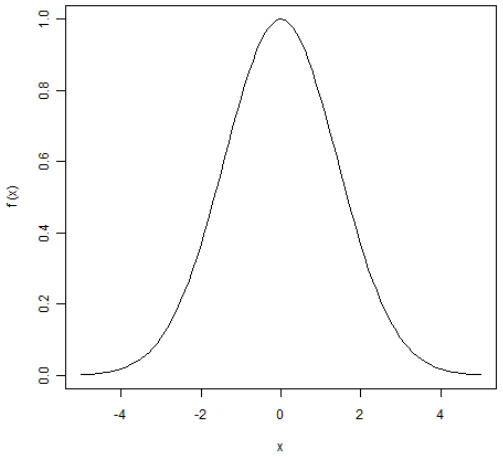
$$f(x_1, x_2, ..., x_n) = e^{-v}, \tag{8.3}$$

This equation continues as follows.

$$v = \sum_{i=0}^{i<n} \frac{x_i - c^2}{2w^2} \tag{8.4}$$

You can see the Gaussian function graphed in Figure 8.3. Where **n** is the number of dimensions, **c** is the center, **w** is the width of the Gaussian curve. The number of dimensions is equal to the number of input neurons. The width starts out at some fixed number and decreases as learning progresses. By the final training iteration the width should be one.

Figure 8.3: Gaussian Function Graphed



From the above figure, you can see that the Gaussian function is a radial basis function. The value only depends on the distance from the origin. That is to say that **f(x)** has the same value regardless of if **x** is -1 or +1.

Looking at Figure 8.3, you can see how the Gaussian function scales the amount of training received by each neuron. The BMU would have zero distance from itself, so the BMU would receive full training of 1.0. As you move further away from the BMU, in either direction, the amount of training quickly falls off. As a neuron that was -4 or +4 from the BMU would receive hardly any training at all.

The Gaussian function is not the only function available for SOM training. Another common choice is the Ricker wavelet, or "Mexican hat" neighborhood function. This function is generally only known as the "Mexican Hat" function in the Americas, due to its resemblance to a "sombrero". In technical nomenclature this function is known as the Ricker wavelet, where it is frequently employed to model seismic data. The equation for the Mexican Hat function is shown in Equation 8.4.

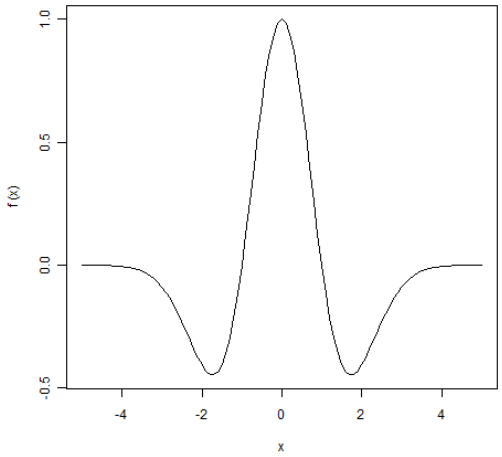
$$f(x_1, x_2, ..., x_n) = (1 - v)e^{-0.5v}, \tag{8.5}$$

This equation continues with the following.

$$v = \sum_{i=0}^{i<n} (x - c)^2 \tag{8.6}$$

You will see the value of the Mexican Hat neighborhood function when you examine its graph in Figure 8.4.

Figure 8.4: Graph of the Mexican Hat



Just as before, the BMU is at the origin of **x**. As you can see from the above chart, the Mexican Hat function actually punishes neurons just at the edge of the radius of the BMU. Their share of the learning will actually be less than zero. As you get even further from the BMU, the learning again returns to zero, and remains at zero. However, neurons just at the edge will have negative learning applied. This can cause the SOM to classes to better differentiate among each other.

8.2.5 Example Weight Update

Now that we have seen how to calculate the BMU and the neighborhood function, we are finally ready to calculate the actual weight update. The formula for the weight update was given in Equation 8.2. We will now calculate the components of this equation for an actual

weight update. Recall that the weights given earlier in this chapter. Neuron one had the following weights.

```
Output Neuron 1: -0.2423, 0.4837, 0.8723
```

Neuron one was the BMU when provided with the following input.

```
-1,1,0.5
```

As you can see, the BMU is somewhat similar to the input. During training, we now want to modify the BMU to be even more like the input. We will also modify the neighbors to receive some of this learning as well.

