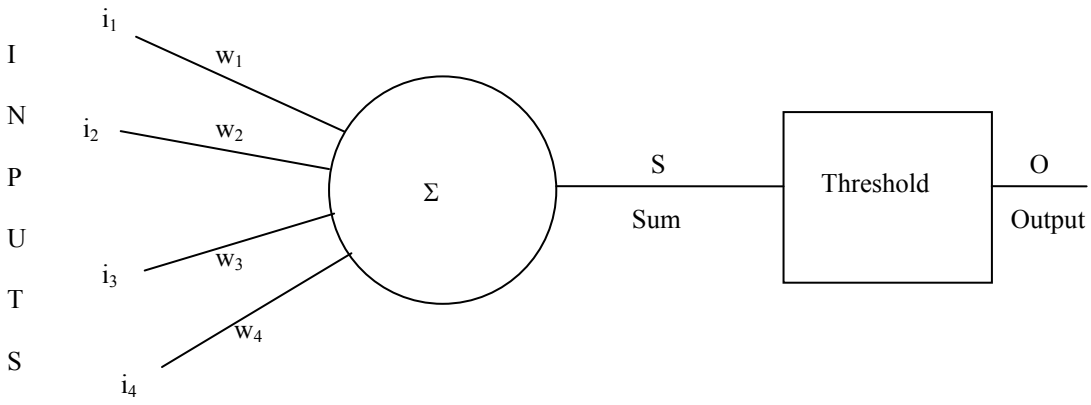
2. Artificial Neural Networks

The history of Artificial Neural Networks began in 1943 when the authors Warren McCulloch and Walter Pitts proposed a simple artificial model of the neuron[1,2]. Their model (in a slightly modified and improved form) is what most Artificial Neural Networks are based on to this day. Such simple neurons are often called *Perceptrons*. The basic neuron is shown in figure 2.1.

Figure 2.1, a basic Artificial Neuron.



2.1 The basic Artificial Neuron

The diagram shows that the inputs to the neuron are represented by *i* (in the biological sense, these are the activities of the other connecting neurons or of the outside world, transmitted through the dendrites). Each input is weighted by a factor which represents the strength of the synaptic connection of its dendrite, represented by *w*. The sum of these inputs and their weights is called the *activity* or *activation* of the neuron and is denoted *S*. In mathematical terms:

$$S = i_1 w_1 + i_2 w_2 + i_3 w_3 + i_4 w_4$$

A threshold is then applied, which is a simple binary level:

if S >0.5 then O = 1 *if S <0.5 then O = 0*	Threshold set at 0.5

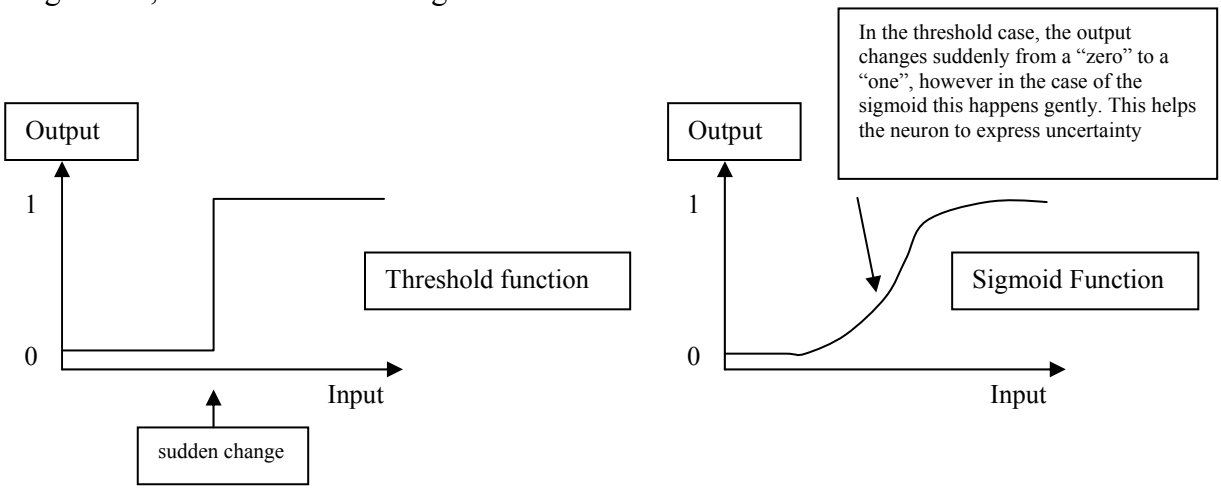
So, the neuron takes its inputs, weights them according to how strong the connection is and if the total sum of the weighted inputs is above a certain threshold, the neuron "fires", just like the biological one. As we will see later, such a neuron learns by changing its weights.

