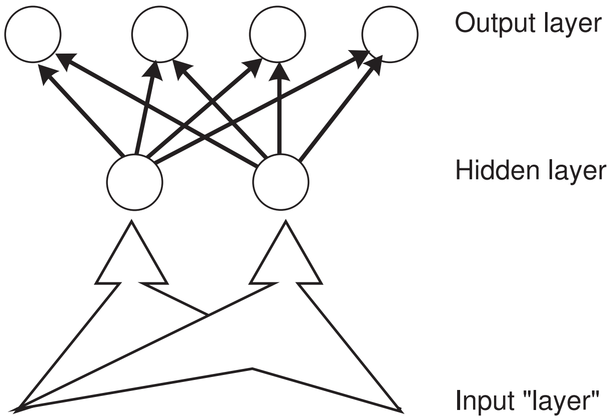
4: Multilayer nets and backpropagation

Kevin Gurney

Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

1 Introduction

Recall that the goal we set out to achieve last time was to train a two layer net *in toto* without the clumsy approach of training the intermediate layer separately and then hand-crafting the output layer. This also required prior knowledge about the way the training patterns fell in pattern space. A new training rule was introduced - the *delta rule* - which, it was claimed, could be generalised from the single unit/layer case to multilayer nets. This is now done heuristically for a two layer net of semilinear nodes*.



2 layer net as drawn last time

2 Backpropagation

2.1 Theory - where does it come from?

We analysed the delta rule with just one node. With more than one node on the output layer (N, say) the error has to be summed over all nodes

$$E_p = \frac{1}{2} \sum_{j=1}^{N} (t^j - y^j)^2 \tag{1}$$

*For a full proof, see the additional sheet given out in the lecture. This is *not* an examinable part of the course!

The idea is still to perform a gradient descent on the error considered as a function of the weights. This time, however, we are to take into account *all* the weights, for both *hidden* and *output* nodes. The former are nodes in the intermediate layer(s) which we do not have direct access to for the purposes of training (we can't say what their output will be). The output nodes are the ones which tell us the net's response to an input and to which we may show a supervisory or target value during training.

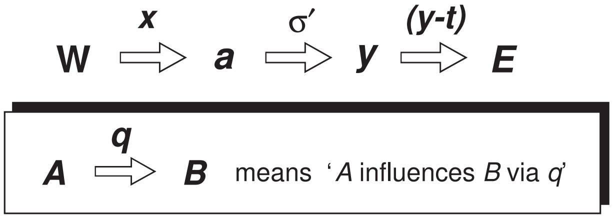
The analysis for the output nodes is just the same as in the delta rule given in eqn. (8) last time.

$$\Delta w_i^j = \alpha \sigma'(a^j)(t^j - y^j)x_i^j \tag{2}$$

Where a superscript has been introduced to denote which node is being described. This follows because the gradient of the error with respect a weight on the jth node can only be affected by that part of (1) which contains reference to that node.

In order to gain insight, it is useful to split the right hand side (RHS) of (2) up in the following way. The term $(t^j - y^j)$ represents a measure of the error on the jth node. The term $\sigma'(a^j)$ relates to how quickly (rate of change or slope) the activation can change the output (and hence the error). If this is small then we are on one of the 'tails' of the sigmoid and changing the activation won't change the output by much. If, however, it is large, then we can expect a rapid change for a given change in activation (see Qn. 3, problem sheet 3). The factor of x_i^j is related to, in turn, the amount that the ith input has affected the activation. If it is zero then that input cannot be 'held responsible' for the error and so the weight change should also be zero. If on the other hand, it is large (1, say) then the ith input had a large contribution to the activation which gave the error and so the weight needs to be changed by a correspondingly larger amount.

To summarise: x_i^j tells us how much the ith input was 'responsible for' the activation; $\sigma'(a^j)$ tells us, in turn, how fast the output is changing in response to changes in the activation and $(t^j - y^j)$ is the error on the jth node. It is therefore not unreasonable that the product of these gives us something that is a measure of the rate of change (slope) of the error with respect to the weight w_i^j. The situation is shown diagramatically below.



