Chapter 1

Neural Network Basics

- Neurons and Layers
- Neuron Types
- Activation Functions
- Logic Gates

This book is about neural networks and how to train, query, structure, and interpret them. We present many neural network architectures as well as the plethora of algorithms that can train these neural networks. Training is the process in which a neural network is adapted to make predictions from data. In this chapter, we will introduce the basic concepts that are most relevant to the neural network types featured in the book.

Deep learning, a relatively new set of training techniques for multilayered neural networks, is also a primary topic. It encompasses several algorithms that can train complex types of neural networks. With the development of deep learning, we now have effective methods to train neural networks with many layers.

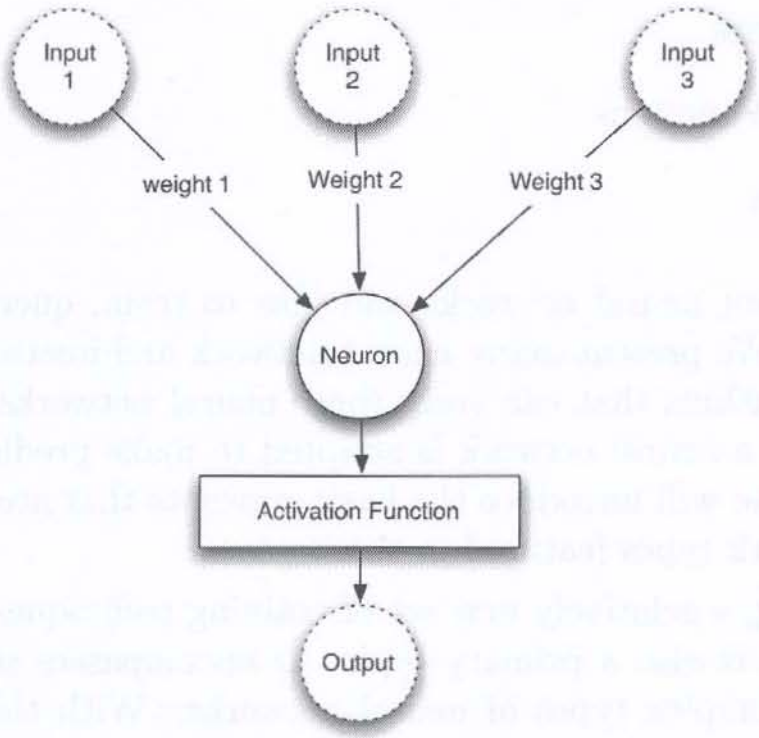
This chapter will include a discussion of the commonalities among the different neural networks. Additionally, you will learn how neurons form weighted connections, how these neurons create layers, and how activation functions affect the output of a layer. We begin with neurons and layers.

1.1 Neurons and Layers

Most neural network structures use some type of neuron. Many different kinds of neural networks exist, and programmers introduce experimental neural network structures all the time. Consequently, it is not possible to cover every neural network architecture. However, there are some commonalities among neural network implementations. An algorithm that is called a neural network will typically be composed of individual, interconnected units even though these units may or may not be called neurons. In fact, the name for a neural network processing unit varies among the literature sources. It could be called a node, neuron, or unit.

Figure 1.1 shows the abstract structure of a single artificial neuron:

Figure 1.1: An Artificial Neuron



The artificial neuron receives input from one or more sources that may be other neurons or data fed into the network from a computer program. This input is usually floating-point or binary. Often binary input is encoded to floating-point by representing true or false as 1 or 0. Sometimes the program

also depicts the binary input as using a bipolar system with true as 1 and false as -1.

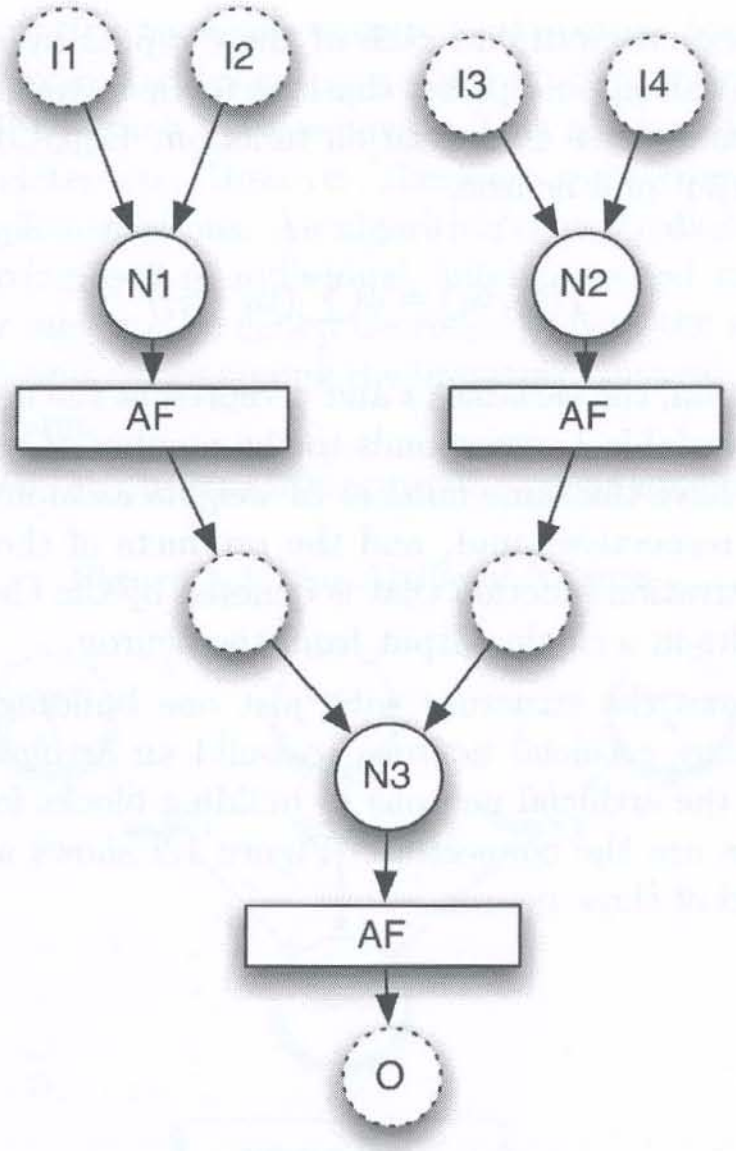
An artificial neuron multiplies each of these inputs by a weight. Then it adds these multiplications and passes this sum to an activation function. Some neural networks do not use an activation function. Equation 1.1 summarizes the calculated output of a neuron:

