test for sync

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.

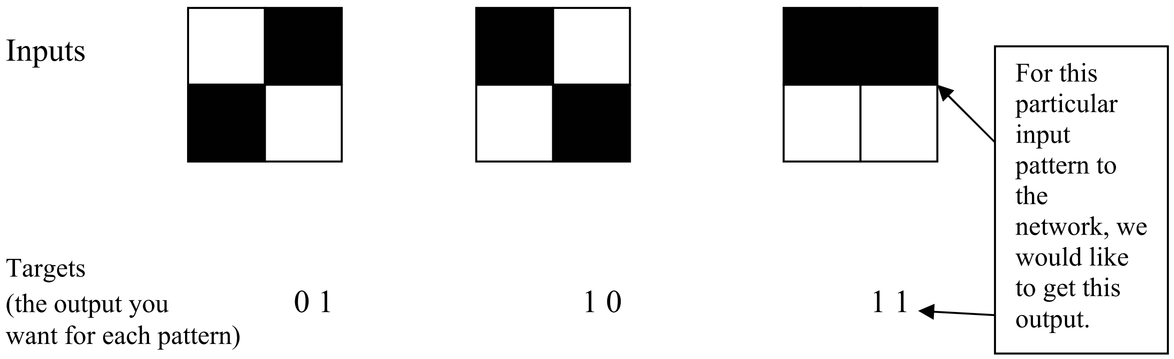
3.1 The algorithm

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.

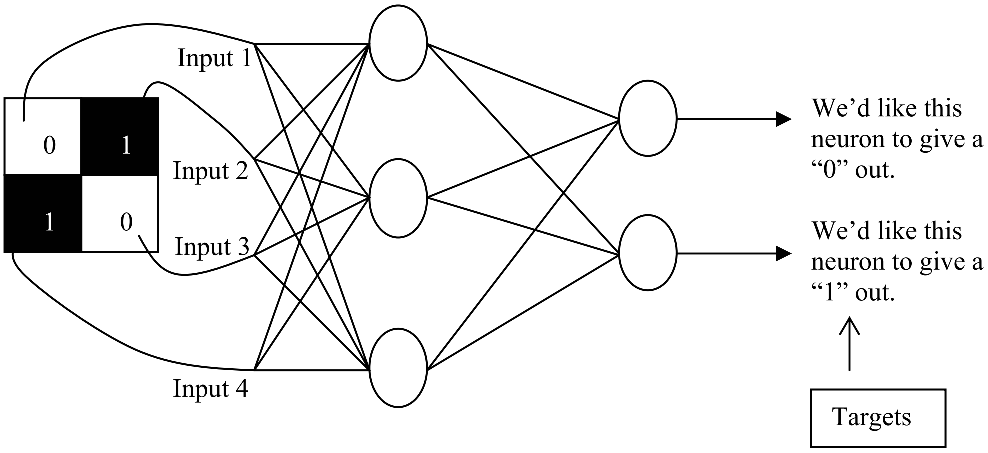
A Back Propagation network learns by example. You give the algorithm examples of what you want the network to do and it changes the network's weights so that, when training is finished, it will give you the required output for a particular input. Back Propagation networks are ideal for simple Pattern 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:

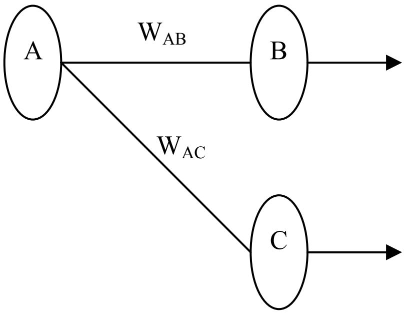
Redraw the diagram in figure 3.2 to show the inputs and targets for the second pattern.

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 output of B is float number, we wanna predict 0 or 1
① use sigmoid function, then error is equation 1-1
② just chase a threshold, then error is $(T_B - O_B)$

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:

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

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)*.

Sigmoid function: $S(t) = \frac{1}{1+e^{-t}}$ $S'(t) = S\cdot(1-S)$

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

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

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.