Causal chain for determining error

Using our previous notation we may combine two of these elements as follows †

$$\delta^j = \sigma'(a^j)(t^j - y^j) \tag{3}$$

The delta rule for output units may now be written

†Note the learning rate has now been excluded from the definition which we used when dealing with TLUs - this is, in fact, the normal convention.

$$\Delta w_i^j = \alpha \delta^j x_i \tag{4}$$

Consider now, the two layer net in the first diagram and, in particular, the kth hidden node. The problem in assigning a set of weight changes to this type of node is related to the so-called *credit assignment problem* - how much influence has this node had on the error. The resulting weight changes will be a result of including the right combination of 'responsibility' factors, rates of change and errors in the same way that these occurred for the output nodes. A full mathematical derivation is supplied in the supplement to these notes; this however, in itself, does not give insight. The purpose here is to shed some light on where the resulting formula comes from. As a start, we notice that, for the ith input to the hidden node, the value of the input will play a similar role as before so we might write

$$\Delta w_i^k = \alpha \delta^k x_i \tag{5}$$

and the task now is to find what goes into the factor δ^k

To this end, consider just a single output from the hidden node to an output node.

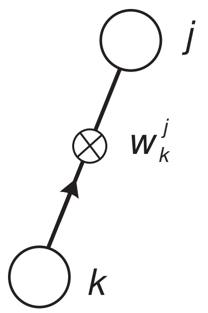


diagram of kth hidden and jth output nodes

The effect this node has on the error depends on two things: first how much it can influence the output of node j and, via this, how the output of node j affects the error. The more k can affect j, the greater the effect we expect there to be on the error, but this will only be significant if j is having some effect on the error at its output. The contribution that node j makes towards the error is, of course, expressed in the 'delta', for that node - δ^j. The influence that k has on j is given by the weight w_k^j. Therefore we may expect the find the product $w_k^j \delta^j$ in the expression for δ^k. However, the kth node may be giving output to several nodes and so the contributions to the error from all of these must be taken into account. Thus, we must sum these products over all j. Finally, the factor $\sigma'(a^k)$ will occur for exactly the same reasons that it did for the output nodes. This results in the following expression for δ^k

$$\delta^k = \sigma'(a^k) \sum_{j \in I_k} \delta^j w_k^j \tag{6}$$

where I_k is the set of nodes which take an input from the hidden node k. This set is called the *fan-out* of k. Using this in (5) gives us a means for calculating the weight changes for the hidden nodes.

2.2 Practice - using the training rules

It remains to develop a training algorithm around the rules we have developed. This basic iteration is the same as for the perceptron rule or delta rule

```
repeat
    for each training pattern
        train on that pattern
    end for loop
until the error is 'acceptably low'
```

Before examining the crucial step 'train on a pattern' a couple of points need comment. First, it is implied in the algorithm defined above that there is a fixed presentation sequence of training vectors. The alternative is to present vectors randomly. If we were to imagine our network in a real learning environment then this second option is more realistic. Empirically, however, it is often found that training is faster if the vectors are ordered in some way and that order is maintained in presentation. Second, what constitutes an acceptable error? One possible definition for Boolean training sets might be to ensure that all output nodes had responses in the correct one of the pair of intervals [0, 0.5], [0.5, 1] as defined by the target, since then, if we were to replace the sigmoid with a hard limiting threshold, the 'correct' response would be guaranteed. Another might simply prescribe some low value like 0.001. Whatever approach is used, one has to interpret the significance of the criterion.

The main step of training on a pattern may now be expanded into the following steps.

1. Present the pattern at the input layer
2. Let the hidden units evaluate their output using the pattern
3. Let the output units evaluate their output using the result in step 2) from the hidden units.

   The steps 1) - 3) are collectively known as the *forward pass* since information is flowing forward, in the natural sense, through the network.
4. Apply the target pattern to the output layer
5. Calculate the δ's on the output nodes according to (3)
6. Train each output node using gradient descent (4)
7. For each hidden node, calculate its δ according to (6)
8. For each hidden node, use the δ found in step 7) to train according to gradient descent (5).

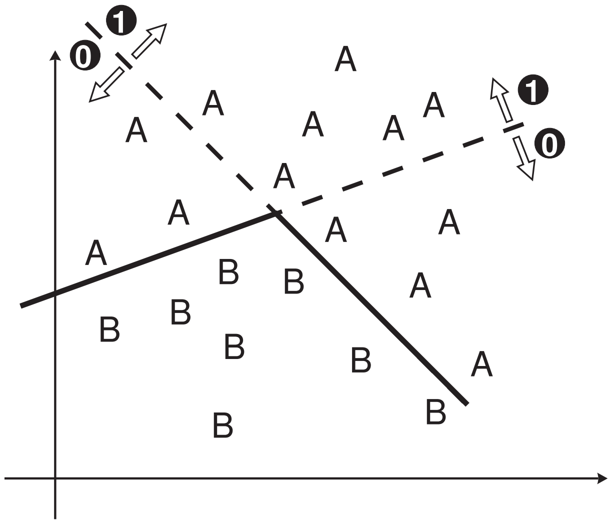
   Steps 4) - 8) are collectively known as the *backward pass*

Step 7) involves *propagating* the δ's from those output nodes in the hidden unit's fan-out *back* towards this node so that it can process them. This is where the name of the algorithm comes from.