$$f(x_i, w_i) = \phi(\sum_i (w_i \cdot x_i)) \tag{1.1}$$

In the above equation, the variables x and w represent the input and weights of the neuron. The variable i corresponds to the number of weights and inputs. You must always have the same number of weights as inputs. Each weight is multiplied by its respective input, and the products of these multiplications are fed into an activation function that is denoted by the Greek letter ϕ (phi). This process results in a single output from the neuron.

Figure 1.1 shows the structure with just one building block. You can chain together many artificial neurons to build an artificial neural network (ANN). Think of the artificial neurons as building blocks for which the input and output circles are the connectors. Figure 1.2 shows an artificial neural network composed of three neurons:

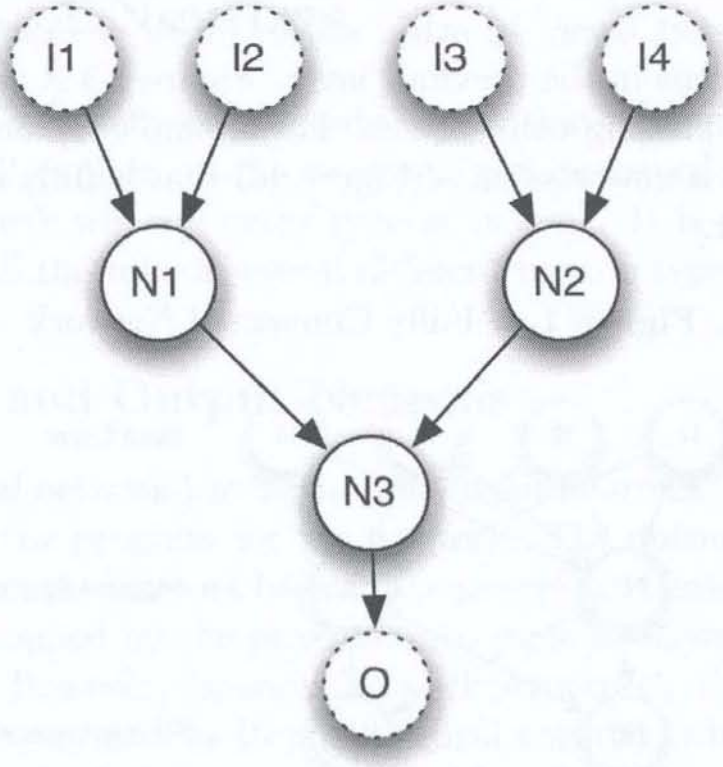
Figure 1.2: Simple Artificial Neural Network (ANN)



The above diagram shows three interconnected neurons. This representation is essentially Figure 1.1, minus a few inputs, repeated three times and then connected. It also has a total of four inputs and a single output. The output of neurons N1 and N2 feed N3 to produce the output O. To calculate the output for Figure 1.2, we perform Equation 1.1 three times. The first two times calculate N1 and N2, and the third calculation uses the output of N1 and N2 to calculate N3.

Neural network diagrams do not typically show the level of detail seen in Figure 1.2. To simplify the diagram, we can omit the activation functions and intermediate outputs, and this process results in Figure 1.3:

Figure 1.3: Simplified View of ANN



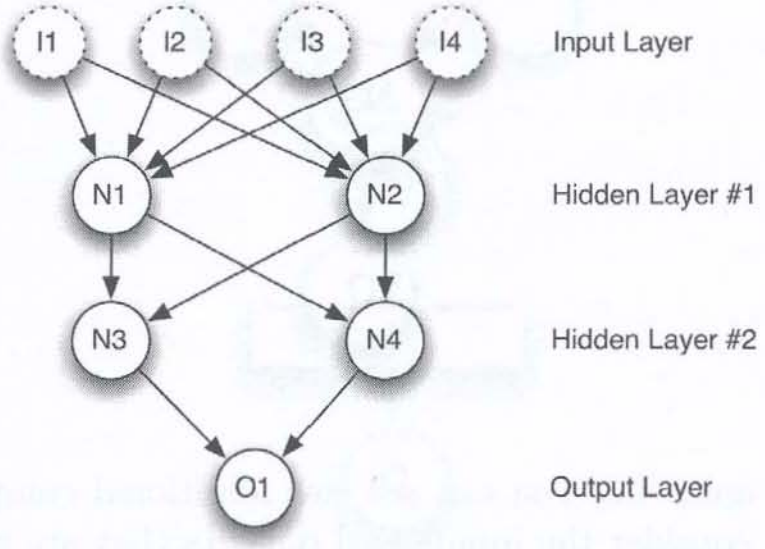
Looking at Figure 1.3, you can see two additional components of neural networks. First, consider the inputs and outputs that are shown as abstract dotted line circles. The input and output could be parts of a larger neural network. However, the input and output are often a special type of neuron that accepts data from the computer program using the neural network, and the output neurons return a result back to the program. This type of neuron is called an input neuron. We will discuss these neurons in the next section.

Figure 1.3 also shows the neurons arranged in layers. The input neurons are the first layer, the N1 and N2 neurons create the second layer, the third layer contains N3, and the fourth layer has O. While most neural networks arrange neurons into layers, this is not always the case. Stanley (2002) introduced

a neural network architecture called Neuroevolution of Augmenting Topologies (NEAT). NEAT neural networks can have a very jumbled, non-layered architecture.

The neurons that form a layer share several characteristics. First, every neuron in a layer has the same activation function. However, the layers themselves might have different activation functions. Second, layers are fully connected to the next layer. In other words, every neuron in one layer has a connection to neurons in the previous layer. Figure 1.3 is not fully connected. Several layers are missing connections. For example, I1 and N2 do not connect. Figure 1.4 is a new version of Figure 1.3 that is fully connected and has an additional layer.

