Chapter 2

Self-Organizing Maps

- Self-Organizing Maps
- Neighborhood Functions
- Unsupervised Training
- Dimensionality

Now that you have explored the abstract nature of a neural network introduced in the previous chapter, you will learn about several classic neural network types. This chapter covers one of the earliest types of neural networks that are still useful today. Because neurons can be connected in various ways, many different neural network architectures exist and build on the fundamental ideas from Chapter 1, "Neural Network Basics." We begin our examination of classic neural networks with the self-organizing map (SOM).

The SOM is used to classify neural input data into one of several groups. Training data is provided to the SOM, as well as the number of groups into which you wish to classify these data. While training, the SOM will group these data into groups. Data that have the most similar characteristics will be grouped together. This process is very similar to clustering algorithms, such as k-means. However, unlike k-means, which only groups an initial set of data, the SOM can continue classifying new data beyond the initial data set that was used for training. Unlike most of the neural networks in this book,

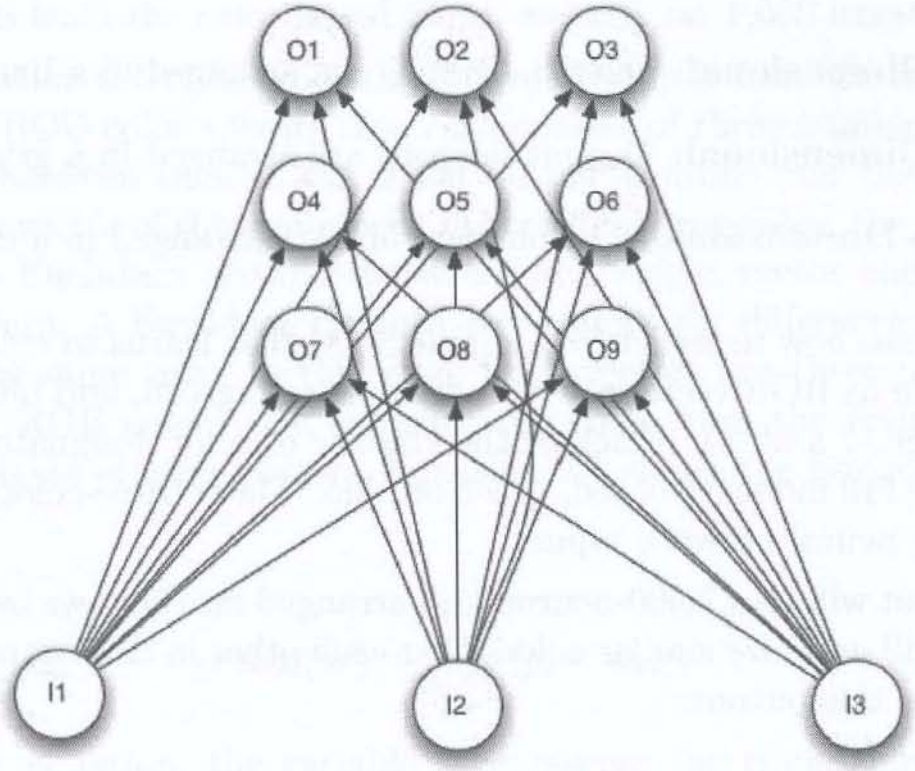
SOM is unsupervised–you do not tell it what groups you expect the training data to fall into. The SOM simply figures out the groups itself, based on your training data, and then it classifies any future data into similar groups. Future classification is performed using what the SOM learned from the training data.

2.1 Self-Organizing Maps

Kohonen (1988) introduced the self-organizing map (SOM), a neural network consisting of an input layer and an output layer. The two-layer SOM is also known as the Kohonen neural network and functions when the input layer maps data to the output layer. As the program presents patterns to the input layer, the output neuron is considered the winner when it contains the weights most similar to the input. This similarity is calculated by comparing the Euclidean distance between the set of weights from each output neuron. The shortest Euclidean distance wins. Calculating Euclidean distance is the focus of the next section.

