Chapter 4

Feedforward Neural Networks

- Classification
- Regression
- Network Layers
- Normalization

In this chapter, we shall examine one of the most common neural network architectures, the feedforword neural network. Because of its versatility, the feedforward neural network architecture is very popular. Therefore, we will explore how to train it and how it processes a pattern.

The term feedforward describes how this neural network processes and recalls patterns. In a feedforward neural network, each layer of the neural network contains connections to the next layer. For example, these connections extend forward from the input to the hidden layer, but no connections move backward. This arrangement differs from the Hopfield neural network featured in the previous chapter. The Hopfield neural network was fully connected, and its connections were both forward and backward. We will analyze the structure of a feedforward neural network and the way it recalls a pattern later in the chapter.

We can train feedforward neural networks with a variety of techniques from the broad category of backpropagation algorithms, a form of supervised

training that we will discuss in greater detail in the next chapter. We will focus on applying optimization algorithms to train the weights of a neural network in this chapter. If you need more information about optimization algorithms, Volumes 1 and 2 of *Artificial Intelligence for Humans* contain sections on this subject. Although we can employ several optimization algorithms to train the weights, we will primarily direct our attention to simulated annealing.

Optimization algorithms adjust a vector of numbers to achieve a good score from an objective function. The objective function gives the neural network a score based closely on the neural network's output that matches the expected output. This score allows any optimization algorithm to train neural networks.

A feedforward neural network is similar to the types of neural networks that we have already examined. Just like other types of neural networks, the feedforward neural network begins with an input layer that may connect to a hidden layer or to the output layer. If it connects to a hidden layer, the hidden layer can subsequently connect to another hidden layer or to the output layer. Any number of hidden layers can exist.

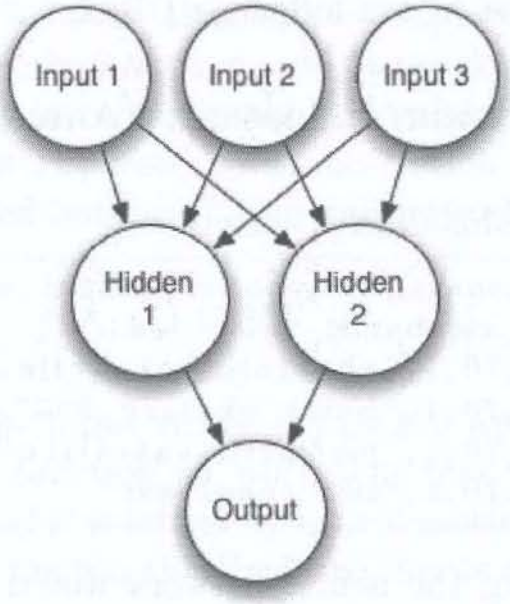
4.1 Feedforward Neural Network Structure

In Chapter 1, "Neural Network Basics," we discussed that neural networks could have multiple hidden layers and analyzed the purposes of these layers. In this chapter, we will focus more on the structure of the input and output neurons, beginning with the structure of the output layer. The type of problem dictates the structure of the output layer. A classification neural network will have an output neuron for each class, whereas a regression neural network will have one output neuron.

4.1.1 Single-Output Neural Networks for Regression

Though feedforward neural networks can have more than one output neuron, we will begin by looking at a single-output neuron network in a regression problem. A regression network is capable of predicting a single numeric value. Figure 4.1 illustrates a single-output feedforward neural network:

Figure 4.1: Single-Output Feedforward Network



This neural network will output a single numeric value. We can use this type of neural network in the following ways:

- **Regression** - Compute a number based on the inputs. (e.g., How many miles per gallon (MPG) will a specific type of car achieve?)
- **Binary Classification** - Decide between two options, based on the inputs. (e.g., Of the given characteristics, which is a cancerous tumor?)

We provide a regression example for this chapter that utilizes data about various car models and predicts the miles per gallon that the car will achieve. You can find this data set at the following URL:

https://archive.ics.uci.edu/ml/datasets/Auto+MPG

A small sampling of this data is shown here:

```
mpg,cylinders,displacement,horsepower,weight,acceleration,
    model_year,origin,car_name
18,8,307,130,3504,12,70,1,"chevrolet chevelle malibu"
15,8,350,165,3693,11,70,1,"buick skylark 320"
18,8,318,150,3436,11,70,1,"plymouth satellite"
16,8,304,150,3433,12,70,1,"amc rebel sst"
```

For a regression problem, the neural network would create columns such as cylinders, displacement, horsepower, and weight to predict the MPG. These values are all fields used in the above listing that specify qualities of each car. In this case, the target is MPG; however, we could also utilize MPG, cylinders, horsepower, weight, and acceleration to predict displacement.

To make the neural network perform regression on multiple values, you might apply multiple output neurons. For example, cylinders, displacement, and horsepower can predict both MPG and weight. Although a multi-output neural network is capable of performing regression on two variables, we don't recommend this technique. You will usually achieve better results with separate neural networks for each regression outcome that you are trying to predict.

