Chapter 4

The Multilayer Perceptron: backprop

4.1 Objectives

After this chapter, you should

1. understand the backpropagation algorithm.
2. be able to implement the backprop algorithm.
3. understand
   (a) batch vs on-line learning.
   (b) the importance of differentiable activation functions. .
   (c) methods of speeding convergence.
   (d) stopping criteria .
   (e) local minima.
   (f) generalisation.
4. be able to list potential applications of the algorithm.

4.2 Introduction

As we have seen, the Perceptron (and Adeline) proved to be powerful learning machines but there were certain mappings which were (and are) simply impossible using these networks. Such mappings are characterised by being linearly inseparable. Now it is possible to show that many linearly inseparable mappings may be modelled by multi-layered perceptrons; this indeed was known in the 1960s but what was not known was a rule which would allow such networks to learn the mapping. Such a rule appears to have been discovered independently several times [Werbos,1974; Parker, 1985] but has been spectacularly popularised by the PDP(Parallel Distributed Processing) Group [Rumelhart et al, 1986] under the name backpropagation.

An example of a multi-layered perceptron (MLP) is shown in Figure 1.3. Activity in the network is propagated forwards via weights from the input layer to the hidden layer where some function of the net activation is calculated. Then the activity is propagated via more weights to the output neurons. Now two sets of weights must be updated - those between the hidden and output layers and those between the input and hidden layers. The error due to the first set of weights is clearly calculable by the previously described LMS rule; however, now we require to propagate backwards that part of the error due to the errors which

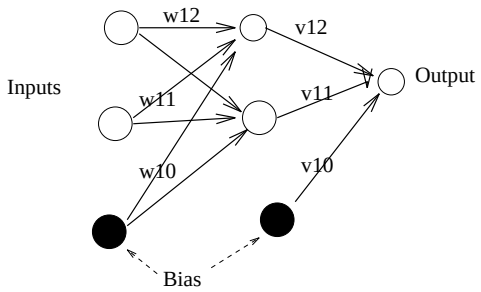

Figure 4.1: The net which will be used for the solution of the XOR problem using backpropagation

exist in the second set of weights and assign the error proportionately to the weights which cause it. You may see that we have a problem - **the credit assignment problem** - in that we must decide how much effect each weight in the first layer of weights has on the final output of the network. This assignment is the core result of the **backprop** method.

We may have any number of hidden layers which we wish since the method is quite general; however, the limiting factor is usually training time which can be excessive for many-layered networks. In addition, it has been shown that networks with a single hidden layer are sufficient to approximate any continuous function (or indeed any function with only a finite number of discontinuities) provided we use non-linear (differentiable) activation functions in the hidden layer.

4.3 The Backpropagation Algorithm

Because it is so important, we will repeat the whole algorithm in a 'how-to-do-it' form and then give a simple walk through for the algorithm when it is trained on the XOR problem:

1. Initialise the weights to small random numbers
2. Choose an input pattern, $\mathbf{x}$, and apply it to the input layer
3. Propagate the activation forward through the weights till the activation reaches the output neurons
4. Calculate the δs for the output layer $\delta_j^k = (D_j^k - y_j^k) f'(Net_j^k)$ using the desired target values for the selected input pattern.
5. Calculate the δs for the hidden layer using $\delta_i^k = \sum_{j=1}^{N} \delta_j^k w_{ji} . f'(Net_i^k)$
6. Update all weights according to $\Delta^k w_{im} = \gamma . \delta_i^k . y_m^k$
7. Repeat steps 2 to 6 for all patterns.

A final point is worth noting: the actual update rule after the errors have been backpropagated is local. This makes the backpropagation rule a candidate for parallel implementation.

4.3.1 The XOR problem

You will use the net shown in Figure 4.1 to solve the XOR problem. The procedure is

Initialisation .

- Initialise the W-weights and V-weights to small random numbers.
- Initialise the learning rate, η to a small value e.g. 0.001.
- Choose the activation function e.g. tanh().