Unlike the feedforward neural network discussed in Chapter 1, there are no bias values in the SOM. It just has weights from the input layer to the output layer. Additionally, it uses only a linear activation function. Figure 2.1 shows the SOM:

Figure 2.1: Self-Organizing Map



The SOM pictured above shows how the program maps three input neurons to nine output neurons arranged in a three-by-three grid. The output neurons of the SOM are often arranged into a grid, cube, or other higher-dimensional construct. Because the ordering of the output neurons in most neural networks typically conveys no meaning at all, this arrangement is very different. For example, the close proximity of the output neurons #1 and #2 in most neural networks is not significant. However, for the SOM, the closeness of one output neuron to another is important. Computer vision applications make use of the closeness of neurons to identify images more accurately. Convolutional neural networks (CNNs), which will be examined in Chapter 10, "Convolutional Neural Networks," group neurons into overlapping regions based on how close these input neurons are to each other. When recognizing images, it is very important to consider which pixels are near each other. The program recognizes patterns such as edges, solid regions, and lines by looking at pixels near each other.

Common structures for the output neurons of SOMs include the following:

- **One-Dimensional:** Output neurons are arranged in a line.
- **Two-Dimensional:** Output neurons are arranged in a grid.
- **Three-Dimensional:** Output neurons are arranged in a cube.

We will now see how to structure a simple SOM that learns to recognize colors that are given as RGB vectors. The individual red, green, and blue values can range between -1 and +1. Black or the absence of color designates -1, and +1 expresses the full intensity of red, green or blue. These three-color components comprise the neural network input.

The output will be a 2,500-neuron grid arranged into 50 rows by 50 columns. This SOM will organize similar colors near each other in this output grid. Figure 2.2 shows this output:

Figure 2.2: The Output Grid

Although the above figure may not be as clear in the black and white editions of this book as it is in the color e-book editions, you can see similar colors grouped near each other. A single, color-based SOM is a very simple example that allows you to visualize the grouping capabilities of the SOM.

How are SOMs trained? The training process will update the weight matrix, which is 3 by 2,500. To start, the program initializes the weight matrix to random values. Then it randomly chooses 15 training colors.

The training will progress through a series of iterations. Unlike other neural network types, the training for SOM networks involves a fixed number of iterations. To train the color-based SOM, we will use 1,000 iterations.

Each iteration will choose one random color sample from the training set, a collection of RGB color vectors that each consist of three numbers. Likewise, the weights between each of the 2,500 output neurons and the three input neurons are a vector of three numbers. As training progresses, the program will calculate the Euclidean distance between each weight vector and the current training pattern. A Euclidean distance determines the difference between two vectors of the same size. In this case, both vectors are three numbers that represent an RGB color. We compare the color from the training data to the three weights of each neuron. Equation 2.1 shows the Euclidean distance calculation:

$$\mathrm{d}(\mathbf{p}, \mathbf{w}) = \sqrt{\sum_{i=1}^{n}(p_i - w_i)^2} \tag{2.1}$$

In the above equation, the variable p represents the training pattern. The variable w corresponds to the weight vector. By squaring the differences between each vector component and taking the square root of the resulting sum, we calculate the Euclidean distance. This calculation measures the difference between each weight vector and the input training pattern.

The program calculates the Euclidean distance for every output neuron, and the one with the shortest distance is called the best matching unit (BMU). This neuron will learn the most from the current training pattern. The neighbors of the BMU will learn less. To perform this training, the program loops over every neuron and determines the extent to which it should be trained. Neurons that are closer to the BMU will receive more training. Equation 2.2 can make this determination:

$$W_v(t+1) = W_v(t) + \theta(v,t)\alpha(t)(D(t) - W_v(t)) \tag{2.2}$$

In the above equation, the variable t, also known as the iteration number, represents time. The purpose of the equation is to calculate the resulting weight vector Wv(t+1). You will determine the next weight by adding to the current weight, which is Wv(t). The end goal is to calculate how different the cur-

rent weight is from the input vector, and it is done by the clause D(T)-Wv(t). Training the SOM is the process of making a neuron's weights more similar to the training element. We do not want to simply assign the training element to the output neurons weights, making them identical. Rather, we calculate the difference between the training element and the neurons weights and scale this difference by multiplying it by two ratios. The first ratio, represented by θ (theta), is the neighborhood function. The second ratio, represented by α (alpha), is a monotonically decreasing learning rate. In other words, as the training progresses, the learning rate falls and never rises.