4.2 Calculating the Output

In Chapter 1, "Neural Network Basics," we explored how to calculate the individual neurons that comprise a neural network. As a brief review, the output of an individual neuron is simply the weighted sum of its inputs and a bias. This summation is passed to an activation function. Equation 4.1 summarizes the calculated output of a neural network:

$$f(x_i, w_i) = \phi(\sum_i (w_i \cdot x_i)) \tag{4.1}$$

The neuron multiplies the input vector (x) by the weights (w) and passes the result into an activation function (ϕ, phi). The bias value is the last value in the weight vector (w), and it is added by concatenating a 1 value to the input. For example, consider a neuron that has two inputs and a bias. If the inputs were 0.1 and 0.2, the input vector would appear as follows:

```
[0.1, 0.2, 1.0]
```

In this example, add the value 1.0 to support the bias weight. We can also calculate the value with the following weight vector:

```
[0.01, 0.02, 0.3]
```

The values 0.01 and 0.02 are the weights for the two inputs to the neuron. The value 0.3 is the bias. The weighted sum is calculated as follows:

```
(0.1*0.01) + (0.2*0.02) + (1.0*0.3) = 0.305
```

The value 0.305 is then passed to an activation function.

Calculating an entire neural network is essentially a matter of following this same procedure for each neuron in the network. This process allows you to work your way from the input neurons to the output. You can implement this process by creating objects for each connection in the network or by aligning these connection values into matrices.

Object-oriented programming allows you to define an object for each neuron and its weights. This approach can produce very readable code, but it has two significant problems:

- The weights are stored across many objects.
- Performance suffers because it takes many function calls and memory accesses to piece all the weights together.

It is valuable to create weights in the neural network as a single vector. A variety of different optimization algorithms can adjust a vector to perfect a scoring function. *Artificial Intelligence for Humans, Volumes 1 & 2* include a discussion of these optimization functions. Later in this chapter, we will see how simulated annealing optimizes the weight vector for the neural network.

To construct a weight vector, we will first look at a network that has the following attributes:

- Input Layer: 2 neurons, 1 bias
- Hidden Layer: 2 neurons, 1 bias
- Output Layer: 1 neuron

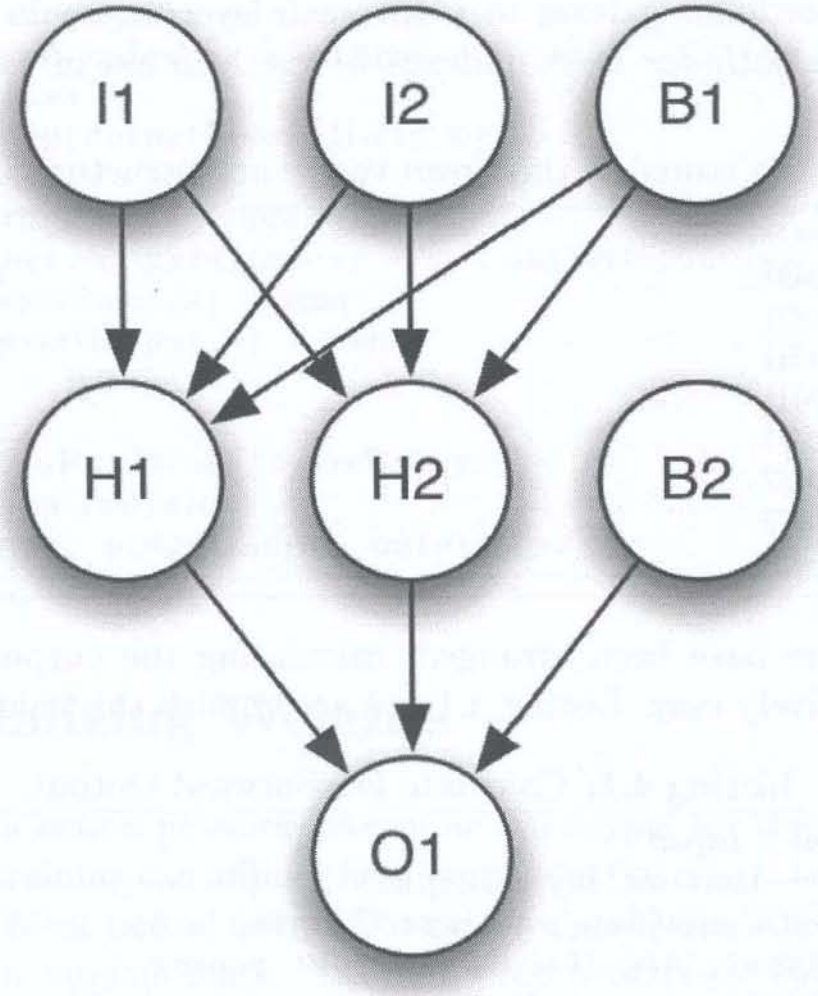
These characteristics give this network a total of 7 neurons.