Early Artificial Neural Networks used simple binary outputs in this way. However, it was found that a continuous output function was more flexible. One example is the Sigmoid Function:

$$O = \frac{1}{1 + e^{-S}}$$

This function simply replaces the Threshold Function and produces an output which is always between zero and one, it is therefore often called a Squashing (or Activation) Function. Other Activation Functions[3] are also sometimes used, including: Linear, Logarithmic and Tangential Functions; however, the Sigmoid Function is probably the most common. Figure 2.2 shows how the Sigmoid and Threshold Functions compare with each other.

Figure 2.2, the Threshold and Sigmoid Functions.



The preceding example may be formalised for a neuron of *n* inputs:

$$S = w_1 i_1 + w_2 i_2 + \ldots\ldots + w_n i_n$$

Which may be written conveniently as:

$$S = \sum_{x=1}^{x=n} w_x i_x$$

Or, if the inputs are considered as forming a vector $\overline{I}$, and the weights a vector or matrix $\overline{W}$:

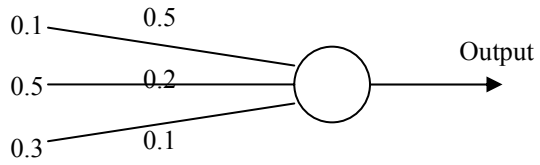
$$S = \overline{I} \,.\, \overline{W}$$

In which case the normal rules of matrix arithmetic apply. The Output Function remains a Threshold or Sigmoid. This model is not the only method of representing an Artificial Neuron; however, it is by far the most common. A Neural Network consists of a number of these neurons connected together.

The best way to clarify the neuron's operation is with actual figures, so here are couple of examples with simple neurons.

Worked example 2.1:

a. Calculate the output from the neuron below assuming a threshold of 0.5:



Sum = (0.1 x 0.5) + (0.5 x 0.2) + (0.3 x 0.1) = 0.05 + 0.1 + 0.03 = 0.18
Since 0.18 is less than the threshold, the Output = 0

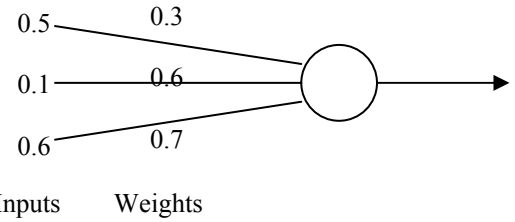
b. Repeat the above calculation assuming that the neuron has a sigmoid output function:

Sum is still 0.18, but now Output = $\frac{1}{1+e^{-0.18}} = 0.545$

Tutorial question 2.1:

Now try these yourself (answers at end of the chapter).

a. Calculate the output from the neuron below assuming a threshold of 0.5:

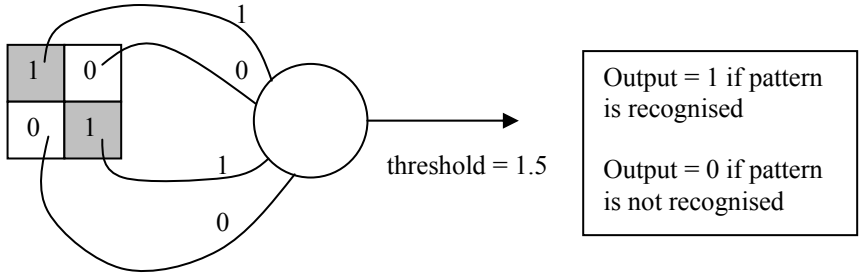


b. Repeat the above calculation assuming that the neuron has a sigmoid output function.

2.2 Recognising patterns

On a practical level, what can the neural network be used for? Well, in the late 1950s Rosenblatt[4] succeeded in making the first successful practical networks. To everyone's surprise they turned out to have some remarkable qualities, one of which was that they could be used for pattern recognition. Let's take a very simple example to see how this works, figure 2.3.

Figure 2.3, recognising a simple pattern.

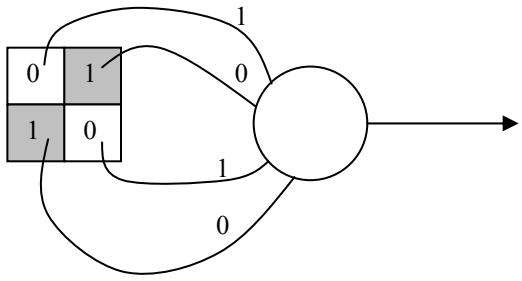


Here we have a simple picture with four pixels. Each shaded pixel is given a value 1 and each white pixel a value 0. These are connected up to the neuron, with the weights as shown. The total sum of the neuron is (1 x 1) + (0 x 0) + (1 x 1) + (0 x 0) = 2. So the neuron's output is 1 and it recognises the picture.

Even if there were some "interference" to the basic picture (some of the white pixels weren't quite 0 and some of the shaded ones weren't quite 1) the neuron would still recognise it, as long as the total sum was greater than 1.5. So the neuron is "Noise Tolerant" or able to "Generalise" which means it will still successfully recognise the picture, even if it isn't perfect - this is one of the most important attributes of Neural Networks.

Now let's put a different pattern into the same neuron as shown in figure 2.4.

Figure 2.4, a different pattern.

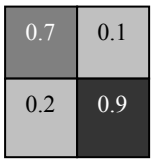


Now the sum is (0 x 1) + (1 x 0) + (0 x 1) + (1 x 0) = 0 - the neuron won't fire and doesn't recognise this picture. So, neurons can recognise simple images which their weights have been set up for. This is important because recognising images is a fundamental attribute of intelligence in animals. After all, if you want to avoid

bumping into an object, find a mate or food or avoid a predator, recognising them is a good start!

Tutorial question 2.2:

a) Show whether the network shown in figures 2.3 and 2.4 would recognise this pattern:



b) What advantage would using the sigmoid function give in this case.

2.3 Learning

Hopefully you can see the application of neural nets to pattern recognition. We'll enlarge on this shortly, but first let us turn again to the question of learning.

It was mentioned earlier, that making the network learn is done by changing its weights. Obviously, setting the weights by hand (as we've done in the example above) is fine for demonstration purposes. However, in reality, neurons have to learn to recognise not one but several patterns and the learning algorithms are not, in these cases, just common sense. The next chapter will look at this subject in detail and go through the most popular learning algorithm – Back Propagation. But just to get the idea, let's look at a very simple example of how a network could learn.