The neighborhood function considers how close each output neuron is to the BMU. For neurons that are nearer, the neighborhood function will return a value that approaches 1. For distant neighbors, the neighborhood function will approach 0. This range between 0 and 1 controls how near and far neighbors are trained. Nearer neighbors will receive more of the training adjustment to their weights. In the next section, we will analyze how the neighborhood function determines the training adjustments. In addition to the neighborhood function, the learning rate also scales how much the program will adjust the output neuron.

2.1.1 Understanding Neighborhood Functions

The neighborhood function determines the degree to which each output neuron should receive a training adjustment from the current training pattern. The function usually returns a value of 1 for the BMU. This value indicates that the BMU should receive the most training. Those neurons farther from the BMU will receive less training. The neighborhood function determines this weighting.

If the output neurons are arranged in only one dimension, you should use a simple one-dimensional neighborhood function, which will treat the output as one long array of numbers. For instance, a one-dimensional network might have 100 output neurons that form a long, single-dimensional array of 100 values.

A two-dimensional SOM might take these same 100 values and represent them as a grid, perhaps of 10 rows and 10 columns. The actual structure remains the same; the neural network has 100 output neurons. The only dif-

ference is the neighborhood function. The first would utilize a one-dimensional neighborhood function; the second would use a two-dimensional neighborhood function. The function must consider this additional dimension and factor it into the distance returned.

It is also possible to have three, four, and even more dimensional functions for the neighborhood function. Typically, neighborhood functions are expressed in vector form so that the number of dimensions does not matter. To represent the dimensions, the Euclidian norm (represented by two vertical bars) of all inputs is taken, as seen in Equation 2.3:

$$||p - w|| = \sqrt{\sum_{i=1}^{n}(p_i - w_i)^2} \tag{2.3}$$

For the above equation, the variable p represents the dimensional inputs. The variable w represents the weights. A single dimension has only a single value for p. Calculating the Euclidian norm for *[2-0]* would simply be the following:

$$||2 - 0|| = \sqrt{2^2} = 2 \tag{2.4}$$

Calculating the Euclidean norm for *[2-0, 3-0]* is only slightly more complex:

$$|| \ [2 - 0, 3 - 0] \ || = \sqrt{2^2 + 3^2} = 3.605551 \tag{2.5}$$

The most popular choice for SOMs is the two-dimensional neighborhood function. One-dimensional neighborhood functions are also common. However, neighborhood functions with three or more dimensions are more unusual. Choosing the number of dimensions really comes down to the programmer deciding how many ways an output neuron can be close to another. This decision should not be taken lightly because each additional dimension significantly affects the amount of memory and processing power needed. This additional processing is why most programmers choose two or three dimensions for the SOM application.

It can be difficult to understand why you might have more than three dimensions. The following analogy illustrates the limitations of three dimensions. While at the grocery store, John noticed a package of dried apples. As