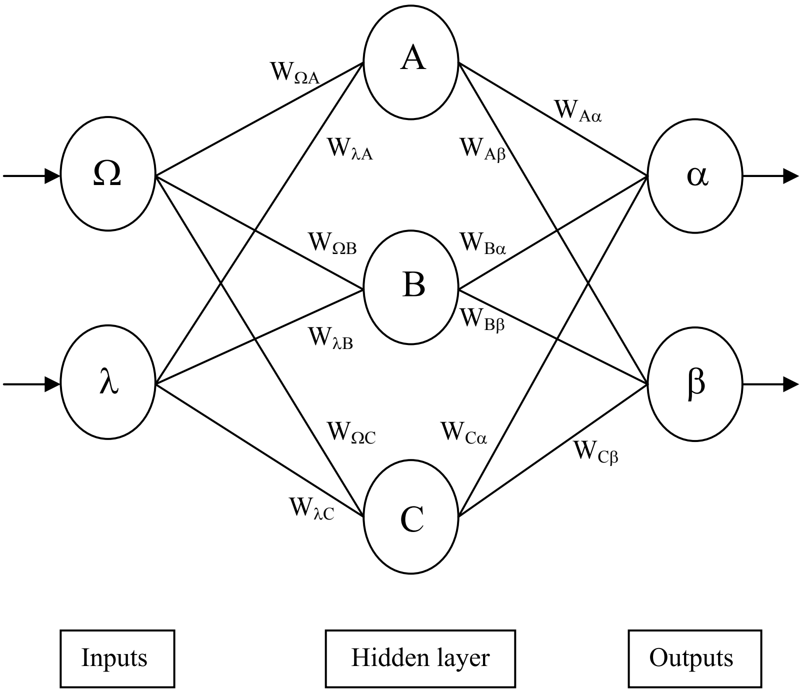
$$\text{Error}_A = \text{Output}_A\,(1 - \text{Output}_A)(\text{Error}_B\,W_{AB} + \text{Error}_C\,W_{AC})$$

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

$$\delta_\alpha = \text{out}_\alpha\ (1 - \text{out}_\alpha)\ (\text{Target}_\alpha - \text{out}_\alpha)$$
$$\delta_\beta = \text{out}_\beta\ (1 - \text{out}_\beta)\ (\text{Target}_\beta - \text{out}_\beta)$$

2. Change output layer weights

$$W^+_{A\alpha} = W_{A\alpha} + \eta\delta_\alpha\,\text{out}_A \qquad W^+_{A\beta} = W_{A\beta} + \eta\delta_\beta\,\text{out}_A$$
$$W^+_{B\alpha} = W_{B\alpha} + \eta\delta_\alpha\,\text{out}_B \qquad W^+_{B\beta} = W_{B\beta} + \eta\delta_\beta\,\text{out}_B$$
$$W^+_{C\alpha} = W_{C\alpha} + \eta\delta_\alpha\,\text{out}_C \qquad W^+_{C\beta} = W_{C\beta} + \eta\delta_\beta\,\text{out}_C$$

3. Calculate (back-propagate) hidden layer errors

$$\delta_A = \text{out}_A\ (1 - \text{out}_A)\ (\delta_\alpha W_{A\alpha} + \delta_\beta W_{A\beta})$$
$$\delta_B = \text{out}_B\ (1 - \text{out}_B)\ (\delta_\alpha W_{B\alpha} + \delta_\beta W_{B\beta})$$
$$\delta_C = \text{out}_C\ (1 - \text{out}_C)\ (\delta_\alpha W_{C\alpha} + \delta_\beta W_{C\beta})$$

4. Change hidden layer weights

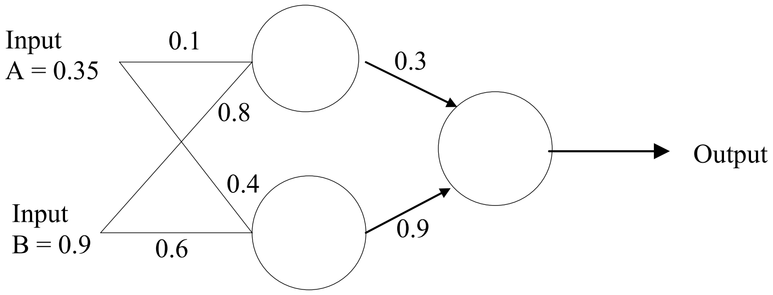
$$W^+_{\lambda A} = W_{\lambda A} + \eta\delta_A\,\text{in}_\lambda \qquad W^+_{\Omega A} = W^+_{\Omega A} + \eta\delta_A\,\text{in}_\Omega$$
$$W^+_{\lambda B} = W_{\lambda B} + \eta\delta_B\,\text{in}_\lambda \qquad W^+_{\Omega B} = W^+_{\Omega B} + \eta\delta_B\,\text{in}_\Omega$$
$$W^+_{\lambda C}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

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.

