Chapter 2

Structure of a Neural Network

2.1 The Artificial Neuron

The biological neuron is simulated in an ANN by an *activation function* (AF), or switch. If the input is above a user defined threshold, the AF switches state, e.g. from 0 to 1, -1 to 1 or from 0 to > 0. A commonly used activation function is the sigmoid function,

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

Graphically, the sigmoid function is shown in Figure 2.1.

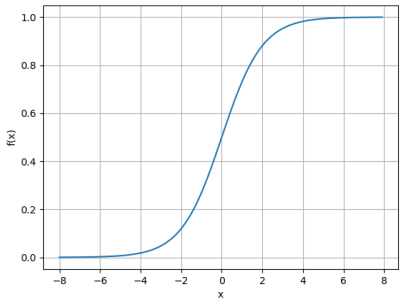

Figure 2.1: The artificial neuron.

The sigmoid function can be written and plotted numerically using Python:

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Listing 2.1: The artificial neuron.

In Machine Learning (ML) literature we say that the function is "activated", i.e., it moves from 0 to 1, when the input is greater than some threshold. It is desirable to have a smooth transition between states, as we will see in the following sections. Additionally it is required to have an algebraically smooth function – a function with a derivative at every point – which is required to train the algorithm, as discussed in Section **??**.

2.2 Nodes

Analogous to biological neurons, ANN's have a similar structure with the output of one neuron connected to the input of another neuron. We represent these networks as connected layers of *nodes*, where each node has many "weighted" inputs from previous nodes. The output of each node is the result of an AF applied to each input, followed by the sum of the AF's, multiplied by the weight of that node. For three inputs plus a weight, the graphical representation of a node is shown in Figure 2.2.

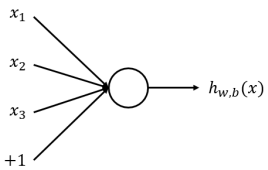

Figure 2.2: The Node Diagram.

The circle in Figure 2.2 represents the node, with inputs x_1, x_2, x_3, weight of $+1$, and output $h_{w,b}(x)$. In some literature this diagram is called a *perceptron*. Mathematically, Figure 2.2 is

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b,$$

where x_i are the inputs $[x_1, x_2, x_3]$, w_i are weights $[w_1, w_2, w_3]$, and b is the bias.

The weights are scalar values applied to each input, and while randomly initialized, are "optimized" during the learning process so that the structure of all nodes produce the desired "learned" result. The bias increases the flexibility of the node during the learning process.

2.3 Bias

Consider a simple node with only one input and one output (see Figure 2.3). The input to the activation function of the node in this case is simply $x_1 w_1$. Changing w_1 changes the *threshold* above which the AF changes state, as illustrated in Figure 2.4. This can be programmed in Python, as shown below.

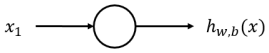

Figure 2.3: The Simple Diagram.

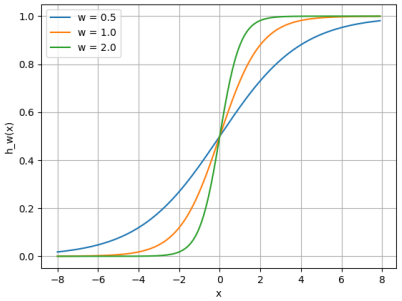

Figure 2.4: Effect of adjusting weights.

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
w1 = 0.5
w2 = 1.0
w3 = 2.0
l1 = 'w = 0.5'
l2 = 'w = 1.0'
l3 = 'w = 2.0'
```