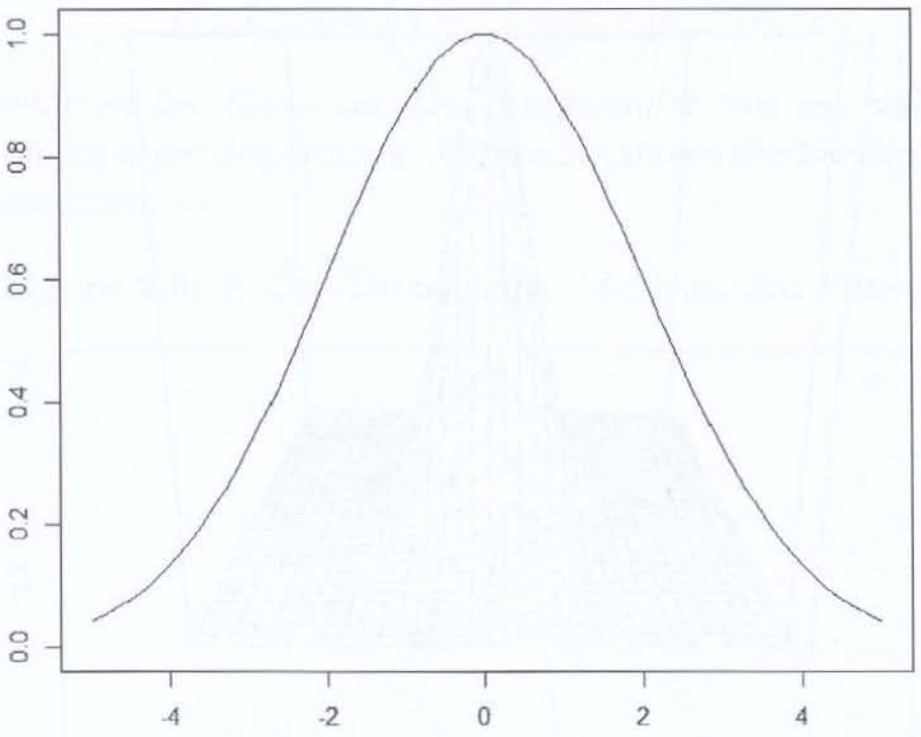
he turned his head to the left or right, traveling in the first dimension, he saw other brands of dried apples. If he looked up or down, traveling in the second dimension, he saw other types of dried fruit. The third dimension, depth, simply gives him more of exactly the same dried apples. He reached behind the front item and found additional stock. However, there is no fourth dimension, which could have been useful to allow fresh apples to be located near to the dried apples. Because the supermarket only had three dimensions, this type of link is not possible. Programmers do not have this limitation, and they must decide if the extra processing time is necessary for the benefits of additional dimensions.

The Gaussian function is a popular choice for a neighborhood function. Equation 2.4 uses the Euclidean norm to calculate the Gaussian function for any number of dimensions:

$$f(x, c, w) = e^{-(w||x-c||)^2} \tag{2.6}$$

The variable x represents the input to the Gaussian function, c represents the center of the Gaussian function, and w represents the widths. The variables x, w and c all are vectors with multiple dimensions. Figure 2.3 shows the graph of the two-dimensional Gaussian function:

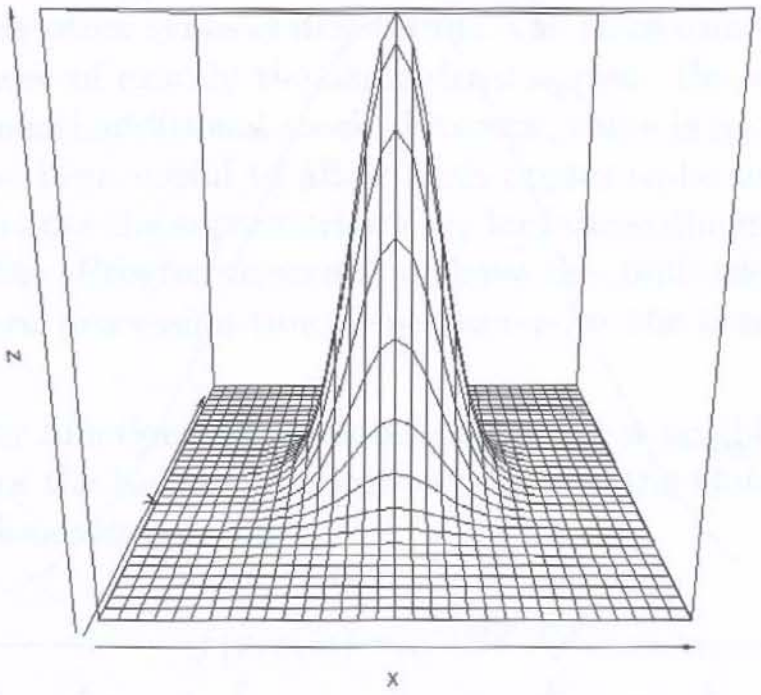
Figure 2.3: A Single-Dimensional Gaussian Function



This figure illustrates why the Gaussian function is a popular choice for a neighborhood function. Programmers frequently use the Gaussian function to show the normal distribution, or bell curve. If the current output neuron is the BMU, then its distance (x-axis) will be 0. As a result, the training percent (y-axis) is 1.0 (100%). As the distance increases either positively or negatively, the training percentage decreases. Once the distance is large enough, the training percent approaches 0.