Before going further it is useful to note some alternative terms used in the literature. The backpropagation (BP) algorithm is also known as *error backpropagation* or *back error propagation* or the *generalised delta rule*. The networks that get trained like this are sometimes known as *multilayer perceptrons* or MLPs.

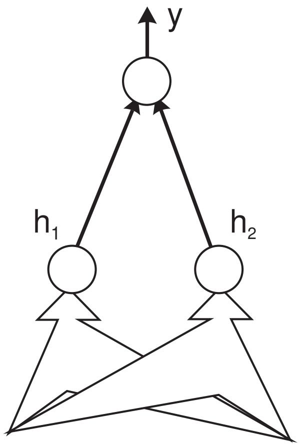
3 Non-linearly separable problems

The two layer net was originally introduced in the context of classifying more than 2 classes. Consider now, the following situation in pattern space



2 non linearly separable classes - 2 planes

The two classes A and B cannot be separated by a single hyperplane. In general we require an arbitrarily shaped *decision surface*. In the diagram, we have approximated this surface by two planes. We may now construct a two layer net to solve this problem. Each plane is determined by one of a pair of hidden nodes. Suppose each of these nodes learns to signal a '1' (or at least a value close to this with its sigmoid output) for class A. The output node can now classify A as a '1' if it learns the logical OR function.



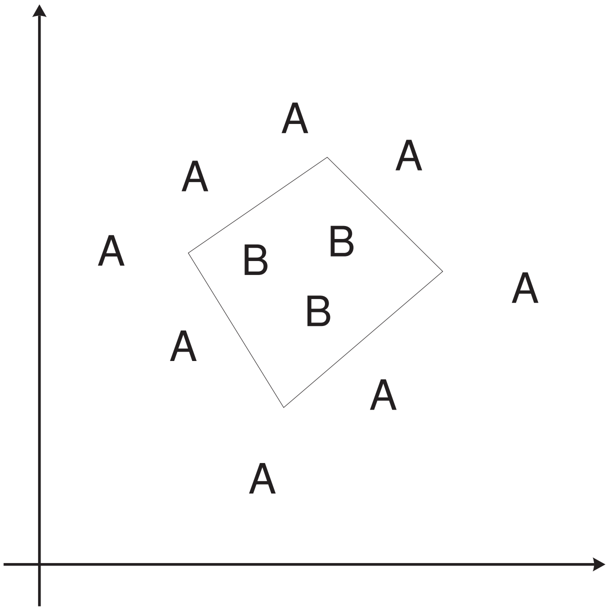
h_1	h_2	y
0	0	0
0	1	1
1	0	1
1	1	1

$A \leftrightarrow 1$

truth table for output node - OR gate

If the hidden units had coded class A in some other way then the output node would learn some other function in which a single corner of its pattern space square was 'lopped off'.

For more complex decision surfaces we need more hidden units



complex decision surface with a 'hole'

To summarise the power of the tools that have now been developed: we can train a multilayer net to perform categorisation of an arbitrary number of classes and with an arbitrary decision surface. All that is required is that we have a set of inputs and targets, that we fix the number of hyperplanes (hidden units) that are going to be used, and perform gradient descent on the error with the backpropagation algorithm. There are (as always) however, subtleties that emerge which make life difficult. The first of these concerns the number of hidden units used and relates to inadequate *training set generalisation*. The second concerns the nature of gradient descent.