```
for w, l in [(w1, l1), (w2, l2), (w3, l3)]:
    f = 1 / (1 + np.exp(-x*w))
plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_w(x)')
plt.legend(loc=2)
plt.show()
```

Listing 2.2: Effect of Adjusting Weights.

Notice that changing the weight, changes the slope of the output of the AF, and is useful when modeling strengths of relationships between input and output variables. However, a bias can be used if we only want the output to change when x is greater than 1. Consider the simple network of Figure 2.3, with the addition of a bias input shown in Figure 2.5.

Adjusting the bias b, translates the AF along the x-axis, resulting in a change of activation (see Figure 2.6, and the Python code below). Therefore, by adding a bias term, you can make the node simulate a generic *if* function, e.g., *if* $(x > z)$ *then* 1 *else* 0. Without a bias term, you are unable to vary z, and the function will be always near 0. This is simple example demonstrates the need for a bias to simulate conditional relationships.

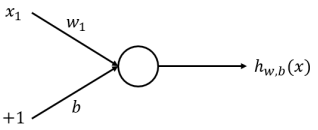

Figure 2.5: Effect of bias.

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
w = 5.0
b1 = -8.0
b2 = 0.0
b3 = 8.0
l1 = 'b = -8.0'
l2 = 'b = 0.0'
l3 = 'b = 8.0'
for b, l in [(b1, l1), (b2, l2), (b3, l3)]:
    f = 1 / (1 + np.exp(-(x*w+b)))
```

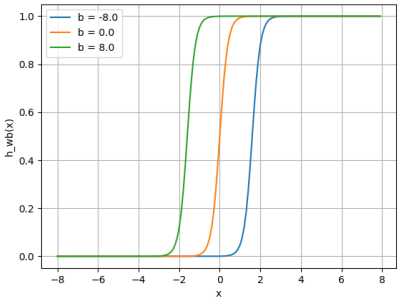

Figure 2.6: Effect of bias.

```
14 plt.plot(x, f, label=l)
15 plt.xlabel('x')
16 plt.ylabel('h_wb(x)')
17 plt.legend(loc=2)
18 plt.show()
```

Listing 2.3: Effect of Bias.

2.4 A Simple ANN Structure

Figures 2.2 -2.5 provide the structure of node-weight-bias in an ANN. While there are many forms of interconnected nodes in ANN's, a simple ANN consists of an *input layer*, a *hidden layer* and an *output layer*, as shown in Figure 2.7

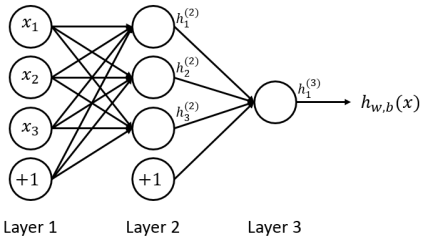

Figure 2.7: Three layer neural network.

Figure 2.7 has three *layers*:

- Layer 1 – **input layer** – where the external input data enters the network.
- Layer 2 – **hidden layer** – as this layer is not part of the input or output. Note: most neural networks can have many hidden layers, however, for simplicity we have only included one.
- Layer 3 – **output layer** – the result of the ANN.

To simplify notation, use L1 for Layer 1, N1 for node 1, etc. As shown, each node in L1 has a connection to all nodes in L2. Likewise, the nodes in L2 each connect to the single output node L3. Each of these connections have an associated weight.

2.5 Notation

The maths in the following chapters require precise notation to accurately track each step. Herein, we adopt he Stanford deep learning notation.

Weights use the notation $w_{ij}^{(l)}$, i refers to the node number of the connection in layer $l+1$, and j refers to the node number of the connection in layer l. Take special note of this order. For example, the notation for the connection between N1 in L1 and N2 in L2 is $w_{21}^{(1)}$. This notation may seem a bit odd, as you would expect the i and j to refer the node numbers in layers l and $l+1$ respectively (i.e., in the direction of input to output), rather than the opposite. However, this notation makes more sense when you consider the bias.

As shown in Figure 2.7, the (+1) bias is connected to each of the nodes in the subsequent layer. So the bias in L1 is connected to the all the nodes in L2. Because the bias is not a true node with an activation function, it has no inputs. The bias notation is $b_i^{(l)}$, where i is the node number in the layer $l+1$ – the same as used for the normal weight notation $w_{21}^{(1)}$. Continuing the previous example; the weight on the connection between the bias in L1 and N2 in L2 is $b_2^{(1)}$.

Note: the values, $w_{ij}^{(l)}$ and $b_i^{(l)}$, are computed and optimized during the training phase of the ANN development process.

The output node notation is $h_j^{(l)}$, where j denotes the node number in layer l of the network. Continuing the previous example; the output of N2 in L2 is $h_2^{(2)}$.