Let us say that the neuron is tackling a problem similar to that explained above and we make all the weights random numbers (this is a typical starting point for a training algorithm). Next, we apply our pattern to the network and calculate its output - of course, this output will be just a random number at the moment. To force the network output to come closer to what we want, we could adopt a simple learning algorithm like this:

- **If** the output is correct **then** do nothing.
- **If** the output is too high, but should be low **then** decrease the weights attached to high inputs
- **If** the output is too low, but should be high **then** increase the weights attached to high inputs

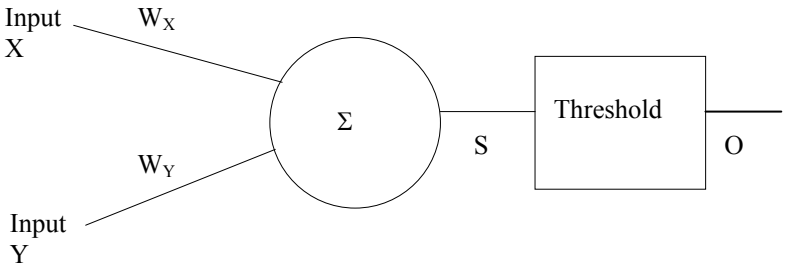
You might want to compare this with Hebb's proposal for learning mentioned in section 1.3. The artificial learning algorithms used in networks are mathematical versions of these simple ideas.

2.4 Limitation of single neurons

We can now turn to the subject of neural networks - after all, the neurons that we have been discussing are normally used in the form of networks, rather than as single units. To see the importance of this, we need to have a brief look at an important problem in Neural Nets called the Exclusive-Or (ExOR or XOR) problem.

The most famous book in the history of Neural Nets was 'Perceptrons' by Marvin Minsky and Semour Papert[5], published in 1969. It was a critique of Neural Nets and put forward some strong arguments against them which caused many researchers in the field to abandon it. One of the main arguments used in the book is that a simple Perceptron type neuron cannot simulate a two input XOR gate. This is known as the XOR problem. After all, what use was a processor so limited that it couldn't even do such a simple task? We won't go into the details of the problem here (we'll look at it in depth in chapter 6) but the idea is shown in figure 2.5.

Figure 2.5,
a simple two input neuron.

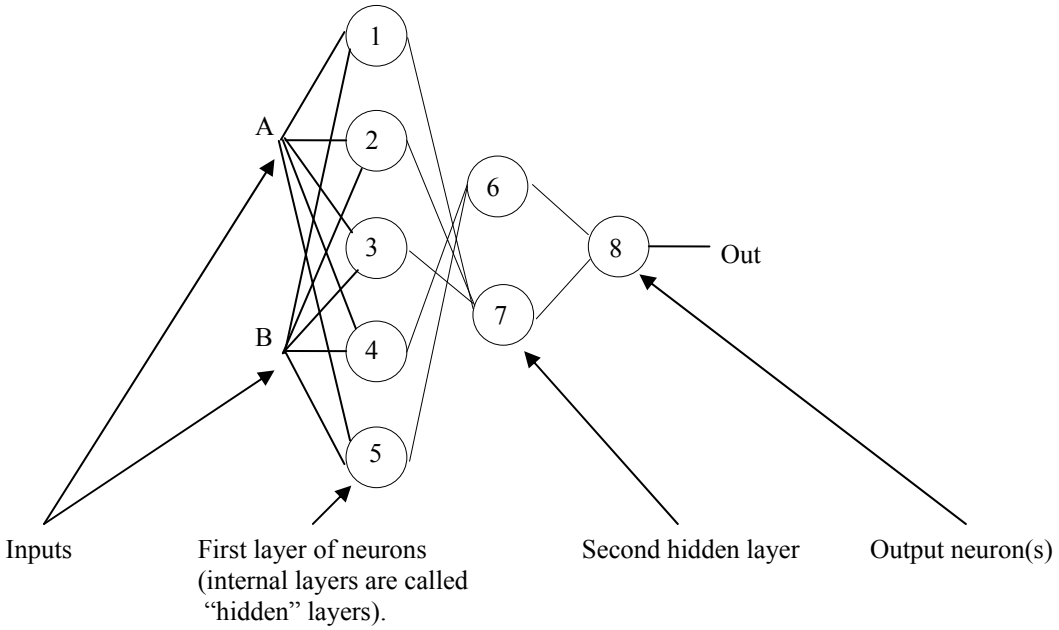


X	Y	O
0	0	0
0	1	1
1	0	1
1	1	0

No matter how you change the weights, W_x and W_Y, you can never get the neuron to give you the truth table shown