Select Pattern It will be one of only 4 patterns for this problem. Note that the pattern chosen determines not only the inputs but also the target pattern.

Feedforward to the hidden units first, labelled 1 and 2.

$$\begin{aligned} Net_1 &= w_{10} + w_{11}x_1 + w_{12}x_2 \\ Net_2 &= w_{20} + w_{21}x_1 + w_{22}x_2 \\ y_1 &= tanh(Net_1) \\ y_2 &= tanh(Net_2) \end{aligned}$$

Now feedforward to the output unit which we will label 3

$$\begin{aligned} Net_3 &= v_{10} + v_{11}y_1 + v_{12}y_2 \\ y_3 &= tanh(Net_3) \end{aligned}$$

Feedback errors calculate error at output

$$\delta_3 = (D - y_3) * f'(Net_3) = (D - y_3)(1 - y_3^2)$$

and feedback error to hidden neurons

$$\begin{aligned} \delta_1 &= \delta_3 v_{11} f'(Net_1) = \delta_3 v_{11}(1 - y_1^2) \\ \delta_2 &= \delta_3 v_{12} f'(Net_2) = \delta_3 v_{12}(1 - y_2^2) \end{aligned}$$

Change weights

$$\begin{aligned} \Delta v_{11} &= \eta.\delta_3.y_1 \\ \Delta v_{12} &= \eta.\delta_3.y_2 \\ \Delta v_{10} &= \eta.\delta_3.1 \\ \Delta w_{11} &= \eta.\delta_1.x_1 \\ \Delta w_{12} &= \eta.\delta_1.x_2 \\ \Delta w_{10} &= \eta.\delta_1.1 \\ \Delta w_{21} &= \eta.\delta_2.x_1 \\ \Delta w_{22} &= \eta.\delta_2.x_2 \\ \Delta w_{20} &= \eta.\delta_2.1 \end{aligned}$$

Go back to Select Pattern

4.4 Backpropagation Derivation

We must first note that our activation functions will be non-linear in this chapter: if we were to be using linear activation functions, our output would be a linear combination of linear combinations of the inputs

i.e. would simply be linear combinations of the inputs and so we would gain nothing by using a three layer net rather than a two layer net.

As before, consider a particular input pattern, $\mathbf{x}^k$, we have an output y^k and target D^k. Now however we will use a non-linear activation function, $f()$. $y_i = f(Net_i) = f(\sum_j w_{ij}y_j)$ where we have taken any threshold into the weights as before. Notice that the y_j represent the outputs of neurons in the preceeding layer. Thus if the equation describes the firing of a neuron in the (first) hidden layer, we revert to our previous definition where $y_i = f(Net_i) = f(\sum_j w_{ij}x_j)$ while if we wish to calculate the firing in an output neuron the y_j will represent the firing of hidden layer neurons. However now f() must be a differentiable function (unlike the perceptron) and a non-linear function (unlike the Adaline). Now we still wish to minimise the sum of squared errors,

$$E = \sum_{k=1}^{P} E^k = \frac{1}{2}\sum_{k=1}^{P}(D^k - y^k)^2 \tag{4.1}$$

at the outputs. To do so, we find the gradient of the error with respect to the weights and move the weights in the opposite direction. Formally, $\Delta^k w_{ij} = -\gamma\frac{\partial E^k}{\partial w_{ij}}$.

Now we have, for all neurons,

$$\frac{\partial E^k}{\partial w_{ij}} = \frac{\partial E^k}{\partial Net_i^k}.\frac{\partial Net_i^k}{\partial w_{ij}} \tag{4.2}$$

and $\frac{\partial Net_i^k}{\partial w_{ij}} = y_j$. Therefore if we define $\delta_i^k = -\frac{\partial E^k}{\partial Net_i^k}$ we get an update rule of

$$\Delta^k w_{ij} = \gamma.\delta_i^k.y_j^k \tag{4.3}$$

Note how like this rule is to that developed in the previous chapter (where the last o is replaced by the input vector, $\mathbf{x}$). However, we still have to consider what values of δ are appropriate for individual neurons. We have