If the input vector to the Gaussian function has two dimensions, the graph appears as Figure 2.4:

Figure 2.4: A Two-Dimensional Gaussian Function



How does the algorithm use Gaussian constants with a neural network? The center (c) of a neighborhood function is always 0, which centers the function on the origin. If the algorithm moves the center from the origin, a neuron other than the BMU would receive the full learning. It is unlikely you would ever want to move the center from the origin. For a multi-dimensional Gaussian, set all centers to 0 in order to position the curve at the origin.

The only remaining Gaussian parameter is the width. You should set this parameter to something slightly less than the entire width of the grid or array. As training progresses, the width gradually decreases. Just like the learning rate, the width should decrease monotonically.

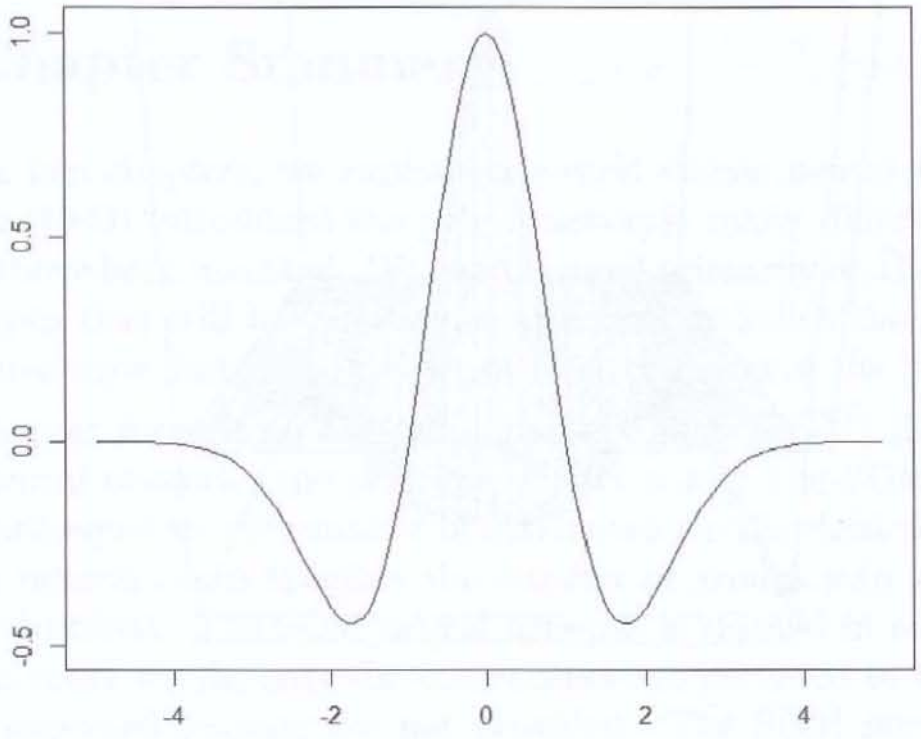
2.1.2 Mexican Hat Neighborhood Function

Though it is the most popular, the Gaussian function is not the only neighborhood function available. The Ricker wave, or Mexican hat function, is another popular neighborhood function. Just like the Gaussian neighborhood function, the vector length of the x dimensions is the basis for the Mexican hat function, as seen in Equation 2.5:

$$f(x, c, w) = \left(1 - \frac{||x - c||^2}{w}\right) e^{-\frac{||x-c||^2}{2w}} \tag{2.7}$$