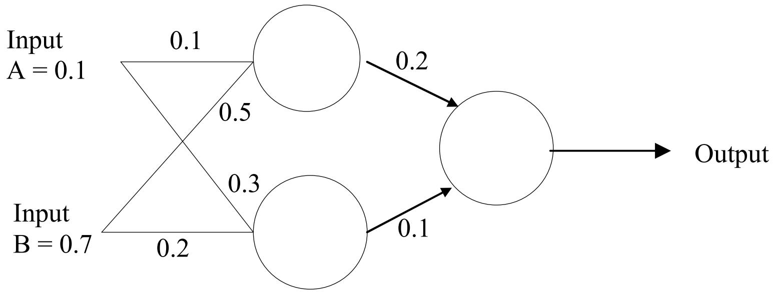
Errors for hidden layers:
δ1 = δ x w1 = -0.0406 x 0.272392 x (1-o)o = $-2.406\text{x}10^{-3}$
δ2= δ x w2 = -0.0406 x 0.87305 x (1-o)o = $-7.916\text{x}10^{-3}$

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:

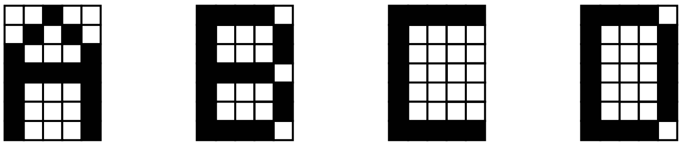


See “Tips for Programmers” in appendix A for ideas about coding networks (but try programming exercise 3.1 first).

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.

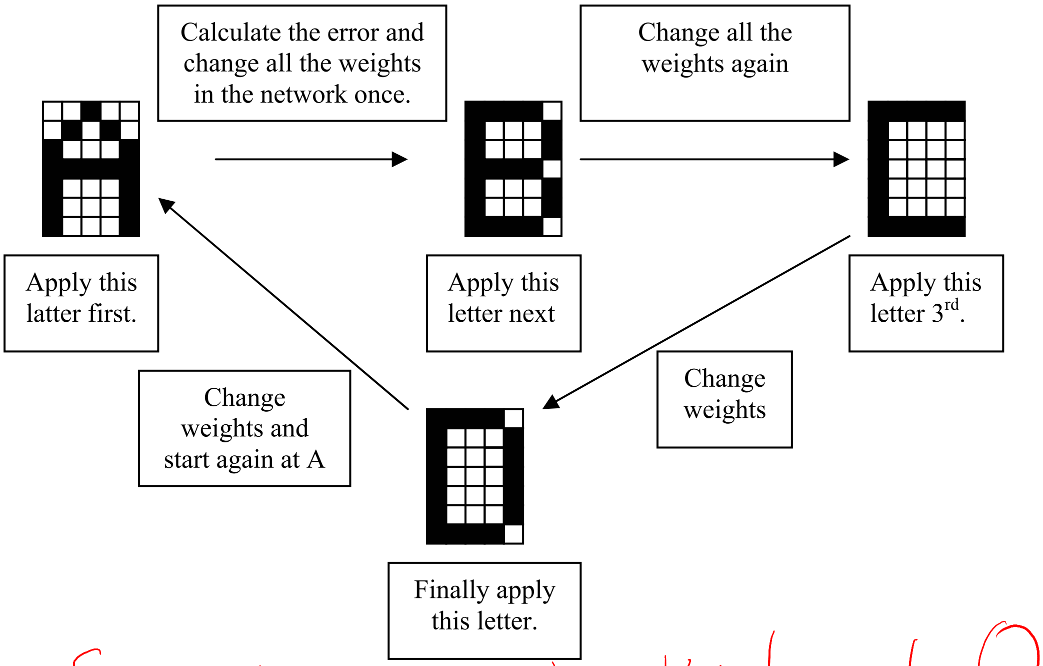
Figure 3.5, the first four letters of the alphabet.



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.

Stochastic Gradient Descend

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 and start again at A
Change weights
Finally apply this letter.