You can number these neurons for the vector in the following manner:

```
Neuron 0: Output 1
Neuron 1: Hidden 1
Neuron 2: Hidden 2
Neuron 3: Bias 2 (set to 1, usually)
Neuron 4: Input 1
Neuron 5: Input 2
Neuron 6: Bias 1 (set to 1, usually)
```

Graphically, you can see the network as Figure 4.2:

Figure 4.2: Simple Neural Network



You can create several additional vectors to define the structure of the network. These vectors hold index values to allow the quick navigation of the weight vector. These vectors are listed here:

```
layerFeedCounts: [1, 2, 2]
layerCounts: [1, 3, 3]
layerIndex: [0, 1, 4]
layerOutput: [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]
weightIndex: [0, 3, 9]
```

Each vector stores the values for the output layer first and works its way to the input layer. The *layerFeedCounts* vector holds the count of non-bias neurons in each layer. This characteristic is essentially the count of non-bias neurons. The *layerOutput* vector holds the current value of each neuron. Initially, all neurons start with 0.0 except for the bias neurons, which start at 1.0. The *layerIndex* vector holds indexes to where each layer begins in the *layerOuput* vector. The *weightIndex* holds indexes to the location of each layer in the weight vector.

The weights are stored in their own vector and structured as follows:

```
Weight 0: H1->O1
Weight 1: H2->O1
Weight 2: B2->O1
Weight 3: I1->H1
Weight 4: I2->H1
Weight 5: B1->H1
Weight 6: I1->H2
Weight 7: I2->H2
Weight 8: B1->H2
```

Once the vectors have been arranged, calculating the output of the neural network is relatively easy. Listing 4.1 can accomplish this calculation:

Listing 4.1: Calculate Feedforward Output

```
def compute(net, input):
  sourceIndex = len(net.layerOutput)
    - net.layerCounts[len(net.layerCounts) - 1]
  # Copy the input into the layerOutput vector
  array_copy(input, 0, net.layerOutput, sourceIndex, net.
      inputCount)
  # Calculate each layer
  for i in reversed(range(0,len(layerIndex))):
    compute_layer(i)
  # update context values
  offset = net.contextTargetOffset[0]
  # Create result
  result = vector(net.outputCount)
  array_copy(net.layerOutput, 0, result, 0, net.outputCount)
  return result
```

```
def compute_layer(net,currentLayer):
  inputIndex = net.layerIndex[currentLayer]
  outputIndex = net.layerIndex[currentLayer - 1]
  inputSize = net.layerCounts[currentLayer]
  outputSize = net.layerFeedCounts[currentLayer - 1]
  index = this.weightIndex[currentLayer - 1]
  limit_x = outputIndex + outputSize
  limit_y = inputIndex + inputSize
  # weight values
  for x in range(outputIndex,limit_x):
    sum = 0;
    for y in range(inputIndex,limit_y):
      sum += net.weights[index] * net.layerOutput[y]
        net.layerSums[x] = sum
        net.layerOutput[x] = sum
        index = index + 1

  net.activationFunctions[currentLayer - 1]
    .activation_function(
 net.layerOutput, outputIndex, outputSize)
```

4.3 Initializing Weights

The weights of a neural network determine the output for the neural network. The process of training can adjust these weights so the neural network produces useful output. Most neural network training algorithms begin by initializing the weights to a random state. Training then progresses through a series of iterations that continuously improve the weights to produce better output.

The random weights of a neural network impact how well that neural network can be trained. If a neural network fails to train, you can remedy the problem by simply restarting with a new set of random weights. However, this solution can be frustrating when you are experimenting with the architecture of a neural network and trying different combinations of hidden layers and neurons. If you add a new layer, and the network's performance improves, you must ask yourself if this improvement resulted from the new layer or from a new set of weights. Because of this uncertainty, we look for two key attributes

in a weight initialization algorithm:

- How consistently does this algorithm provide good weights?
- How much of an advantage do the weights of the algorithm provide?

One of the most common, yet least effective, approaches to weight initialization is to set the weights to random values within a specific range. Numbers between -1 and +1 or -5 and +5 are often the choice. If you want to ensure that you get the same set of random weights each time, you should use a seed. The seed specifies a set of predefined random weights to use. For example, a seed of 1000 might produce random weights of 0.5, 0.75, and 0.2. These values are still random; you cannot predict them, yet you will always get these values when you choose a seed of 1000.