$$\delta_i^k = -\frac{\partial E^k}{\partial Net_i^k} = -\frac{\partial E^k}{\partial y_i^k}.\frac{\partial y_i^k}{\partial Net_i^k} \tag{4.4}$$

for all neurons. Now, for all output neurons, $\frac{\partial E^k}{\partial y^k} = -(D^k - y^k)$, and $\frac{\partial y_i^k}{\partial Net_i^k} = f'(Net_i^k)$. This explains the requirement to have an activation function which is differentiable. Thus for output neurons we get the value

$$\delta_i^k = (D_i^k - y_i^k)f'(Net_i^k) \tag{4.5}$$

However, if the neuron is a hidden neuron, we must calculate the responsibility of that neuron's weights to the final error. To do this we take the error at the output neurons and propagate this backward through the current weights (the very same weights which were used to propagate the activation forward). Consider a network with N output neurons and H hidden neurons. We use a chain rule to calculate the effect on unit i in the hidden layer:

$$\frac{\partial E^k}{\partial y_i^k} = \sum_{j=1}^{N}\frac{\partial E^k}{\partial Net_j^k}.\frac{\partial Net_j^k}{\partial y_i^k} = \sum_{j=1}^{N}\frac{\partial E^k}{\partial Net_j^k}.\frac{\partial}{\partial y_i^k}\sum_{m=1}^{H} w_{jm}y_m^k = \sum_{j=1}^{N}\frac{\partial E^k}{\partial Net_j^k}.w_{ji} = -\sum_{j=1}^{N}\delta_j^k.w_{ji} \tag{4.6}$$

Note that the terms $\frac{\partial E^k}{\partial Net_j^k}$ represent the effect of change on the error from the change in activation in the output neurons. On substitution, we get

$$\delta_i^k = -\frac{\partial E^k}{\partial Net_j^k} = -\frac{\partial E^k}{\partial y_i^k}.\frac{\partial y_i^k}{\partial Net_j^k} = \sum_{j=1}^{N}\delta_j^k w_{ji}.f'(Net_i^k) \tag{4.7}$$

This may be thought of as assigning the error term in the hidden layer proportional to the hidden neuron's contribution to the final error as seen in the output layer.

4.5 Issues in Backpropagation

4.5.1 Batch vs On-line Learning

The backpropagation algorithm is only theoretically guaranteed to converge if used in batch mode i.e. if all patterns in turn are presented to the network, the total error calculated and the weights updated in a separate stage at the end of each training epoch. However, it is more common to use the on-line (or pattern) version where the weights are updated after the presentation of each individual pattern. It has been found empirically that this leads to faster convergence though there is the theoretical possibility of entering a cycle of repeated changes. Thus in on-line mode we usually ensure that the patterns are presented to the network in a random and changing order.

The on-line algorithm has the advantage that it requires less storage space than the batch method. On the other hand the use of the batch mode is more accurate: the on-line algorithm will zig-zag its way to the final solution. It can be shown that the expected change (where the expectation is taken over all patterns) in weights using the on-line algorithm is equal to the batch change in weights.

4.5.2 Activation Functions

The most popular activation functions are the logistic function and the tanh() function. Both of these functions satisfy the basic criterion that they are differentiable. In addition, they are both monotonic and have the important property that their rate of change is greatest at an intermediate values and least at extreme values. This makes it possible to saturate a neuron's output at one or other of their extreme values. The final point worth noting is the ease with which their derivatives can be calculated:

- if f(x) = tanh(bx), then f'(a) = b(1 - f(a)*f(a));
- similarly, if f(x) = 1/(1+exp(-bx)) then f'(a)=bf(a)(1 - f(a)).

There is some evidence to suggest that convergence is faster when tanh() is used rather than the logistic function. Note that in each case the target function must be within the output range of the respective functions. If you have a wide spread of values which you wish to approximate, you must use a linear output layer.