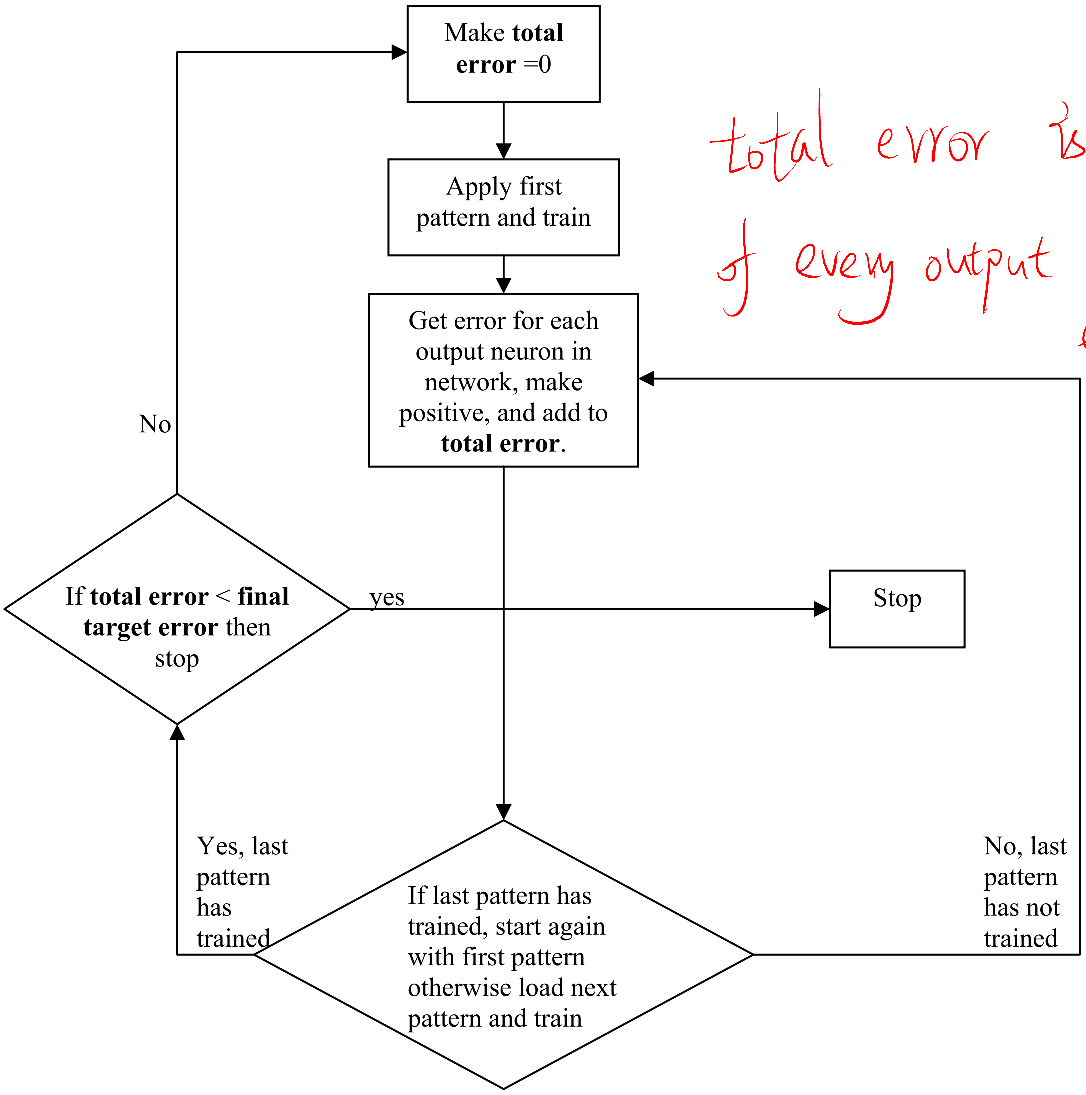
So every example is applied for only ONCE

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

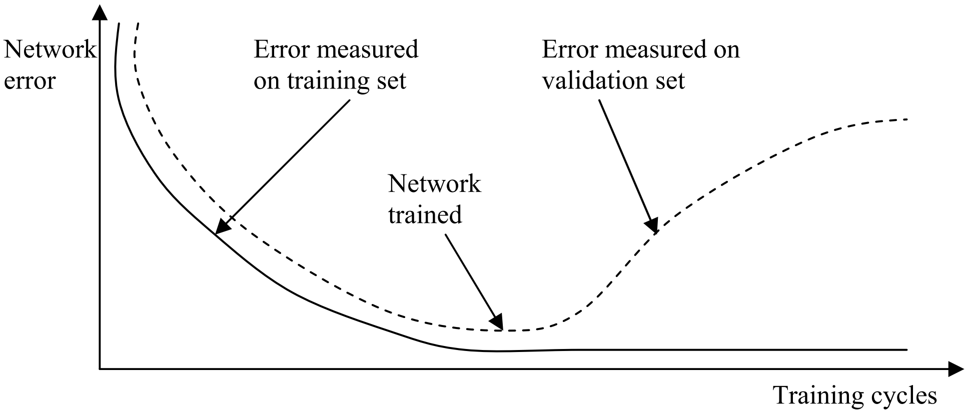
total error is the sum of every output neurons in every pattern

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.