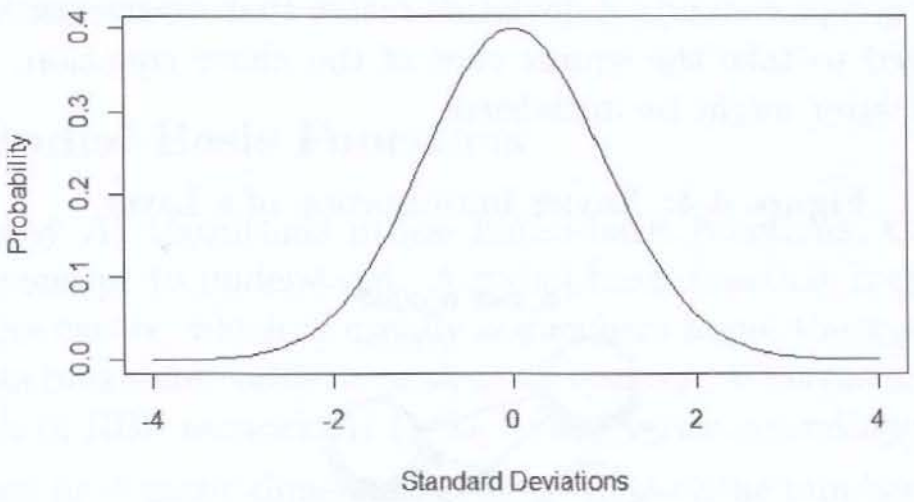
Not all seeds are created equal. One problem with random weight initialization is that the random weights created by some seeds are much more difficult to train than others. In fact, the weights can be so bad that training is impossible. If you find that you cannot train a neural network with a particular weight set, you should generate a new set of weights using a different seed.

Because weight initialization is a problem, there has been considerable research around it. Over the years we have studied this research and added six different weight initialization routines to the Encog project. From our research, the Xavier weight initialization algorithm, introduced in 2006 by Glorot & Bengio, produces good weights with reasonable consistency. This relatively simple algorithm uses normally distributed random numbers.

To use the Xavier weight initialization, it is necessary to understand that normally distributed random numbers are not the typical random numbers between 0 and 1 that most programming languages generate. In fact, normally distributed random numbers are centered on a mean (μ, mu) that is typically 0. If 0 is the center (mean), then you will get an equal number of random numbers above and below 0. The next question is how far these random numbers will venture from 0. In theory, you could end up with both positive and negative numbers close to the maximum positive and negative ranges supported by your computer. However, the reality is that you will more likely see random numbers that are between 0 and three standard deviations from the center.

The standard deviation σ (sigma) parameter specifies the size of this standard deviation. For example, if you specified a standard deviation of 10, then you would mainly see random numbers between -30 and +30, and the numbers nearer to 0 have a much higher probability of being selected. Figure 4.3 shows the normal distribution:

Figure 4.3: The Normal Distribution



The above figure illustrates that the center, which in this case is 0, will be generated with a 0.4 (40%) probability. Additionally, the probability decreases very quickly beyond -2 or +2 standard deviations. By defining the center and how large the standard deviations are, you are able to control the range of random numbers that you will receive.

Most programming languages have the capability of generating normally distributed random numbers. In general, the Box-Muller algorithm is the basis for this functionality. The examples in this volume will either use the built-in normal random number generator or the Box-Muller algorithm to transform regular, uniformly distributed random numbers into a normal distribution. *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms* contains an explanation of the Box-Muller algorithm, but you do not necessarily need to understand it in order to grasp the ideas in this book.

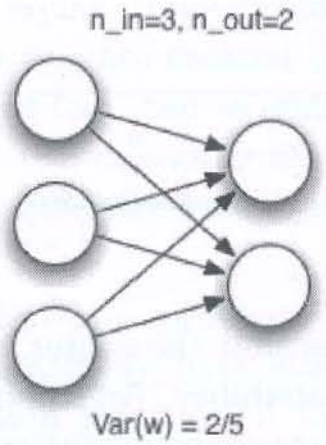
The Xavier weight initialization sets all of the weights to normally distributed random numbers. These weights are always centered at 0; however,

their standard deviation varies depending on how many connections are present for the current layer of weights. Specifically, Equation 4.2 can determine the standard deviation:

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{4.2}$$

The above equation shows how to obtain the variance for all of the weights. The square root of the variance is the standard deviation. Most random number generators accept a standard deviation rather than a variance. As a result, you usually need to take the square root of the above equation. Figure 4.4 shows how one layer might be initialized:

Figure 4.4: Xavier Initialization of a Layer



This process is completed for each layer in the neural network.

4.4 Radial-Basis Function Networks

Radial-basis function (RBF) networks are a type of feedforward neural network introduced by Broomhead and Lowe (1988). These networks can be used for both classification and regression. Though they can solve a variety of problems, RBF networks seem to losing popularity. By their very definition, RBF networks cannot be used in conjunction with deep learning.

The RBF network utilizes a parameter vector, a model that specifies weights and coefficients, in order to allow the input to generate the correct output. By adjusting a random parameter vector, the RBF network produces output consistent with the iris data set. The process of adjusting the parameter vector to produce the desired output is called training. Many different methods exist for training an RBF network. The parameter vectors also represent its long-term memory.

In the next section, we will briefly review RBFs and describe the exact makeup of these vectors.