The "ultimate" learning is to simply copy the input into the weights. Then the Euclidean distance of the BMU will become zero and the BMU is perfectly trained for this input vector. However, we do not want to go that extreme. We simply want to move the weights in the direction of the input. This is where equation 8.2 comes in. We will use Equation 8.2 for every weight in the SOM, not just the BMU weights. For this example, we will start with the BMU. Calculating the update to the first weight, which is -1, we have.

$$w = -0.2423 + (N * r * (-1\acute{\acute{U}}(-0.2423))) \tag{8.7}$$

Before we calculate the above equation, we will take a look at what we are actually doing. Look at the term at the far right. We are taking the difference between the input and the weight. As I said before, the ultimate is just to assign the input to the weight. If we simply added the last term to the weight, the weight would be the same as the input vector.

We do not want to be this extreme. Therefore we scale this difference. First we scale it by the learning rate, which is the variable **r**. Then we also scale it by the result of the neighborhood function, which is **N**. For the BMU, **N** is the value 1.0 and has no effect. If the learning rate were 1.0, as well, then the input would actually be copied to the weight. However a learning rate is never above 1.0. Additionally, the learning rate typically decays as the learning iterations progress. Considering that we have a learning rate of 0.5, our weight update becomes the following.

$$w = -0.2423 + (1.0 * 0.5 * (-1 - -0.2423)) = -0.62115 \tag{8.8}$$

As you can see, the weight moved from -0.2423 to -0.62115. This moved the weight closer to the input of -1. We can perform a similar update for the other two weights that feed into the BMU. This can be seen here.

$$w = 0.4837+(1.0*0.5*(1-0.4837)) = 0.74185 w = 0.8723+(1.0*0.5*(0.5-0.8723)) = 0.68615 \quad (8.9)$$

As you can see, both weights moved closer to the input values.

The neighborhood function is always 1.0 for the BMU. However, consider if we were to calculate the weight update for Neuron Two, which is not the BMU. We would need to calculate the neighborhood function, which was given in Equation 8.3. This assumes we are using a Gaussian neighborhood function. The Mexican Hat function could also be used.

The first step is to calculate **v**. Here we use a width **w** of 3.0. When using Gaussian for a neighborhood function the center **c** is always 0.0.

$$v = ((x1\acute{U}0)/2w^2) + ((x2\acute{U}0)/2w^2) \quad (8.10)$$

We will plug in these values. The values **x1** and **x2** specify how far away the current neuron is from the BMU. The value for **x1** specifies the column distance, and the value of **x2** specifies the row distance. Because neuron two is on the same row, then **x2** will be zero. Neuron two is only one column forward of the BMU, so **x1** will be 1. This gives us the following calculation for v.

$$v = ((0-0)/(2*3)^2) + ((1-0)/(2*3)^2) = 0.0277 \quad (8.11)$$

Now that v has been calculated we can calculate the Gaussian.

$$exp(-v) = 0.9726 \quad (8.12)$$

As you can see, a neuron so close to the BMU gets nearly all the training that the BMU received. Other than using the above value for the neighborhood function, the weight calculation for neuron two is the same as neuron one.

8.3 SOM Error Calculation

While training feedforward neural networks we would typically calculate an error number to indicate if training is successful. A SOM is a unsupervised Neural Network. Because it is unsupervised, the error cannot be calculated by normal means. In fact, because it is unsupervised, there really is no error at all.

If there is no known, ideal data then ere is nothing to calculate an error against. It would be helpful to see some sort of number to indicate the progression of training.

Many neural network implementations do not even report an error for the SOM. Additionally, many articles and books about SOM's do not provide a way to calculate an error. As a result, there is no standard way to calculate an error for a SOM.

Yet, there is a way to report an error. The error for a SOM can be thought of as the worst (or longest) Euclidean distance of any of the best matching units. This is a number that should decrease as training progresses. This can give you an indication of when your SOM is no longer learning. However, it is only an indication. It is not a true error rate. It is also more of a "distance" than an error percent.

8.4 Chapter Summary

This chapter introduced a new neural network and training method. The Self Organizing Map automatically organizes data into classes. No ideal data is provided; the SOM simply groups the input data into similar classes. The number of grouping classes must remain constant.

Training a SOM requires you to determine the best matching unit (BMU). The BMU is the neuron whose weights most closely match the input that we are training. The weights of the BMU are adjusted to more closely match the input that we are training. Additionally, weights near the BMU are also adjusted.

In this next chapter we will look at another neural network architecture. In this chapter we used. This neural network type is called a RBF Neural Network. RBF networks are similar to feedforward neural networks in that they can have multiple output neurons that produce floating point numbers. RBF networks do not classify like SOM's do. Yet RBF networks make use of the same Radial Basis Functions that a SOM uses.