Although this attack on neural nets delayed progress for many years, it was actually quite easy to overcome - what you needed was not a single neuron, but a network of them! In fact a Russian mathematician called Kolmogorov had already proved that a three-layer network, like that shown in figure 2.6, could learn any function[7] (providing it has enough neurons in it). A detailed explanation of this can be found in reference 8 and chapter 6.

Figure 2.6, a three-layer network can learn any function.



It is networks rather than single neurons which are used today and they are capable of recognising many complex patterns.

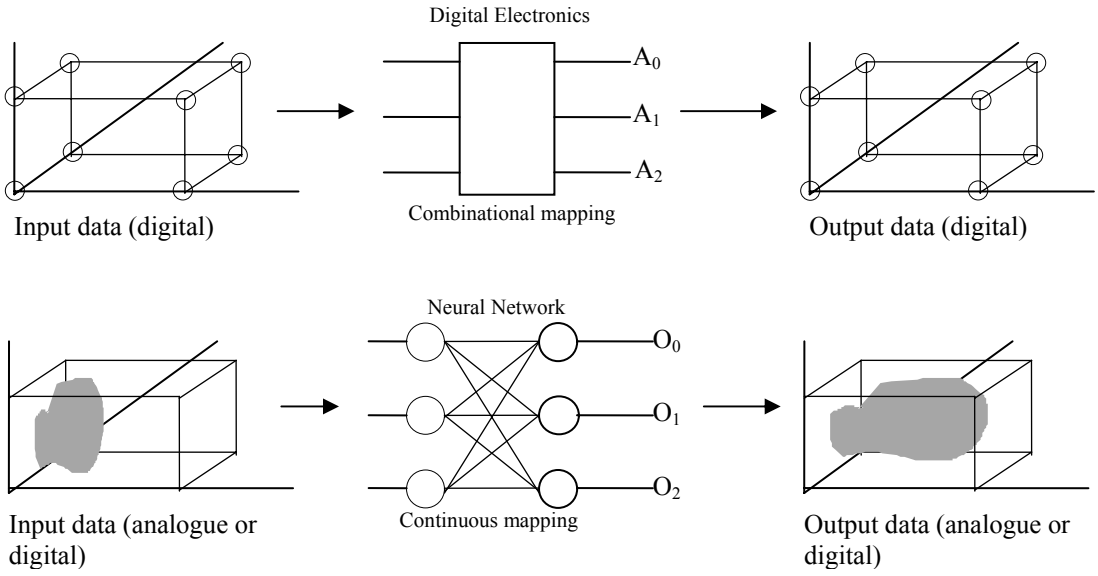
2.5 How ANNs are implemented

Artificial Neural Nets can be constructed in either electronic hardware (analogue or digital) or as software programs. The majority are software based and programming provides an easy method of implementing networks. Appendix A "Tips for Programmers" contains pointers on how to code networks in a high level language; however, before looking at these, try the programming exercises in the main text first. Chapter 10 discusses electronic and low level implementations.

2.6 ANNs and other technologies

We've looked at the Neural Net in terms of Pattern Recognition, but actually it's probably best to compare it with Digital Electronics, because it can do much more that just recognise patterns. Like other systems it does mapping - that is, it takes an input and "maps" (changes) it to a different output, just like a digital circuit takes inputs and maps them to different outputs as shown in Figure 2.7.

Figure 2.7, a neural network in its most general sense.