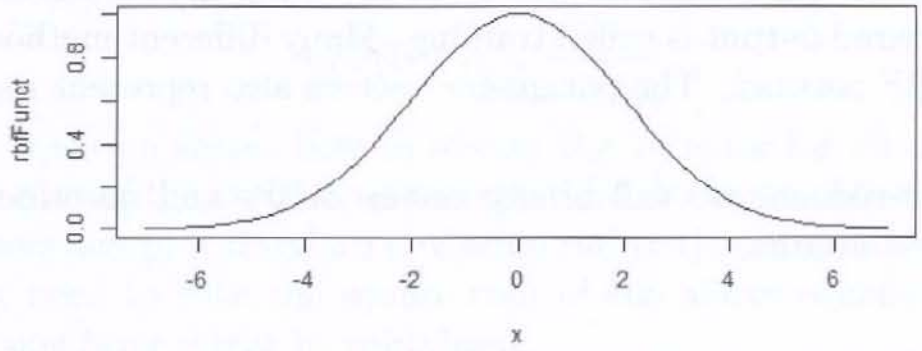
4.4.1 Radial-Basis Functions

Because many AI algorithms utilize radial-basis functions, they are a very important concept to understand. A radial-basis function is symmetric with respect to its center, which is usually somewhere along the x-axis. The RBF will reach its maximum value or peak at the center. Whereas a typical setting for the peak in RBF networks is 1, the center varies accordingly.

RBFs can have many dimensions. Regardless of the number of dimensions in the vector passed to the RBF, its output will always be a single scalar value.

RBFs are quite common in AI. We will start with the most prevalent, the Gaussian function. Figure 4.5 shows a graph of a 1D Gaussian function centered at 0:

Figure 4.5: Gaussian Function



You might recognize the above curve as a normal distribution or a bell curve, which is a radial-basis function. The RBFs, such as a Gaussian function, can selectively scale numeric values. Consider Figure 4.5 above. If you applied this function to scale numeric values, the result would have maximum intensity at the center. As you moved from the center, the intensity would diminish in either the positive or negative directions.

Before we can look at the equation for the Gaussian RBF, we must consider how to process the multiple dimensions. RBFs accept multi-dimensional input and return a single value by calculating the distance between the input and the center vector. This distance is called r. The RBF center and input to the RBF must always have the same number of dimensions for the calculation to occur. Once we calculate r, we can determine the individual RBF. All of the RBFs use this calculated r.

Equation 4.3 shows how to calculate r:

$$r = ||\mathbf{x} - \mathbf{x}_i|| \tag{4.3}$$

The double vertical bars that you see in the above equation signify that the function describes a distance or a norm. In certain cases, these distances can vary; however, RBFs typically utilize Euclidean distance. As a result, the examples that we provide in this book always apply the Euclidean distance. Therefore, r is simply the Euclidean distance between the center and the x vector. In each of the RBFs in this section, we will use this value r. Equation

4.4 shows the equation for a Gaussian RBF:

$$\phi(r) = e^{-r^2} \tag{4.4}$$

Once you've calculated r, determining the RBF is fairly easy. The Greek letter ϕ, which you see at the left of the equation, always represents the RBF. The constant e in Equation 4.4 represents Euler's number, or the natural base, and is approximately 2.71828.

4.4.2 Radial-Basis Function Networks

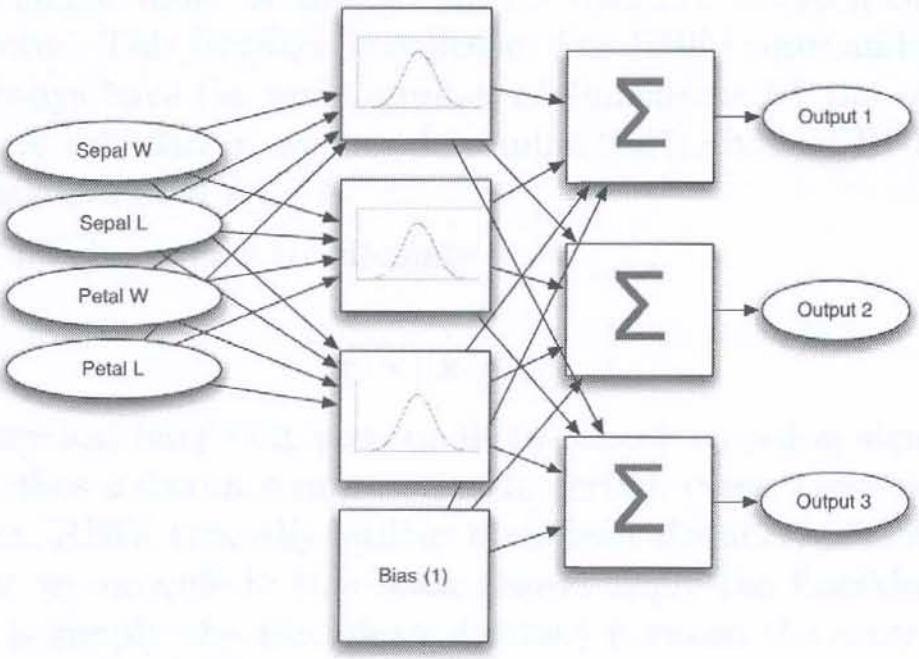
RBF networks provide a weighted summation of one or more radial-basis functions; each of these functions receives the weighted input attributes in order to predict the output. Consider the RBF network as a long equation that contains the parameter vector. Equation 4.5 shows the equation needed to calculate the output of this network:

$$f(X) = \sum_{i=1}^{N} a_i p(||b_i X - c_i||) \tag{4.5}$$

Note that the double vertical bars in the above equation signify that you must calculate the distance. Because these symbols do not specify which distance algorithm to use, you can select the algorithm. In the above equation, X is the input vector of attributes; c is the vector center of the RBF; p is the chosen RBF (Gaussian, for example); a is the vector coefficient (or weight) for each RBF; and b specifies the vector coefficient to weight the input attributes.