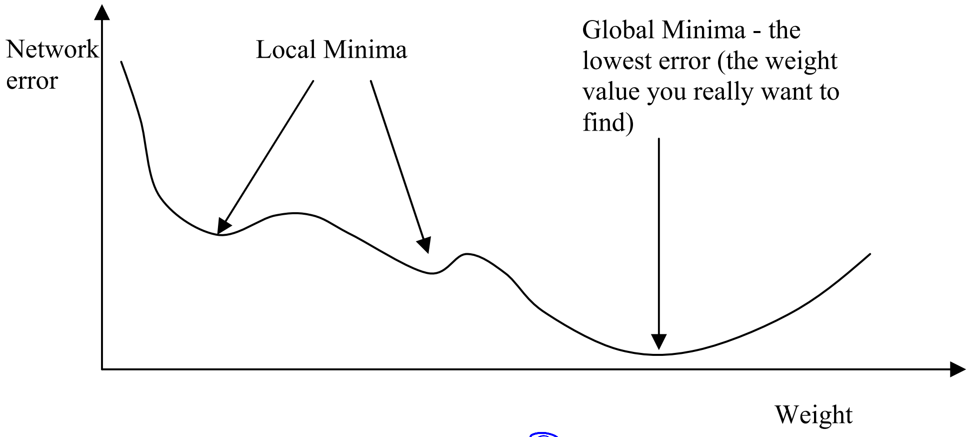


When the network has fully trained, the Validation Set error reaches a minimum. When the network is overtraining (becoming too accurate) the validation set error starts rising. If the network overtrains, it won't be able to handle noisy data so well. This topic will be explored further in section 6.4.

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. 1 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: 2

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

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].

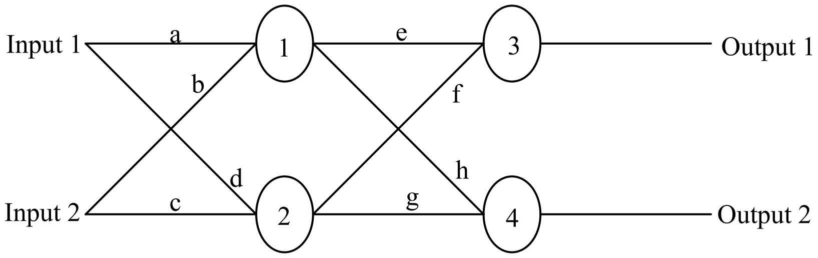
The programming exercise below will test you knowledge of what you gave learned so far and give you a simple network to play with.

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.

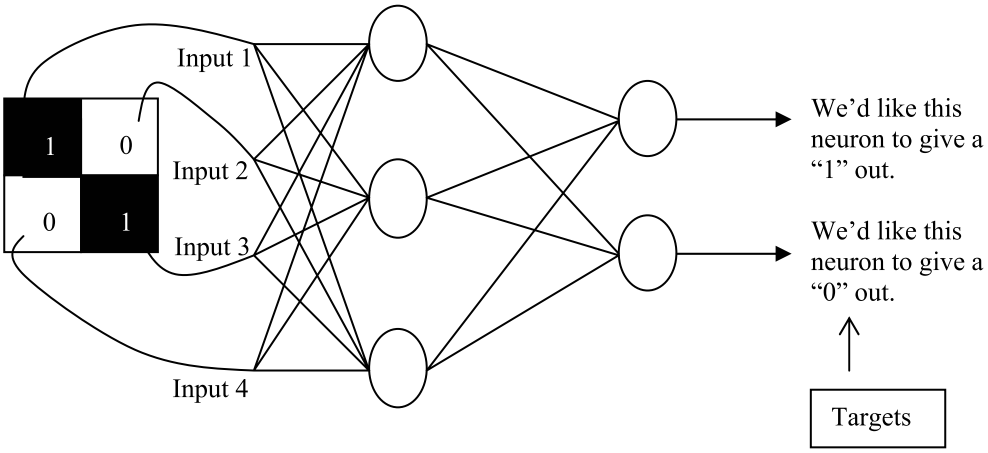
When pattern 2 is applied the outputs should be: Output 1 = 0, Output 0 = 1.

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
1
0
0
1
Input 2
Input 3
Input 4
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.