In this case, if the network uses a squashing function, it can produce analogue outputs rather than just 1s and 0s. It's useful to consider the differences between the two systems for a moment.

- The neural network can learn by example, detailed design is not necessary (electric circuits or fuzzy logic systems need to be carefully designed).
- A neural net degrades slowly in the presence of noise (as was discussed earlier, it will recognise a "non-perfect" pattern).
- The network is fault tolerant when implemented in hardware (if one neuron fails, the network will degrade, but still work, as the information in the network it shared among its weights).

These points are summarised in figure 2.8 below.

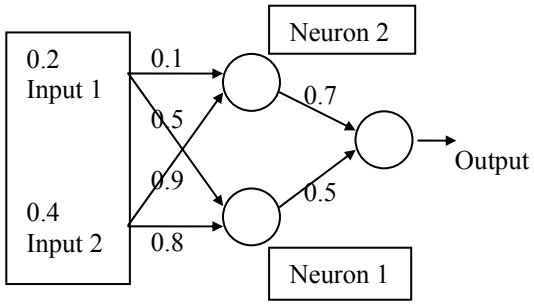
	Continuous outputs	Binary outputs	Can learn	Fixed Functionality	Parallel structure
ANNs	✓	1	✓	x	✓
Digital electronics	x	✓	x	✓	✓
Control system	✓	4	2	✓	x
Fuzzy logic	✓	x	3	✓	x

Notes:
1. The simple threshold ANNs we looked at earlier have binary outputs
2. Adaptive control systems (which can learn to a certain extent) are available
3. Adaptive Fuzzy Logic and Neuro-Fuzzy Systems can also learn
4. Some simple control systems (such as PLCs) have binary outputs

In the next chapter we'll look at some of the common ways of making neural nets learn, but let's finish of this chapter with a couple of problems to make sure that you understand the operation of the network as a whole.

Worked example 2.2:

Calculate the output from this network assuming a Sigmoid Squashing Function.



Input to neuron 1 = (0.2 x 0.5) + (0.4 x 0.8) = 0.42. Output = $\frac{1}{1+e^{-0.42}} = 0.603$

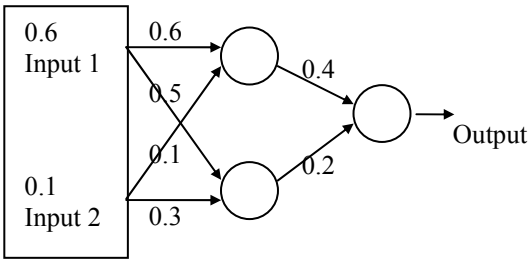
Input to neuron 2 = (0.2 x 0.1) + (0.4 x 0.9) = 0.38. Output = $\frac{1}{1+e^{-0.38}} = 0.594$

Input to final neuron = (0.594 x 0.7) + (0.603 x 0.5) = 0.717.

Final Output = $\frac{1}{1+e^{-0.717}} = 0.672$

Tutorial question 2.3:

Try calculating the output of this network yourself.



Programming exercise 2.1:

a) Using a high level language, program a computer to perform the calculations shown in worked example 2.2 and tutorial question 2.3. Make the program flexible enough so that the user can input any weights and inputs and it will calculate the output.
b) (No need to program this.) Consider how you might be able to make the program flexible enough so that the number of inputs, hidden layer neurons and outputs could be changed (don't look at the programming tips in appendix A until after you've considered your answers).

Answers to tutorial questions

2.1

a) Sum = 0.63. So output = 1

b) Output = 0.6525

2.2

a) Activation of network = 1.6. So network would recognise the pattern (the pattern is just a "noisy" version of the one in figure 2.3).

b) The sigmoid function would allow the output to "express doubt" about the pattern. In other words produce an output between 0 and 1 when the pattern wasn't "certain." In this case it produces an output of 0.83 (you might say the neuron is 83% certain that it recognises the output).

2.3

Answer = 0.587