In our example, we will apply an RBF network to the iris data set. Figure 4.6 provides a graphic representation of this application:

Figure 4.6: The RBF Network for the Iris Data



The above network contains four inputs (the length and width of petals and sepals) that indicate the features that describe each iris species. The above diagram assumes that we are using one-of-n encoding for the three different iris species. Using equilateral encoding for only two outputs is also possible. To keep things simple, we will use one-of-n and arbitrarily choose three RBFs. Even though additional RBFs allow the model to learn more complex data sets, they require more time to process.

Arrows represent all coefficients from the equation. In Equation 4.5, b represents the arrows between the input attributes and the RBFs. Similarly, a represents the arrows between the RBFs and the summation. Notice also the bias box, which is a synthetic function that always returns a value of 1. Because the bias function's output is constant, the program does not require inputs. The weights from the bias to the summation specify the y-intercept for the equation. In short, bias is not always bad. This case demonstrates that bias is an important component to the RBF network. Bias nodes are also very common in neural networks.

Because multiple summations exist, you can see the development of a classification problem. The highest summation specifies the predicted class. A regression problem indicates that the model will output a single numeric value.

You will also notice that Figure 4.4 contains a bias node in the place of an additional RBF. Unlike the RBF, the bias node does not accept any input. It always outputs a constant value of 1. Of course, this constant value of 1 is multiplied by a coefficient value, which always causes the coefficient to be directly added to the output, regardless of the input. When the input is 0, bias nodes are very useful because they allow the RBF layer to output values despite the low value of the input.

The long-term memory vector for the RBF network has several different components:

- Input coefficients
- Output/Summation coefficients
- RBF width scalars (same width in all dimensions)
- RBF center vectors

The RBF network will store all of these components as a single vector that will become its long-term memory. Then an optimization algorithm can set the vector to values that will produce the correct iris species for the features presented. This book contains several optimization algorithms that can train an RBF network.

In conclusion, this introduction provided a basic overview of vectors, distance, and RBF networks. Since this discussion included only the prerequisite material to understand Volume 3, refer to Volumes 1 and 2 for a more thorough explanation of these topics.

4.5 Normalizing Data

Normalization was briefly mentioned previously in this book. In this section, we will see exactly how it is performed. Data are not usually presented to the neural network in exactly the same raw form as you found it. Usually data are scaled to a specific range in a process called normalization. There are many different ways to normalize data. For a full summary, refer to *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms*. This chapter will present a few normalization methods most useful for neural networks.

4.5.1 One-of-N Encoding

If you have a categorical value, such as the species of an iris, the make of an automobile, or the digit label in the MNIST data set, you should use one-of-n

encoding. This type of encoding is sometimes referred to as one-hot encoding. To encode in this way, you would use one output neuron for each class in the problem. Recall the MNSIT data set from the book's introduction, where you have images for digits between 0 and 9. This problem is most commonly encoded as ten output neurons with a softmax activation function that gives the probability of the input being one of these digits. Using one-of-n encoding, the ten digits might be encoded as follows:

```
0 -> [1,0,0,0,0,0,0,0,0,0]
1 -> [0,1,0,0,0,0,0,0,0,0]
2 -> [0,0,1,0,0,0,0,0,0,0]
3 -> [0,0,0,1,0,0,0,0,0,0]
4 -> [0,0,0,0,1,0,0,0,0,0]
5 -> [0,0,0,0,0,1,0,0,0,0]
6 -> [0,0,0,0,0,0,1,0,0,0]
7 -> [0,0,0,0,0,0,0,1,0,0]
8 -> [0,0,0,0,0,0,0,0,1,0]
0 -> [0,0,0,0,0,0,0,0,0,1]
```

One-of-n encoding should always be used when the classes have no ordering. Another example of this type of encoding is the make of an automobile. Usually the list of automakers is unordered unless there is some meaning you wish to convey by this ordering. For example, you might order the automakers by the number of years in business. However, this classification should only be done if the number of years in business has meaning to your problem. If there is truly no order, then one-of-n should always be used.

Because you can easily order the digits, you might wonder why we use one-of-n encoding for them. However, the order of the digits does not mean the program can recognize them. The fact that "1" and "2" are numerically next to each other does nothing to help the program recognize the image. Therefore, we should not use a single-output neuron that simply outputs the digit recognized. The digits 0-9 are categories, not actual numeric values. Encoding categories with a single numeric value is detrimental to the neural network's decisions process.