4.5.3 Initialisation of the weights

The initial values of the weights will in many cases determine the final converged network's values. Consider an energy surface with a number of energy wells; then, if we are using a batch training method, the initial values of the weights constitute the only stochastic element within the training regime. Thus the network will converge to a particular value depending on the basin in which the original vector lies. There is a danger that, if the initial network values are sufficiently large, the network will initially lie in a basin with a small basin of attraction and a high local minimum; this will appear to the observer as a network with all weights at saturation points (typically 0 and 1 or +1 and -1). It is usual therefore to begin with small weights uniformly distributed inside a small range. Haykin recommends (p162) the range $(-\frac{2.4}{F_i}, +\frac{2.4}{F_i})$ where F_i is the fan-in of the i^{th} unit.

4.5.4 Momentum and Speed of Convergence

The basic backprop method described above is not known for its fast speed of convergence. Note that though we could simply increase the learning rate, this tends to introduce instability into the learning rule causing wild oscillations in the learned weights. It is possible to speed up the basic method in a number of ways. The simplest is to add a momentum term to the change of weights. The basic idea is to make the new

change of weights large if it is in the direction of the previous changes of weights while if it is in a different direction make it smaller. Thus we use $\Delta w_{ij}(t+1) = (1-\alpha).\delta_j.o_i + \alpha\Delta w_{ij}(t)$, where the α determines the influence of the momentum. Clearly the momentum parameter α must be between 0 and 1. The second term is sometimes known as the 'flat spot avoidance' term since them momentum has the additonal property that it helps to slide the learning rule over local minima(see below).

4.5.5 Stopping Criteria

We must have a stopping criterion to decide when our network has solved the problem in hand. It is possible to stop when:

1. the Euclidean norm of the gradient vector reaches a sufficiently small value since we know that at the minimum value, the rate of change of the error surface with respect to the weight vector is zero. There are two disadvantages with this method:
   - it may lead to excessively long training times
   - it requires calculating the gradient vector of the error surface with respect to the weights
2. the rate of change of the mean squared error is sufficiently small
3. the mean squared error is sufficiently small
4. a mixture of the last two criteria

Typically we will stop the learning before each pattern has been perfectly learned (see Section 4.5.7) so that learning will stop when outputs are greater than 0.9 or less than 0.1.

4.5.6 Local Minima

Error descent is bedeviled with local minima. You may read that local minima are not much problem to ANNs, in that a network's weights will typically converge to solutions which, even if they are not globally optimal, are good enough. There is as yet little analytical evidence to support this belief. An heuristic often quoted is to ensure that the initial (random) weights are such that the average input to each neuron is approximately unity (or just below it). This suggests randomising the initial weights of neuron j around the value $\frac{1}{\sqrt{N}}$, where N is the number of weights into the j_{th} neuron. A second heuristic is to introduce a little random noise into the network either on the inputs or with respect to the weight changes. Such noise is typically decreased during the course of the simulation. This acts like an annealing schedule(see Chapter 7).

4.5.7 Weight Decay and Generalisation

While we wish to see as good a performance as possible on the training set, we are even more interested in the network's performance on the test set since this is a measure of how well the network generalises. Remember the training set is composed of instances for which we already have the answer. We wish the network to give accurate results on data for which we do not already know the answer. There is a trade-off between accuracy on the training set and accuracy on the test set.

Note also that perfect memory of the patterns which are met during training is essentially a look-up table and look-up tables are discontinuous in that the item looked-up either is found to correspond to a particular result or not. Also generalisation is important not only because we wish a network to perform on new data which it has not seen during learning but also because we are liable to have data which is noisy, distorted or incomplete. Consider the set of 5 training points in Figure 4.2. We have shown two possible models for

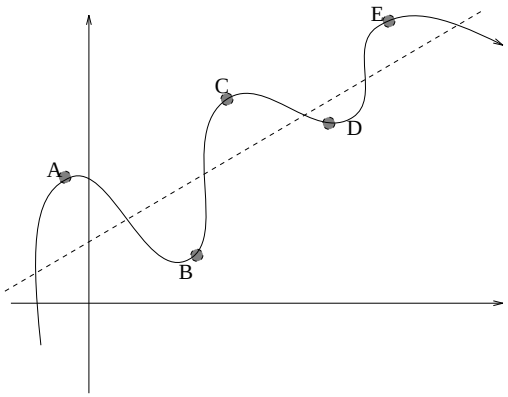

Figure 4.2: A set of data points may be approximated by either the straight line or the curve. Either would seem to fit the data; however, the line may give a better generalisation performance (on the test set) than the curve which is actually producing a lower error on the training set.

these data points - a linear model (perhaps the line minimising the squared error) and a polynomial fit which models the five given points exactly.