Much the same as the Gaussian, the programmer can use the Mexican hat function in one or more dimensions. Figure 2.5 shows the Mexican hat function with one dimension:

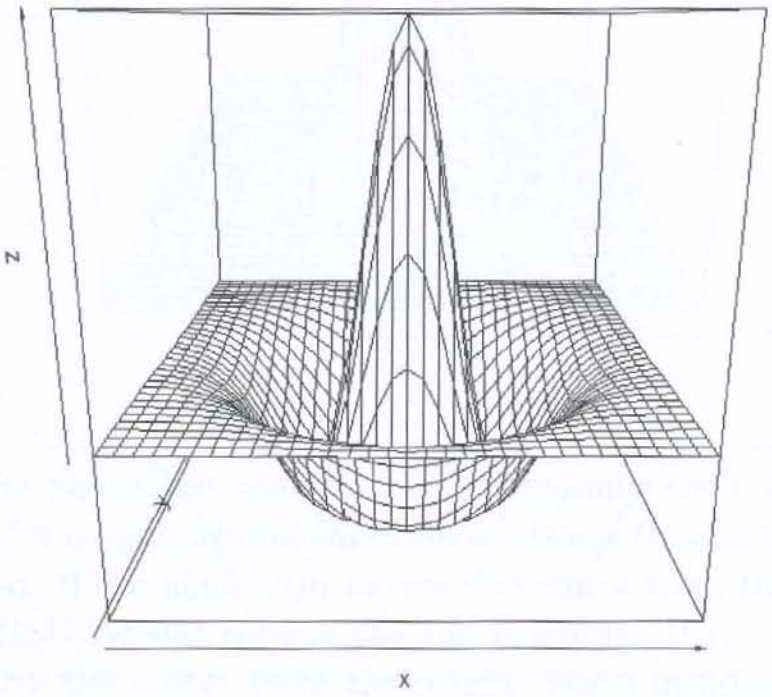
Figure 2.5: A One-Dimensional Mexican Hat Function



You must be aware that the Mexican hat function penalizes neighbors that are between 2 and 4, or -2 and -4 units from the center. If your model seeks to penalize near misses, the Mexican hat function is a good choice.

You can also use the Mexican hat function in two or more dimensions. Figure 2.6 shows a two-dimensional Mexican hat function:

Figure 2.6: A Two-Dimensional Mexican Hat Function



Just like the one-dimensional version, the above Mexican hat penalizes near misses. The only difference is that the two-dimensional Mexican hat function utilizes a two-dimensional vector, which looks more like a Mexican sombrero than the one-dimensional variant. Although it is possible to use more than two dimensions, these variants are hard to visualize because we perceive space in three dimensions.

2.1.3 Calculating SOM Error

Supervised training typically reports an error measurement that decreases as training progresses. Unsupervised models, such as the SOM network, cannot

directly calculate an error because there is no expected output. However, an estimation of the error can be calculated for the SOM (Masters, 1993).

We define the error as the longest Euclidean distance of all BMUs in a training iteration. Each training set element has its own BMU. As learning progresses, the longest distance should decrease. The results also indicate the success of the SOM training since the values will tend to decrease as the training continues.

2.2 Chapter Summary

In the first two chapters, we explained several classic neural network types. Since Pitts (1943) introduced the neural network, many different neural network types have been invented. We have focused primarily on the classic neural network types that still have relevance and that establish the foundation for other architectures that we will cover in later chapters of the book.

This chapter focused on the self-organizing map (SOM) that is an unsupervised neural network type that can cluster data. The SOM has an input neuron count equal to the number of attributes for the data to be clustered. An output neuron count specifies the number of groups into which the data should be clustered. The SOM neural network is trained in an unsupervised manner. In other words, only the data points are provided to the neural network; the expected outputs are not provided. The SOM network learns to cluster the data points, especially the data points similar to the ones with which it trained.

In the next chapter, we will examine two more classic neural network types: the Hopfield neural network and the Boltzmann machine. These neural network types are similar in that they both use an energy function during their training process. The energy function measures the amount of energy in the network. As training progresses, the energy should decrease as the network learns.