Figure 1.4: Fully Connected Network



In Figure 1.4, you see a fully connected, multilayered neural network. Networks, such as this one, will always have an input and output layer. The number of hidden layers determines the name of the network architecture. The network in Figure 1.4 is a two-hidden-layer network. Most networks will have between zero and two hidden layers. Unless you have implemented deep learning strategies, networks with more than two hidden layers are rare.

You might also notice that the arrows always point downward or forward from the input to the output. This type of neural network is called a feedforward neural network. Later in this book, we will see recurrent neural networks that form inverted loops among the neurons.

1.2 Types of Neurons

In the last section, we briefly introduced the idea that different types of neurons exist. Now we will explain all the neuron types described in the book. Not every neural network will use every type of neuron. It is also possible for a single neuron to fill the role of several different neuron types.

1.2.1 Input and Output Neurons

Nearly every neural network has input and output neurons. The input neurons accept data from the program for the network. The output neuron provides processed data from the network back to the program. These input and output neurons will be grouped by the program into separate layers called the input and output layer. However, for some network structures, the neurons can act as both input and output. The Hopfield neural network, which we will discuss in Chapter 3, "Hopfield & Boltzmann Machines," is an example of a neural network architecture in which neurons are both input and output.

The program normally represents the input to a neural network as an array or vector. The number of elements contained in the vector must be equal to the number of input neurons. For example, a neural network with three input neurons might accept the following input vector:

```
[0.5, 0.75, 0.2]
```

Neural networks typically accept floating-point vectors as their input. Likewise, neural networks will output a vector with length equal to the number of output neurons. The output will often be a single value from a single output neuron. To be consistent, we will represent the output of a single output neuron network as a single-element vector.

Notice that input neurons do not have activation functions. As demonstrated by Figure 1.1, input neurons are little more than placeholders. The input is simply weighted and summed. Furthermore, the size of the input and output vectors for the neural network will be the same if the neural network has neurons that are both input and output.

1.2.2 Hidden Neurons

Hidden neurons have two important characteristics. First, hidden neurons only receive input from other neurons, such as input or other hidden neurons. Second, hidden neurons only output to other neurons, such as output or other hidden neurons. Hidden neurons help the neural network understand the input, and they form the output. However, they are not directly connected to the incoming data or to the eventual output. Hidden neurons are often grouped into fully connected hidden layers.

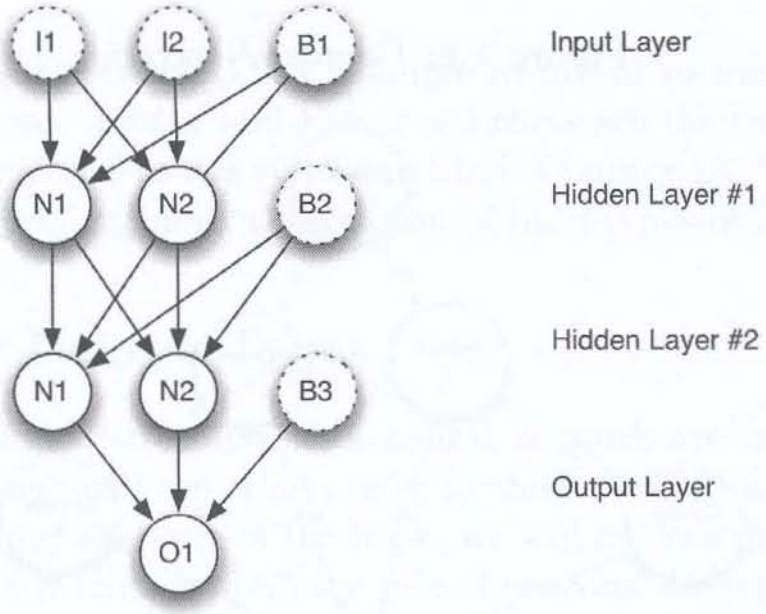
A common question for programmers concerns the number of hidden neurons in a network. Since the answer to this question is complex, more than one section of the book will include a relevant discussion of the number of hidden neurons. Prior to deep learning, it was generally suggested that anything more than a single-hidden layer is excessive (Hornik, 1991). Researchers have proven that a single-hidden-layer neural network can function as a universal approximator. In other words, this network should be able to learn to produce (or approximate) any output from any input as long as it has enough hidden neurons in a single layer.

Another reason why researchers used to scoff at the idea of additional hidden layers is that these layers would impede the training of the neural network. Training refers to the process that determines good weight values. Before researchers introduced deep learning techniques, we simply did not have an efficient way to train a deep network, which are neural networks with a large number of hidden layers. Although a single-hidden-layer neural network can theoretically learn anything, deep learning facilitates a more complex representation of patterns in the data.

1.2.3 Bias Neurons

Programmers add bias neurons to neural networks to help them learn patterns. Bias neurons function like an input neuron that always produces the value of 1. Because the bias neurons have a constant output of 1, they are not connected to the previous layer. The value of 1, which is called the bias activation, can be set to values other than 1. However, 1 is the most common bias activation. Not all neural networks have bias neurons. Figure 1.5 shows a single-hidden-layer neural network with bias neurons:

Figure 1.5: Network with Bias Neurons



The above network contains three bias neurons. Every level, except for the output layer, contains a single bias neuron. Bias neurons allow the output of an activation function to be shifted. We will see exactly how this shifting occurs later in the chapter when activation functions are discussed.