Both the input and output can use one-of-n encoding. The above listing used 0's and 1's. Normally you will use the rectified linear unit (ReLU) and softmax activation, and this type of encoding is normal. However, if you are working with a hyperbolic tangent activation function, you should utilize a value of -1 for the 0's to match the hyperbolic tangent's range of -1 to 1.

If you have an extremely large number of classes, one-of-n encoding can become cumbersome because you must have a neuron for every class. In such cases, you have several options. First, you might find a way to order your categories. With this ordering, your categories can now be encoded as a numeric value, which would be the current category's position within the ordered list.

Another approach to dealing with an extremely large number of categories is frequency-inverse document frequency (TF-IDF) encoding because each class essentially becomes the probability of that class's occurrence relative to the others. In this way, TF-IDF allows the program to map a large number of classes to a single neuron. A complete discussion of TF-IDF is beyond the scope of this book; however, it is built into many machine learning frameworks for languages such as R, Python, and some others.

4.5.2 Range Normalization

If you have a real number or an ordered list of categories, you might choose range normalization because it simply maps the input data's range into the range of your activation function. Sigmoid, ReLU and softmax use a range between 0 and 1, whereas hyperbolic tangent uses a range between -1 and 1.

You can normalize a number with Equation 4.6:

$$norm(x, d_L, d_H, n_L, n_H) = \frac{(x-d_L)(n_H-n_L)}{(d_H-d_L)} + n_L \tag{4.6}$$

To perform the normalization, you need the high and low values of the data to be normalized, given by dl and dh in the equation above. Similarly, you need the high and low values to normalize into (usually 0 and 1), given by nl and nh.

Sometimes you will need to undo the normalization performed on a number and return it to a denormalized state. Equation 4.7 performs this operation:

$$denorm(x, d_L, d_H, n_L, n_H) = \frac{(d_L - d_H)x - (n_H \cdot d_L) + d_H \cdot n_L}{(n_L - n_H)} \quad (4.7)$$

A very simple way to think of range normalization is percentages. Consider the following analogy. You see an advertisement stating that you will receive a $10 (USD) reduction on a product, and you have to decide if this deal is worthwhile. If you are buying a t-shirt, this offer is probably a good deal; however, if you are buying a car, $10 does not really matter. Furthermore, you need to be familiar with the current value of US dollars in order to make your decision. The situation changes if you learn that the merchant had offered a 10% discount. Thus, the value is now more meaningful. No matter if you are buying a t-shirt, car or even a house, the 10% discount has clear ramifications on the problem because it transcends currencies. In other words, the percentage is a type of normalization. Just like in the analogy, normalizing to a range helps the neural network evaluate all inputs with equal significance.

4.5.3 Z-Score Normalization

Z-score normalization is the most common normalization for either a real number or an ordered list. For nearly all applications, z-score normalization should be used in place of range normalization. This normalization type is based on the statistical concept of z-scores, the same technique for grading exams on a curve. Z-scores provide even more information than percentages.

Consider the following example. Student A scored 85% of the points on her exam. Student B scored 75% of the points on his exam. Which student earned the better grade? If the professor is simply reporting the percentage of correct points, then student A earned a better score. However, you might change your answer if you learned that the average (mean) score for student A's very easy exam was 95%. Similarly, you might reconsider your position if you discovered that student B's class had an average score of 65%. Student B performed above average on his exam. Even though student A earned a better score, she performed below average. To truly report a curved score (a z-score) you must have the mean score and the standard deviation. Equation 4.8 shows the calculation of a mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.8}$$

You can calculate the mean (μ, mu) by adding all of the scores and dividing by the number of scores. This process is the same as taking an average. Now that you have the average, you need the standard deviation. If you had a mean score of 50 points, then everyone taking the exam varied from the mean by some amount. The average amount that students varied from the mean is essentially the standard deviation. Equation 4.9 shows the calculation of the standard deviation (σ, sigma):

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{4.9}$$

Essentially, the process of taking a standard deviation is squaring and summing each score's difference from the mean. These values are added together and the square root is taken of this total. Now that you have the standard deviation, you can calculate the z-score with Equation 4.10:

$$z = \frac{x - \mu}{\sigma} \tag{4.10}$$

Listing 4.2 shows the pseudocode needed to calculate a z-score:

Listing 4.2: Calculate a Z-Score

```
# Data to score:
data = [ 5, 10, 3, 20, 4]
# Sum the values
sum = 0
for d in data:
    sum = sum + d
# Calculate mean
mean = float(sum) / len(data)
print( "Mean: " + mean )
# Calculate the variance
variance = 0
for d in data:
    variance = variance + ((mean-d)**2)
variance = variance / len(data)
```

```
print( "Variance: " + variance )
# Calculate the standard deviation
sdev = sqrt(variance)
print( "Standard Deviation: " + sdev )
# Calculate zscore
zscore = []
for d in data:
    zscore.append( (d-mean)/sdev)
print("Z-Scores: " + str(zscore) )
```