4 Generalisation

Consider the situation in pattern space shown below

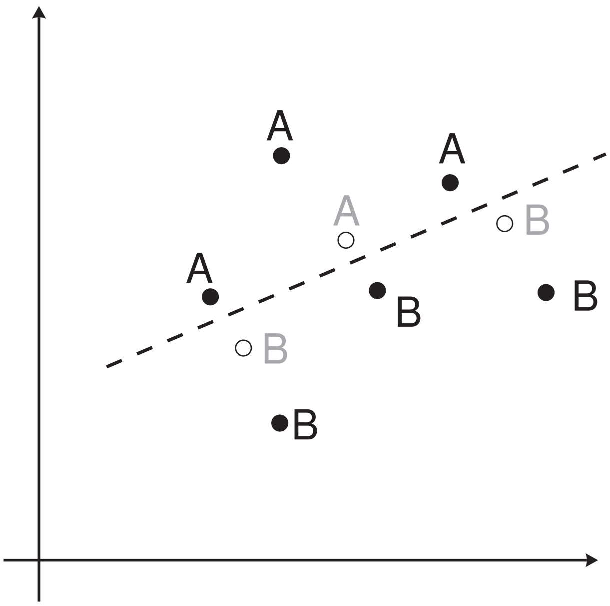
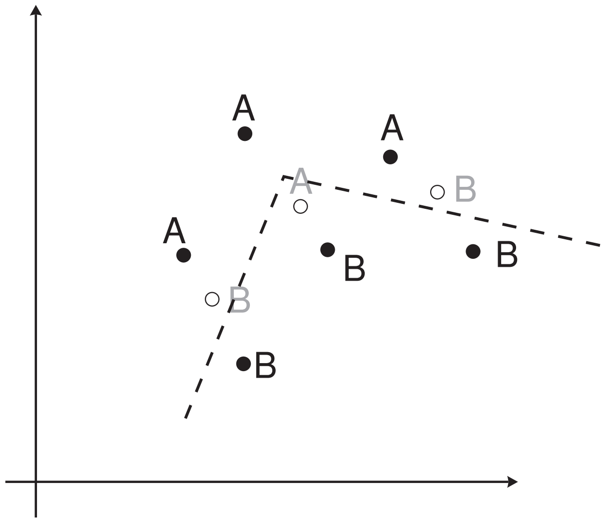


diagram sparse training - two classes

The training pattern are shown by solid dots and there are two classes A and B. Only one node is used to classify these. The circles in each class represent vectors which were not shown during training; these are *test patterns*. Representatives from each class of test data have been classified correctly, even though they were not seen during training. This is the power of the network approach and one of the main reasons for using it. The net is said to have *generalised* from the training data.

4.1 Overfitting the decision surface

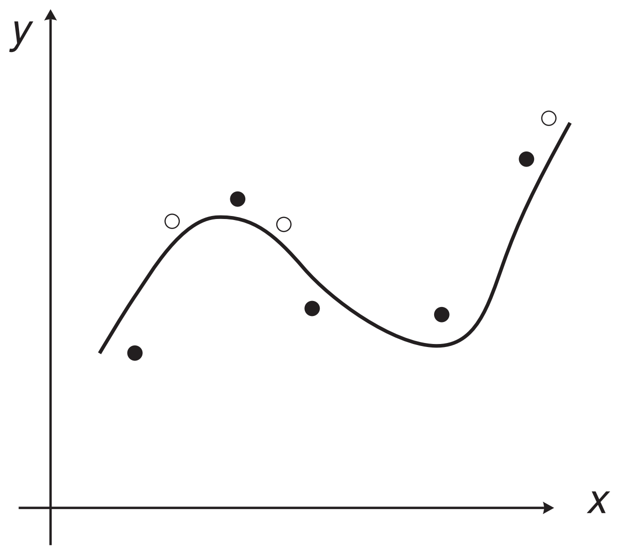
The correct classification of the test patterns shown in the last diagram would seem to vindicate the choice of a single hyperplane for the decision surface. Suppose, in fact, we had used two hyperplanes (two hidden units in a two layer net). In minimising the error, the planes might have aligned themselves as close to the training data as possible.



fitting two planes to A and B

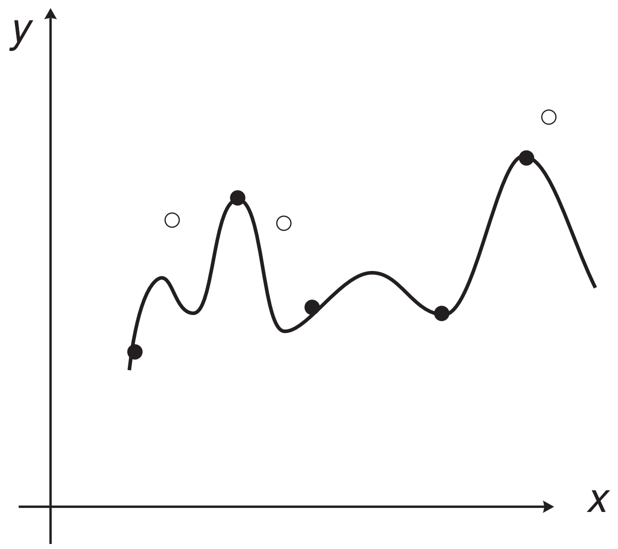
Some of the test data are now misclassified. The problem is that the network, with two hidden units, now has *too much* freedom and has fitted a decision surface to the training data which follows its intricacies in pattern space without extracting the underlying trends.