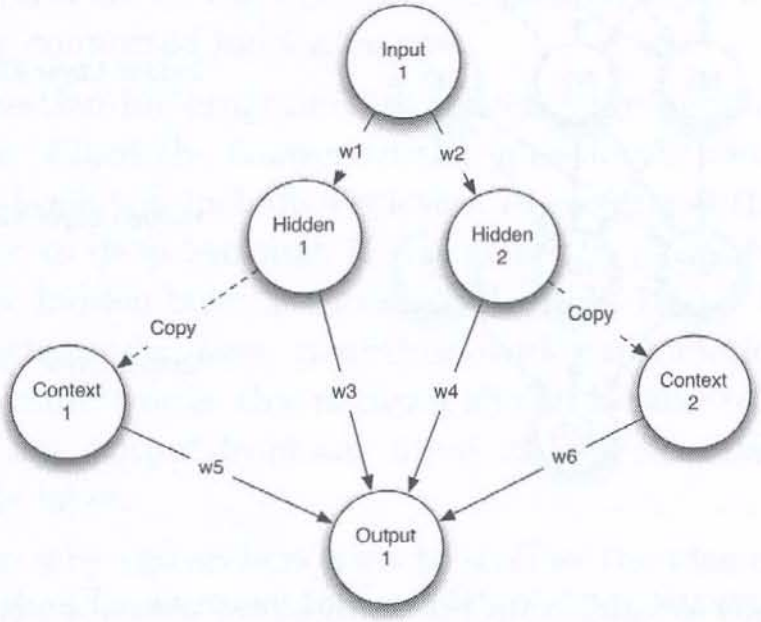
1.2.4 Context Neurons

Context neurons are used in recurrent neural networks. This type of neuron allows the neural network to maintain state. As a result, a given input may not always produce exactly the same output. This inconsistency is similar to the

workings of biological brains. Consider how context factors in your response when you hear a loud horn. If you hear the noise while you are crossing the street, you might startle, stop walking, and look in the direction of the horn. If you hear the horn while you are watching an action adventure film in a movie theatre, you don't respond in the same way. Therefore, prior inputs give you the context for processing the audio input of a horn.

Time series is one application of context neurons. You might need to train a neural network to learn input signals to perform speech recognition or to predict trends in security prices. Context neurons are one way for neural networks to deal with time series data. Figure 1.6 shows how context neurons might be arranged in a neural network:

Figure 1.6: Context Neurons



This neural network has a single input and output neuron. Between the input and output layers are two hidden neurons and two context neurons. Other than the two context neurons, this network is the same as previous networks in the chapter.

Each context neuron holds a value that starts at 0 and always receives a copy of either hidden 1 or hidden 2 from the previous use of the network. The two dashed lines in Figure 1.6 mean that the context neuron is a direct copy with no other weighting. The other lines indicate that the output is weighted by one of the six weight values listed above. Equation 1.1 still calculates the output in the same way. The value of the output neuron would be the sum of all four inputs, multiplied by their weights, and applied to the activation function.

A type of neural network called a simple recurrent neural network (SRN) uses context neurons. Jordan and Elman networks are the two most common types of SRN. Figure 1.6 shows an Elman SRN. Chapter 13, "Time Series and Recurrent Networks," includes a discussion of both types of SRN.

1.2.5 Other Neuron Types

The individual units that comprise a neural network are not always called neurons. Researchers will sometimes refer to these neurons as nodes, units or summations. In later chapters of the book, we will explore deep learning that utilizes Boltzmann machines to fill the role of neurons. Regardless of the type of unit, neural networks are almost always constructed of weighted connections between these units.

1.3 Activation Functions

In neural network programming, activation or transfer functions establish bounds for the output of neurons. Neural networks can use many different activation functions. We will discuss the most common activation functions in this section.

Choosing an activation function for your neural network is an important consideration because it can affect how you must format input data. In this

chapter, we will guide you on the selection of an activation function. Chapter 14, "Architecting Neural Networks," will also contain additional details on the selection process.

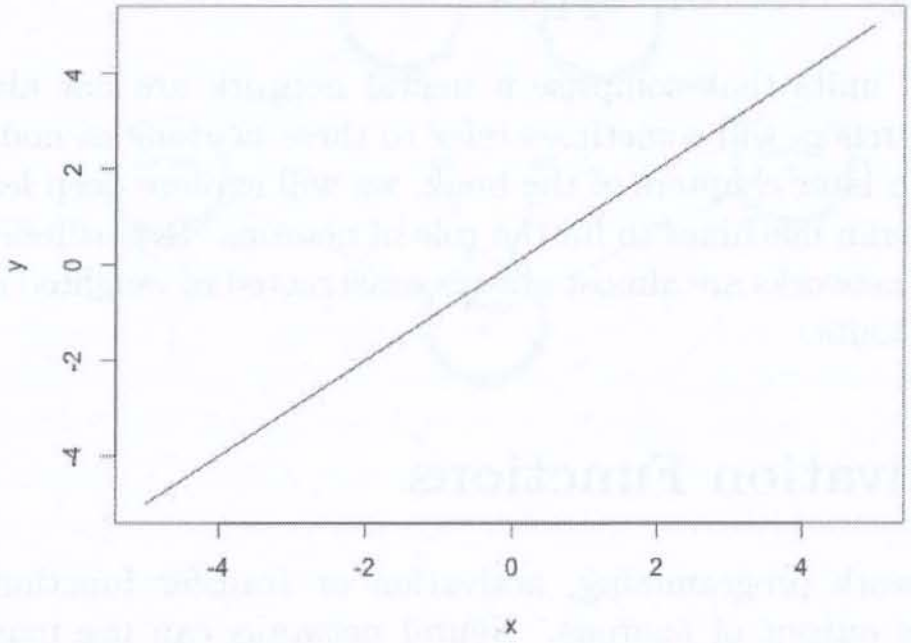
1.3.1 Linear Activation Function

The most basic activation function is the linear function because it does not change the neuron output at all. Equation 1.2 shows how the program typically implements a linear activation function:

$$\phi(x) = x \tag{1.2}$$

As you can observe, this activation function simply returns the value that the neuron inputs passed to it. Figure 1.7 shows the graph for a linear activation function:

Figure 1.7: Linear Activation Function



Regression neural networks, those that learn to provide numeric values, will usually use a linear activation function on their output layer. Classification

neural networks, those that determine an appropriate class for their input, will usually utilize a softmax activation function for their output layer.

