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.

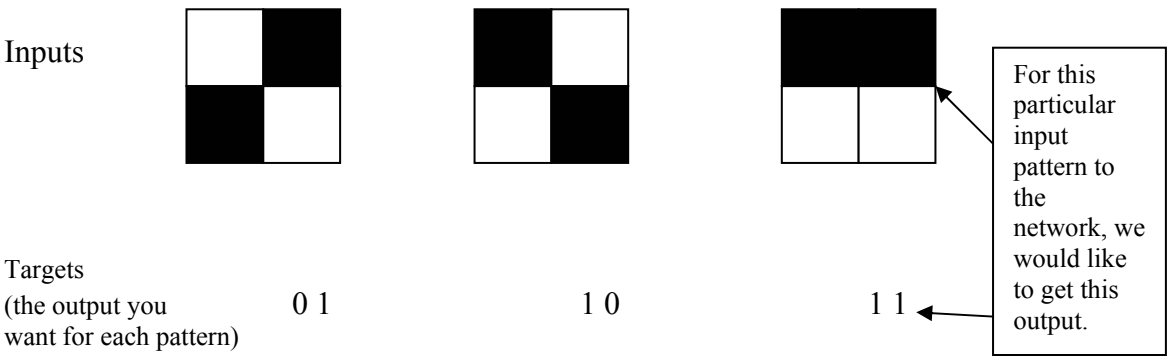
3.1 The algorithm

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.

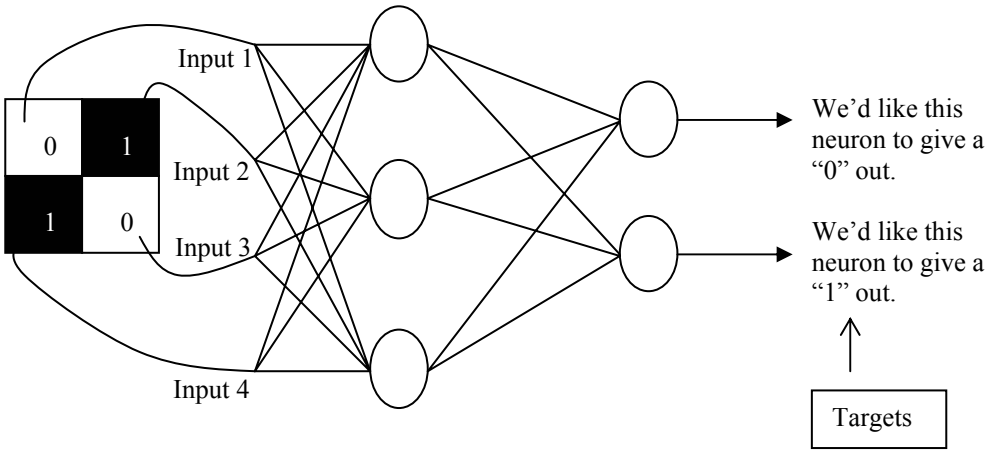
A Back Propagation network learns by example. You give the algorithm examples of what you want the network to do and it changes the network's weights so that, when training is finished, it will give you the required output for a particular input. Back Propagation networks are ideal for simple Pattern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



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.



Tutorial question 3.1:

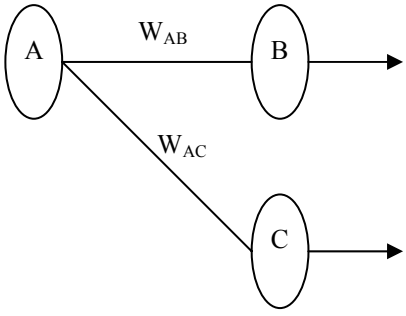
Redraw the diagram in figure 3.2 to show the inputs and targets for the second pattern.

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:

$$Error_B = Output_B (1-Output_B)(Target_B – Output_B)$$

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.

$$W^+_{AB} = W_{AB} + (Error_B \times Output_A)$$

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.