There is another way to view this in terms of the input-output function of the net. The diagram below shows, schematically, the output y of a binary (two-class) classifier, as a function of its input (this is a 1-D representation of the n-D input)



y against x for a binary classifier

The curve shown is the actual output in response to the input x, while the dots represent training data. If the curve had passed exactly through the training set, the error would be identically zero. Although this is not the case, the output has captured the underlying trend in the data. If we use more hidden units, the net has more freedom, its output can vary more quickly in response to a change in the input, and we might get a situation like that shown below



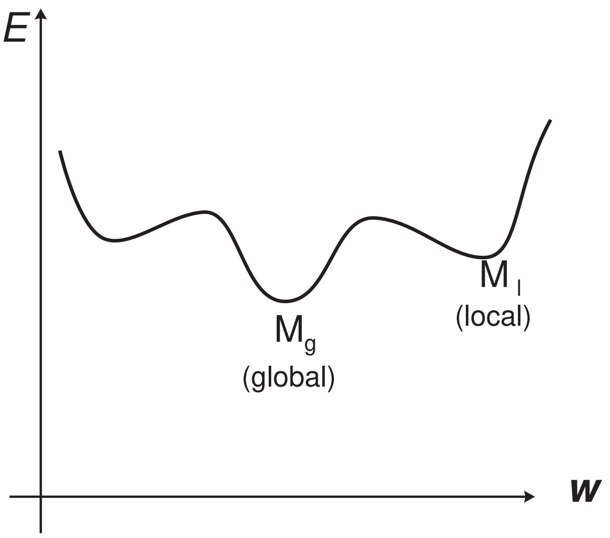
overfitting in x-y space

Now, although the curve fits almost exactly through the training data, giving almost zero error, the test data are poorly classified. The net has generalised poorly.

One of the questions that remained unanswered at the end of the last section was how to determine the number of hyperplanes or hidden units to use. At first, this might not have seemed a problem since it appears that we can always use more hidden units than are strictly necessary. There would simply be redundancy with more than one hidden node per hyperplane. This is the 'more of a good thing is better' approach. Unfortunately, as we have discovered here, things aren't so easy. Too many hidden nodes can make our net a very good look-up-table for the training set at the expense of any useful generalisation. How to fix the number of hidden nodes is an active research problem.

5 Local Minima

Consider the error function shown below.



local minimum

Suppose we start with a weight set for the network corresponding to point P. If we perform gradient descent, the minimum we encounter is the one at M_l, not that at M_g. M_l is called a *local minimum* and corresponds to a partial solution for the network in response to the training data. M_g is the *global minimum* we seek and, unless measures are taken to escape from M_l, M_g will never be reached. This problem will occur again in connection with feedback associative nets, where it will overcome by using noise in the training. Essentially, we opt for a situation where each move in weight space is governed, not only by the error gradient, but includes a random component so that sometimes we may go up the curve rather than down. In the type of nets being discussed here, however, the hope is that situations like that above don't occur.

6 Speeding up learning: the momentum term

The speed of learning is governed by the learning rate α. If this is increased too much, learning becomes unstable; the net oscillates back and forth across the error minimum. One way of overcoming the limitations thus imposed is to alter the training rule from 'pure' gradient descent to include a term which includes a proportion of the last weight change. The new rule is

$$\Delta w_i^j(n) = \alpha \delta^j x_i^j + \lambda \Delta w_i^j(n-1) \tag{7}$$

Thus, if the previous weight change $\Delta w_i^j(n-1)$ was large, so too will the new one $\Delta w_i^j(n)$. That is, the weight change carries along some momentum to the next iteration. This has a tendency to smooth out small fluctuations in the error-weight space (it is a low-pass filter). The parameter λ ('lambda') governs the contribution of the *momentum term*.

7 Further notes and reading

Backpropagation is probably the most well researched training algorithm in neural nets and forms the starting point for most people looking for a quick NN solution to a problem. There is therefore a wealth of literature on BP and its applications.

The theory and some toy applications are given in chapter 9 of PDP vol. 1.

The algorithm was actually discovered before Rumelhart & McClleland made it well-known, independently by P. Werbos and D. B. Parker. The references for these are a PhD thesis and an internal report at Stanford and were therefore not easily available.

One of the most powerful demonstrations of BP which helped it to fame was the NETtalk network of Sejnowski & Rosenberg which learned to translate written text to speech. (unfortunately this is a technical report and not easily available).

Another application is classification of sonar targets. Gorman, R.P. & Sejnowski, T. 'Analysis of hidden units in a layered network trained to classify sonar targets', Neural Networks, **1**, 75 – 89. (I have this).