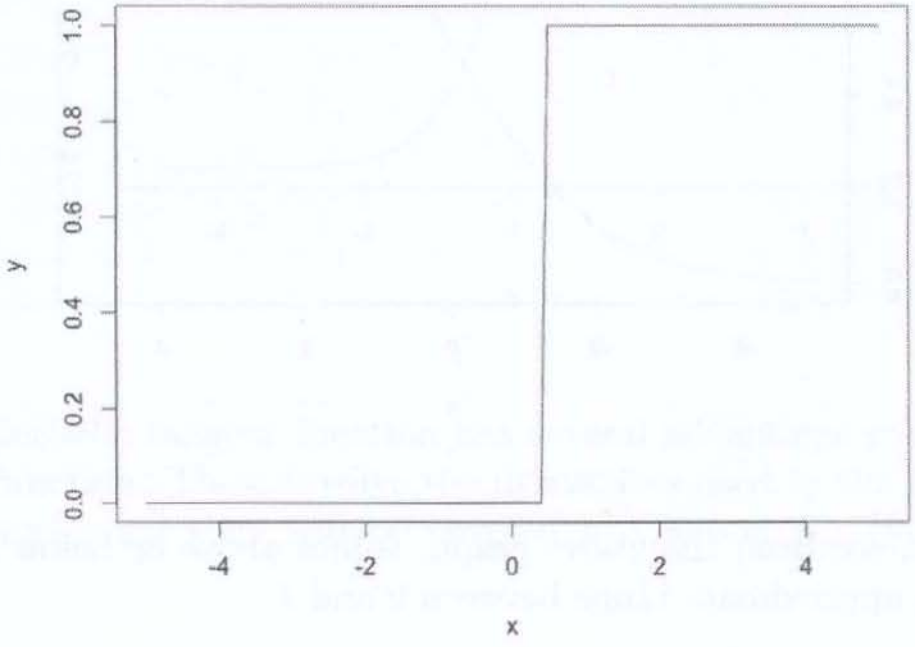
1.3.2 Step Activation Function

The step or threshold activation function is another simple activation function. Neural networks were originally called perceptrons. McCulloch & Pitts (1943) introduced the original perceptron and used a step activation function like Equation 1.3:

$$\phi(x) = \begin{cases} 1, \text{if } x \geq 0.5. \\ 0, \text{otherwise.} \end{cases} \tag{1.3}$$

Equation 1.3 outputs a value of 1.0 for incoming values of 0.5 or higher and 0 for all other values. Step functions are often called threshold functions because they only return 1 (true) for values that are above the specified threshold, as seen in Figure 1.8:

Figure 1.8: Step Activation Function



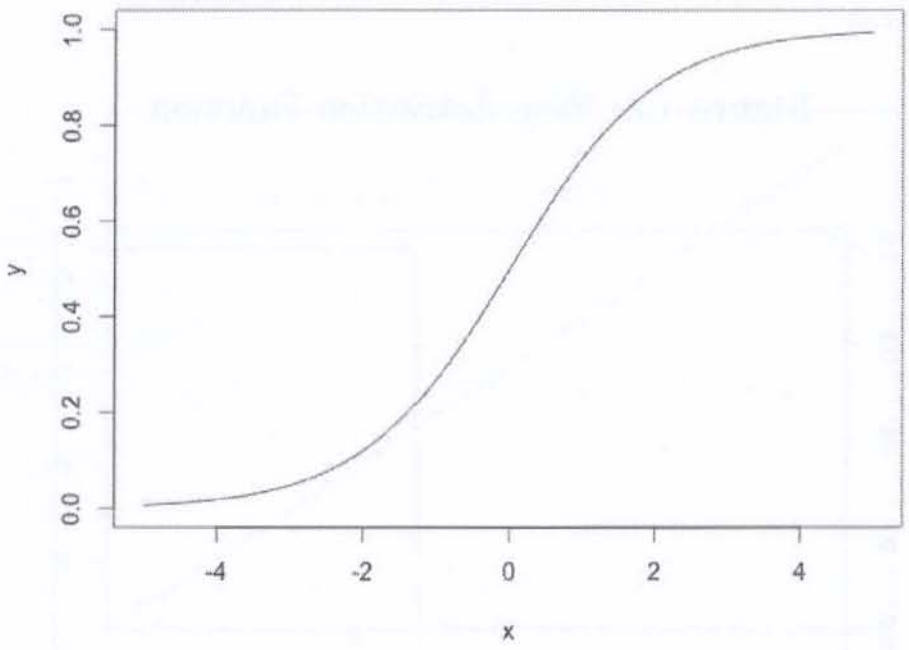
1.3.3 Sigmoid Activation Function

The sigmoid or logistic activation function is a very common choice for feed-forward neural networks that need to output only positive numbers. Despite its widespread use, the hyperbolic tangent or the rectified linear unit (ReLU) activation function is usually a more suitable choice. We introduce the ReLU activation function later in this chapter. Equation 1.4 shows the sigmoid activation function:

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{1.4}$$

Use the sigmoid function to ensure that values stay within a relatively small range, as seen in Figure 1.9:

Figure 1.9: Sigmoid Activation Function



As you can see from the above graph, values above or below 0 are compressed to the approximate range between 0 and 1.

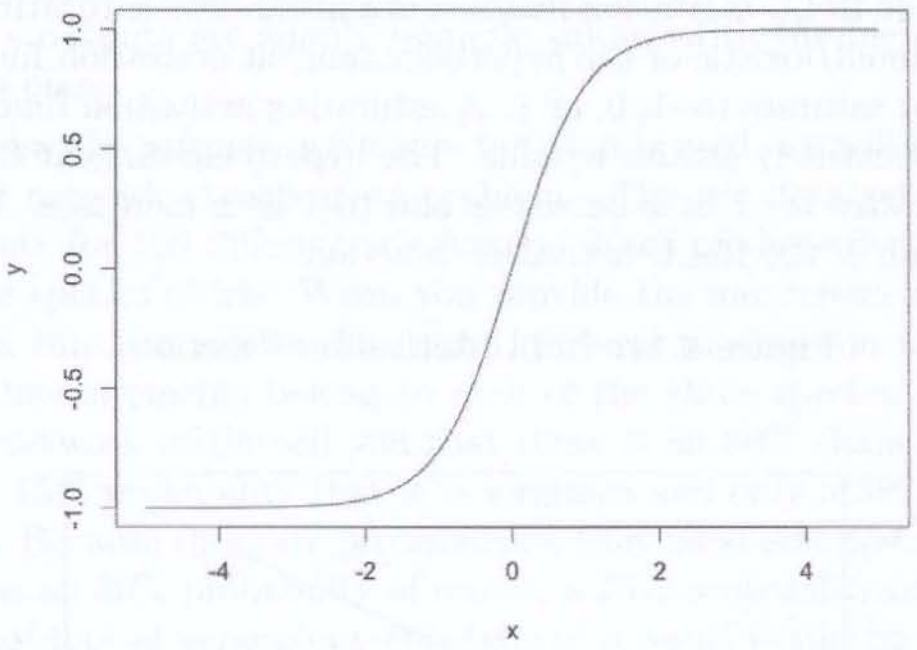
1.3.4 Hyperbolic Tangent Activation Function