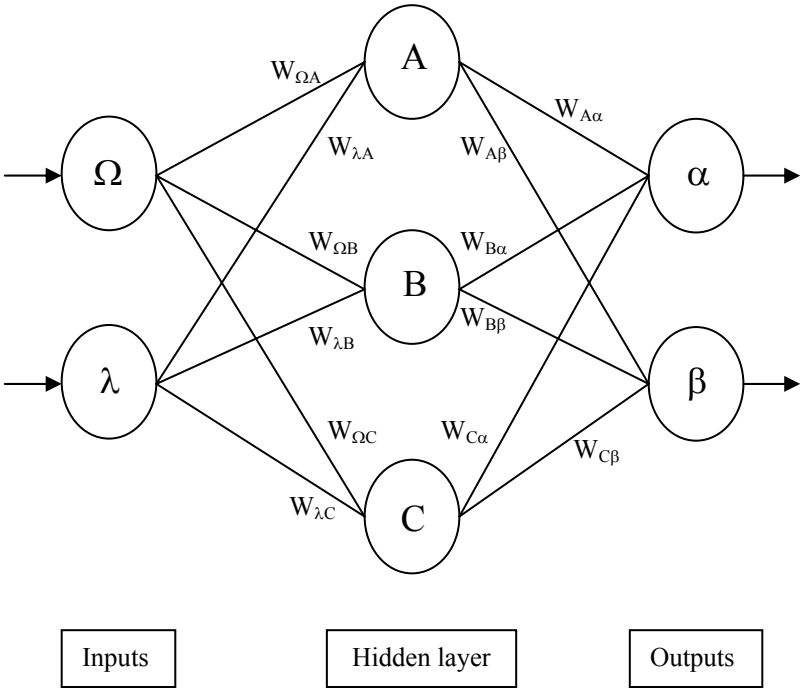
$$Error_A = Output_A (1 - Output_A)(Error_B W_{AB} + Error_C W_{AC})$$

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

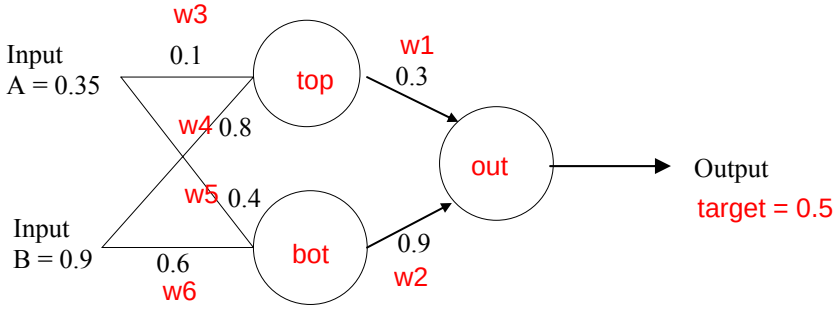
$$W^+_{\lambda A} = W_{\lambda A} + \eta\delta_A\, in_\lambda \qquad W^+_{\Omega A} = W^+_{\Omega A} + \eta\delta_A\, in_\Omega$$
$$W^+_{\lambda B} = W_{\lambda B} + \eta\delta_B\, in_\lambda \qquad W^+_{\Omega B} = W^+_{\Omega B} + \eta\delta_B\, in_\Omega$$
$$W^+_{\lambda C} = W_{\lambda C} + \eta\delta_C\,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

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.

Note that learning rate is set to 1.

Answer:

(i) running through sigmoid

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 $\delta=(t-o)(1-o)o = (0.5-0.69)(1-0.69)0.69 = -0.0406$.

New weights for output layer

$w1^+ = w1+(\delta \times \text{input}) = 0.3 + (-0.0406 \times 0.68) = 0.272392$.
$w2^+ = w2+(\delta \times \text{input}) = 0.9 + (-0.0406 \times 0.6637) = 0.87305$.

Errors for hidden layers:

$\delta 1 = \delta \times w1 = -0.0406 \times 0.272392 \times (1-o)o = -2.406 \times 10^{-3}$
$\delta 2 = \delta \times w2 = -0.0406 \times 0.87305 \times (1-o)o = -7.916 \times 10^{-3}$

New hidden layer weights:

$w3^+ = 0.1 + (-2.406 \times 10^{-3} \times 0.35) = 0.09916$.
$w4^+ = 0.8 + (-2.406 \times 10^{-3} \times 0.9) = 0.7978$.
$w5^+ = 0.4 + (-7.916 \times 10^{-3} \times 0.35) = 0.3972$.
$w6^+ = 0.6 + (-7.916 \times 10^{-3} \times 0.9) = 0.5928$.

(iii)

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