The above code will result in the following output:

```
Mean: 8.4
Variance: 39.440000000000005
Standard Deviation: 6.280127387243033
Z-Scores: [-0.5413902920037097, 0.2547719021193927,
    -0.8598551696529507, 1.8470962903655976, -0.7006227308283302]
```

The z-score is a numeric value where 0 represents a score that is exactly the mean. A positive z-score is above average; a negative z-score is below average. To help visualize z-scores, consider the following mapping between z-scores and letter grades:

```
<-2.0 = D+
-2.0  = C-
-1.5  = C
-1.0  = C+
-0.5  = B-
0.0   = B
+0.5  = B+
+1.0  = A-
+1.5  = A
+2.0  = A+
```

We took the mapping listed above from an undergraduate syllabus. There is a great deal of variation on z-score to letter grade mapping. Most professors will set the 0.0 z-score to either a C or a B, depending on if the professor/university considers C or B to represent an average grade. The above professor considered B to be average. The z-score works well for neural network input as it is centered at 0 and will very rarely go above +3 and below -3.

4.5.4 Complex Normalization

The input to a neural network is commonly called its feature vector. The process of creating a feature vector is critical to mapping your raw data to a form that the neural network can comprehend. The process of mapping the raw data to a feature vector is called encoding. To see this mapping at work, consider the auto MPG data set:

```
1. mpg:             numeric
2. cylinders:       numeric, 3 unique
3. displacement:    numeric
4. horsepower:      numeric
5. weight:          numeric
6. acceleration:    numeric
7. model year:      numeric, 3 unique
8. origin:          numeric, 7 unique
9. car name:        string (unique for each instance)
```

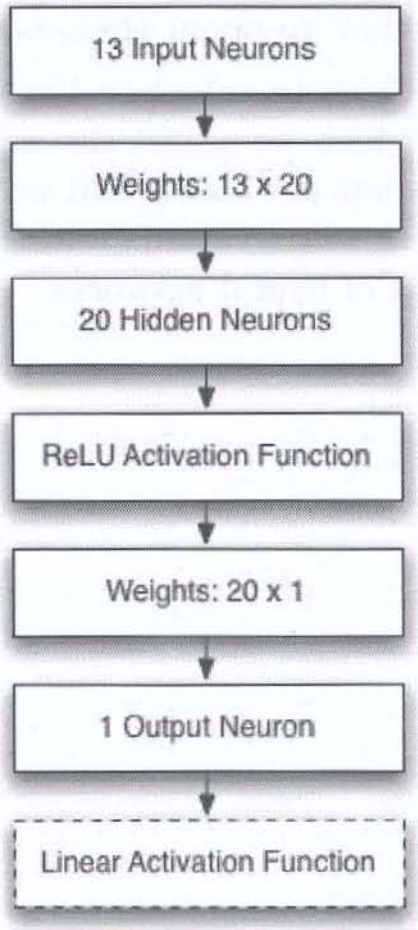
To encode the above data, we will use MPG as the output and treat the data set as regression. The MPG feature will be z-score encoded, and it falls within the range of the linear activation function that we will use on the output.

We will discard the car name. Cylinders and model-year are both one-of-n encoded, the remaining fields will be z-score encoded. The following feature vector results:

```
Input Feature Vector:
Feature 1: cylinders-2, -1 no, +1 yes
Feature 2: cylinders-4, -1 no, +1 yes
Feature 3: cylinders-8, -1 no, +1 yes
Feature 4: displacement z-score
Feature 5: horsepower z-score
Feature 6: weight z-score
Feature 7: acceleration z-score
Feature 8: model year-1977, -1 no, +1 yes
Feature 9: model year-1978, -1 no, +1 yes
Feature 10: model year-1979, -1 no, +1 yes
Feature 11: origin-1
Feature 12: origin-2
Feature 13: origin-3
Output:
mpg z-score
```

As you can see, the feature vector has grown from the nine raw fields to thirteen features plus an output. A neural network for these data would have thirteen input neurons and a single output. Assuming a single-hidden layer of twenty neurons with the ReLU activation, this network would look like Figure 4.7:

Figure 4.7: Simple Regression Neural Network



4.6 Chapter Summary

Feedforward neural networks are one of the most common algorithms in artificial intelligence. In this chapter, we introduced the multilayer feedforward neural network and the radial-basis function (RBF) neural network. Classification and regression apply both of these types of neural network.

Feedforward networks have well-defined layers. The input layer accepts the input from the computer program. The output layer returns the processing result of the neural network to the calling program. Between these layers are hidden neurons that help the neural network to recognize a pattern presented at the input layer and produce the correct result on the output layer.

RBF neural networks use a series of radial-basis functions for their hidden layer. In addition to the weights, it is also possible to change the widths and centers of these RBFs. Though an RBF and feedforward network can approximate any function, they go about the process in different ways.

So far, we've seen only how to calculate the values for neural networks. Training is the process by which we adjust the weights of neural networks so that the neural network outputs the values that we desire. To train neural networks, we also need to have a way to evaluate it. The next chapter introduces both training and validation of neural networks.