The hyperbolic tangent function is also a very common activation function for neural networks that must output values in the range between -1 and 1. This activation function is simply the hyperbolic tangent (tanh) function, as shown in Equation 1.5:

$$\phi(x) = \tanh(x) \tag{1.5}$$

The graph of the hyperbolic tangent function has a similar shape to the sigmoid activation function, as seen in Figure 1.10:

Figure 1.10: Hyperbolic Tangent Activation Function



The hyperbolic tangent function has several advantages over the sigmoid activation function. These involve the derivatives used in the training of the neural network, and they will be covered in Chapter 6, "Backpropagation Training."

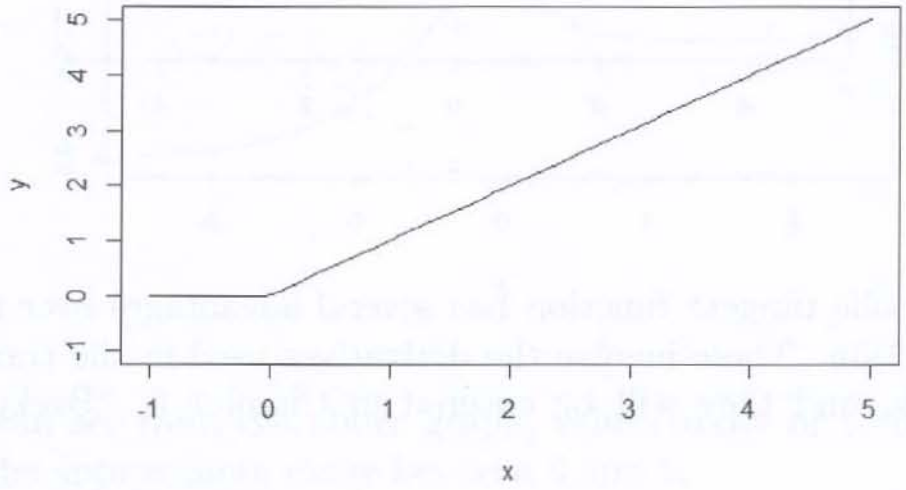
1.4 Rectified Linear Units (ReLU)

Introduced in 2000 by Teh & Hinton, the rectified linear unit (ReLU) has seen very rapid adoption over the past few years. Prior to the ReLU activation function, the hyperbolic tangent was generally accepted as the activation function of choice. Most current research now recommends the ReLU due to superior training results. As a result, most neural networks should utilize the ReLU on hidden layers and either softmax or linear on the output layer. Equation 1.6 shows the very simple ReLU function:

$$\phi(x) = \max(0, x) \tag{1.6}$$

We will now examine why ReLU typically performs better than other activation functions for hidden layers. Part of the increased performance is due to the fact that the ReLU activation function is a linear, non-saturating function. Unlike the sigmoid/logistic or the hyperbolic tangent activation functions, the ReLU does not saturate to -1, 0, or 1. A saturating activation function moves towards and eventually attains a value. The hyperbolic tangent function, for example, saturates to -1 as x decreases and to 1 as x increases. Figure 1.11 shows the graph of the ReLU activation function:

Figure 1.11: ReLU Activation Function



Most current research states that the hidden layers of your neural network should use the ReLU activation. The reasons for the superiority of the ReLU over hyperbolic tangent and sigmoid will be demonstrated in Chapter 6, "Backpropagation Training."

1.4.1 Softmax Activation Function

The final activation function that we will examine is the softmax activation function. Along with the linear activation function, softmax is usually found in the output layer of a neural network. The softmax function is used on a classification neural network. The neuron that has the highest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the output of the neural network to represent the probability that the input falls into each of the classes. Without the softmax, the neuron's outputs are simply numeric values, with the highest indicating the winning class.

To see how the softmax activation function is used, we will look at a common neural network classification problem. The iris data set contains four measurements for 150 different iris flowers. Each of these flowers belongs to one of three species of iris. When you provide the measurements of a flower, the softmax function allows the neural network to give you the probability that these measurements belong to each of the three species. For example, the neural network might tell you that there is an 80% chance that the iris is setosa, a 15% probability that it is virginica and only a 5% probability of versicolour. Because these are probabilities, they must add up to 100%. There could not be an 80% probability of setosa, a 75% probability of virginica and a 20% probability of versicolour–this type of a result would be nonsensical.

To classify input data into one of three iris species, you will need one outp neuron for each of the three species. The output neurons do not inherent specify the probability of each of the three species. Therefore, it is desirable provide probabilities that sum to 100%. The neural network will tell you t probability of a flower being each of the three species. To get the probabili use the softmax function in Equation 1.7:

$$\phi_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}} \quad (1.$$

In the above equation, i represents the index of the output neuron (o) bei calculated, and j represents the indexes of all neurons in the group/level. T variable z designates the array of output neurons. It's important to no that the softmax activation is calculated differently than the other activati functions in this chapter. When softmax is the activation function, the outp of a single neuron is dependent on the other output neurons. In Equation 1. you can observe that the output of the other output neurons is contained the variable z, as none of the other activation functions in this chapter utili z. Listing 1.1 implements softmax in pseudocode:

Listing 1.1: The Softmax Function

```
def softmax(neuron_output):
  sum = 0
  for v in neuron_output:
    sum = sum + v

  sum = math.exp(sum)
  proba = []
  for i in range(len(neuron_output))
proba[i] = math.exp(neuron_output[i])/sum
  return proba
```

To see the softmax function in operation, refer to the following URL:

http://www.heatonresearch.com/aifh/vol3/softmax.html

Consider a trained neural network that classifies data into three categori such as the three iris species. In this case, you would use one output neur

for each of the target classes. Consider if the neural network were to output the following:

```
Neuron 1: setosa: 0.9
Neuron 2: versicolour: 0.2
Neuron 3: virginica: 0.4
```

From the above output, we can clearly see that the neural network considers the data to represent a setosa iris. However, these numbers are not probabilities. The 0.9 value does not represent a 90% likelihood of the data representing a setosa. These values sum to 1.5. In order for them to be treated as probabilities, they must sum to 1.0. The output vector for this neural network is the following:

```
[0.9,0.2,0.4]
```

If this vector is provided to the softmax function, the following vector is returned:

```
[0.47548495534876745 , 0.2361188410001125 , 0.28839620365112]
```

The above three values do sum to 1.0 and can be treated as probabilities. The likelihood of the data representing a setosa iris is 48% because the first value in the vector rounds to 0.48 (48%). You can calculate this value in the following manner:

```
sum=exp(0.9)+exp(0.2)+exp(0.4)=5.17283056695839
j0= exp(0.9)/sum = 0.47548495534876745
j1= exp(0.2)/sum = 0.2361188410001125
j2= exp(0.4)/sum = 0.28839620365112
```

1.4.2 What Role does Bias Play?

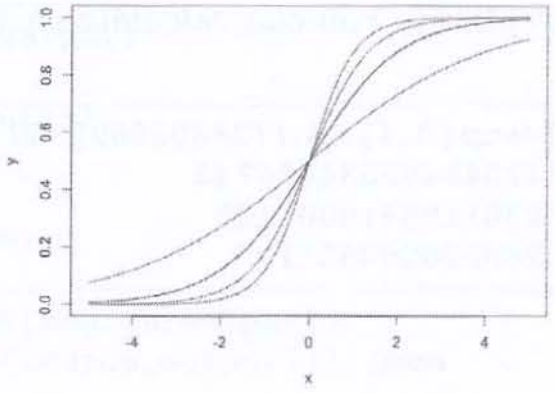
The activation functions seen in the previous section specify the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider Equation 1.8. It represents a single-input sigmoid activation neural network.

$$f(x, w, b) = \frac{1}{1 + e^{-(wx+b)}} \tag{1.8}$$

The x variable represents the single input to the neural network. The w and b variables specify the weight and bias of the neural network. The above equation is a combination of the Equation 1.1 that specifies a neural network and Equation 1.4 that designates the sigmoid activation function.

The weights of the neuron allow you to adjust the slope or shape of the activation function. Figure 1.12 shows the effect on the output of the sigmoid activation function if the weight is varied:

Figure 1.12: Adjusting Neuron Weight



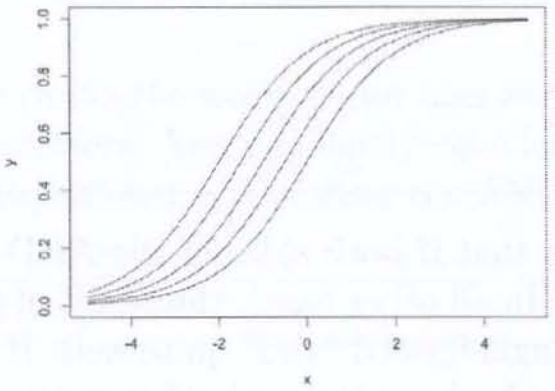
The above diagram shows several sigmoid curves using the following parameters:

```
f(x,0.5,0.0)
f(x,1.0,0.0)
f(x,1.5,0.0)
f(x,2.0,0.0)
```

To produce the curves, we did not use bias, which is evident in the third parameter of 0 in each case. Using four weight values yields four different sigmoid curves in Figure 1.11. No matter the weight, we always get the same value of 0.5 when x is 0 because all of the curves hit the same point when x is 0. We might need the neural network to produce other values when the input is near 0.5.

Bias does shift the sigmoid curve, which allows values other than 0.5 when x is near 0. Figure 1.13 shows the effect of using a weight of 1.0 with several different biases:

Figure 1.13: Adjusting Neuron Bias



The above diagram shows several sigmoid curves with the following parameters:

```
f(x,1.0,1.0)
f(x,1.0,0.5)
f(x,1.0,1.5)
f(x,1.0,2.0)
```

We used a weight of 1.0 for these curves in all cases. When we utilized several different biases, sigmoid curves shifted to the left or right. Because all the curves merge together at the top right or bottom left, it is not a complete shift.

When we put bias and weights together, they produced a curve that created the necessary output from a neuron. The above curves are the output from only one neuron. In a complete network, the output from many different neurons will combine to produce complex output patterns.

1.5 Logic with Neural Networks

As a computer programmer, you are probably familiar with logical programming. You can use the programming operators AND, OR, and NOT to govern how a program makes decisions. These logical operators often define the actual meaning of the weights and biases in a neural network. Consider the following truth table:

```
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
NOT 0 = 1
NOT 1 = 0
```

The truth table specifies that if both sides of the AND operator are true, the final output is also true. In all other cases, the result of the AND is false. This definition fits with the English word "and" quite well. If you want a house with a nice view AND a large backyard, then both requirements must be fulfilled for you to choose a house. If you want a house that has a nice view or a large backyard, then only one needs to be present.

These logical statements can become more complex. Consider if you want a house that has a nice view and a large backyard. However, you would also be satisfied with a house that has a small backyard yet is near a park. You can express this idea in the following way:

```
([nice view] AND [large yard]) OR ((NOT [large yard]) and [park])
```

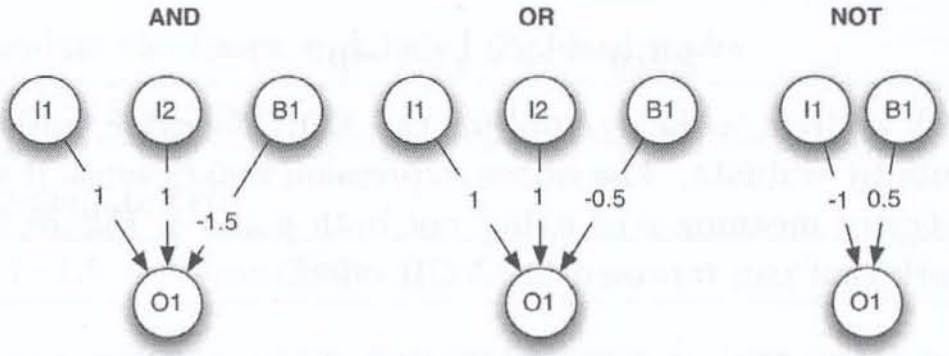
You can express the previous statement with the following logical operators:

$$(NV \wedge LY) \vee (\neg LY \wedge PK) \tag{1.9}$$

In the above statement, the OR looks like a letter "v," the AND looks like an upside down "v," and the NOT looks like half of a box.

We can use neural networks to represent the basic logical operators of AND, OR, and NOT, as seen in Figure 1.14:

Figure 1.14: Basic Logic Operators



The above diagram shows the weights and bias weight for each of the three fundamental logical operators. You can easily calculate the output for any of these operators using Equation 1.1. Consider the AND operator with two true (1) inputs:

$$(1*1) + (1*1) + (-1.5) = 0.5$$

We are using a step activation function. Because 0.5 is greater than or equal to 0.5, the output is 1 or true. We can evaluate the expression where the first input is false:

$$(1*1) + (0*1) + (-1.5) = -0.5$$

Because of the step activation function, this output is 0 or false.

We can build more complex logical structures from these neurons. Consider the exclusive or (XOR) operator that has the following truth table:

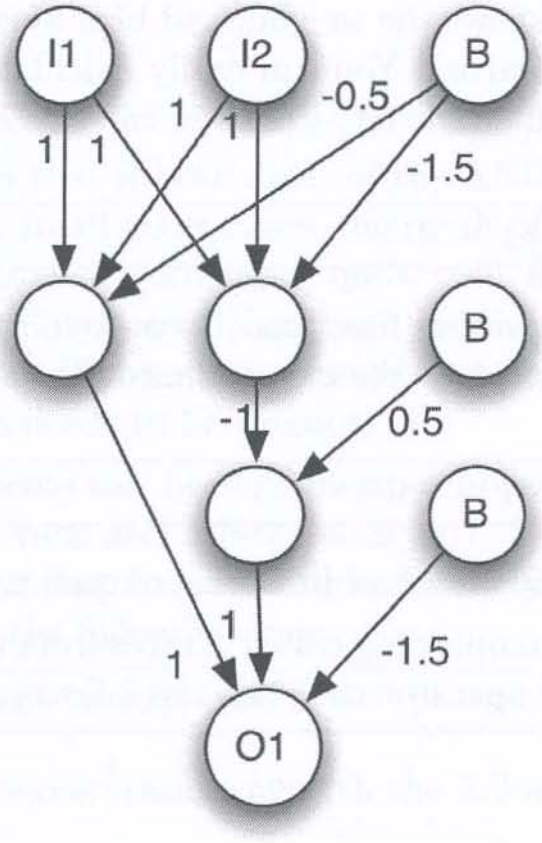
$$0 \text{ XOR } 0 = 0$$
$$1 \text{ XOR } 0 = 1$$
$$0 \text{ XOR } 1 = 1$$
$$1 \text{ XOR } 1 = 0$$

The XOR operator specifies that one, but not both, of the inputs can be true. For example, one of the two cars will win the race, but not both of them will win. The XOR operator can be written with the basic AND, OR, and NOT operators as follows:

$$p \oplus q = (p \vee q) \wedge \neg(p \wedge q) \tag{1.10}$$

The plus with a circle is the symbol for the XOR operator, and p and q are the two inputs to evaluate. The above expression makes sense if you think of the XOR operator meaning p or q, but not both p and q. Figure 1.15 shows a neural network that can represent an XOR operator:

Figure 1.15: XOR Neural Network



Calculating the above neural network would require several steps. First, you must calculate the values for every node that is directly connected to the inputs. In the case of the above neural network, there are two nodes. We will

show an example of calculating the XOR with the inputs [0,1]. We begin by calculating the two topmost, unlabeled (hidden) nodes:

```
(0*1) + (1*1) - 0.5 = 0.5 = True
(0*1) + (1*1) - 1.5 = -0.5 = False
```

Next we calculate the lower, unlabeled (hidden) node:

```
(0*-1)+0.5 = 0.5 = True
```

Finally, we calculate O1:

```
(1*1)+(1*1)-1.5 = 0.5 = True
```

As you can see, you can manually wire the connections in a neural network to produce the desired output. However, manually creating neural networks is very tedious. The rest of the book will include several algorithms that can automatically determine the weight and bias values.

1.6 Chapter Summary

In this chapter, we showed that a neural network is comprised of neurons, layers, and activation functions. Fundamentally, the neurons in a neural network might be input, hidden, or output in nature. Input and output neurons pass information into and out of the neural network. Hidden neurons occur between the input and output neurons and help process information.

Activation functions scale the output of a neuron. We also introduced several activation functions. The two most common activation functions are the sigmoid and hyperbolic tangent. The sigmoid function is appropriate for networks in which only positive output is needed. The hyperbolic tangent function supports both positive and negative output.

A neural network can build logical statements, and we demonstrated the weights to generate AND, OR, and NOT operators. Using these three basic operators, you can build more complex, logical expressions. We presented an example of building an XOR operator.

Now that we've seen the basic structure of a neural network, we will explore in the next two chapters several classic neural networks so that you can use

this abstract structure. Classic neural network structures include the self-organizing map, the Hopfield neural network, and the Boltzmann machine. These classical neural networks form the foundation of other architectures that we present in the book.