The problem with the more explicit representation given by the curve is that it may be misleading in positions other than those directly on the curve. If a neural network has a large number of weights (each weight represents a degree of freedom), we may be in danger of overfitting the network to the training data which will lead to poor performance on the test data. To avoid this danger we may either remove connections explicitly or we may give each weight a tendancy to decay towards zero. The simplest method is $w_{ij}^{new} = (1-\epsilon)w_{ij}^{old}$ after each update of the weights. This does have the disadvantage that it discourages the use of large weights in that a single large weight may be decayed more than a lot of small weights. More complex decay routines can be found which will encourage small weights to disappear.

4.5.8 Adaptive parameters

A heuristic sometimes used in practice is to assign learning rates to neurons in output layers a smaller value than those for hidden layers since the last layers usually have larger local gradients than the early layers and we wish all neurons to learn at the same rate.

Since it is not easy to choose the parameter values a priori one approach is to change them dynamically. e.g. if we are using too small a learning rate, we will find that the error E is decreasing consistently but by too little each time. If our learning rate is too large, we will find that the error is decreasing and increasing haphazardly. This suggests adapting the learning rate according to a schedule such as

$$\begin{aligned} \Delta\eta &= +a, \quad \text{if } \Delta E < 0 \text{ consistently} \\ \Delta\eta &= -b\eta, \quad \text{if } \Delta E > 0 \end{aligned} \tag{4.8}$$

This schedule may be thought of as increasing the learning rate if it seems that we are consistently going in the correct direction but decreasing the learning rate if we are having to change direction sometimes. Notice however that such a method implicitly requires a separate learning parameter for each weight.

4.5.9 The number of hidden neurons

The number of hidden nodes has a particularly large effect on the generalisation capability of the network: networks with too many weights (too many degrees of freedom) will tend to memorise the data; networks

with too few will be unable to perform the task allocated to it. Therefore many algorithms have been derived to create neural networks with a smaller number of hidden neurons. Two obvious methods present themselves

1. Prune weights which are small in magnitude. Such weights can only be refining classifications which have already been made and are in danger of modelling the finest features of the input data
2. Grow networks till their performance is sufficiently good on the test set

4.6 Application - A classification problem

4.6.1 Theory

Consider an m-class classification problem. Let us create a network with m output neurons and require each neuron to output 1 when an input $\mathbf{x}$ is in its class and 0 otherwise. Now input pattern $\mathbf{x}^\mu$. Then the response of the trained MLP is $\mathbf{y}^\mu$ where

$$\mathbf{y}^\mu = (F_1(\mathbf{x}^\mu), F_2(\mathbf{x}^\mu), ..., F_m(\mathbf{x}^\mu)) \tag{4.9}$$

Then we can view the F values as forming an m-dimensional vector and write

$$\mathbf{y}^\mu = \mathbf{F}(\mathbf{x}^\mu) \tag{4.10}$$

where the function $\mathbf{F}$ is derived as a minimisation of the function

$$E = \frac{1}{2N} \sum_{\mu=1}^{N} \| \mathbf{t}^\mu - \mathbf{F}(\mathbf{x}^\mu) \|^2 \tag{4.11}$$

where $\mathbf{t}^\mu$ is the training signal when $\mathbf{x}^\mu$ is presented to the network. Note that $\mathbf{t}^\mu$ is an m-dimensional vector with 0 everywhere except the j^{th} position where j is the class of input $\mathbf{x}^\mu$. If we have enough training examples, then we can invoke the law of large numbers to show that the minimisation of the above function is equivalent to minimising the function over the whole input distribution. Also the optimum vector over all has the property that the converged weight $\mathbf{w}*$ is equal to the conditional expectation of the training examples, $E(\mathbf{t}^j|\mathbf{x})$, the *a posteriori* class probability of class j. This suggests the classification rule:

Classify input $\mathbf{x}$ in class C_j if

$$F_j(\mathbf{x}) > F_k(\mathbf{x}), \text{ for all } k \neq j \tag{4.12}$$

If the class distributions are disjoint, this will give us an unambiguous classification rule since we will have only a single neuron firing.

4.6.2 An Example

Using the Stuttgart Neural Network Simulator, a network was trained to identify all of the 26 roman letters, a,..,z presented on a 5*7 display. Training was stopped sufficiently early that some degree of generalisation was possible. Figure 4.3 shows the network's response during training to the letter B while Figure 4.4 shows the network's response to a noisy B.

Notice how large the effect of just one "pixel" change is on the network's output: the network is "hedging its bets" between classifying the letter as a B or a D. The single "pixel" was specifically chosen because it has just this effect - other "pixels" have a much smaller effect. This "pixel" was chosen for this demonstration precisely because it is so important to the classification process particularly to the differentiation of Bs and Ds. Of interest in this example also is the effect on the hidden neurons of the single pixel change in the input layer. Clearly this pixel was inhibiting pixels 1 and 10 in the hidden layer; in this case we can see quite explicitly how the network is differentiating between the two patterns B and D.

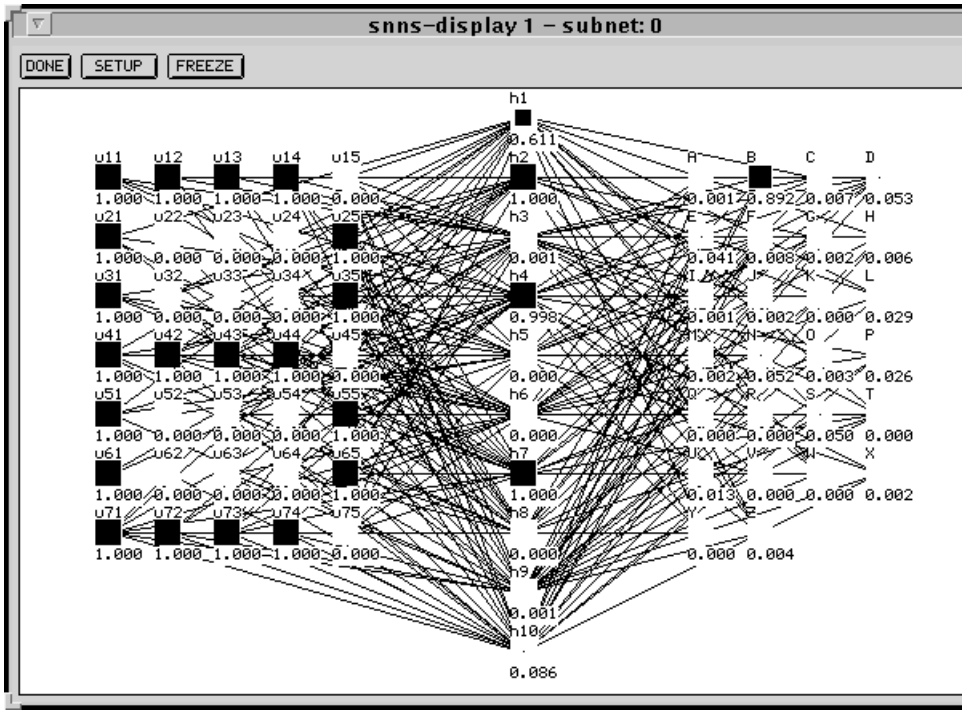

Figure 4.3: The Stuttgart Neural Network Simulator has been trained on the 26 letters. Here we show the response of the network to the presentation on the 5*7 display of the letter B

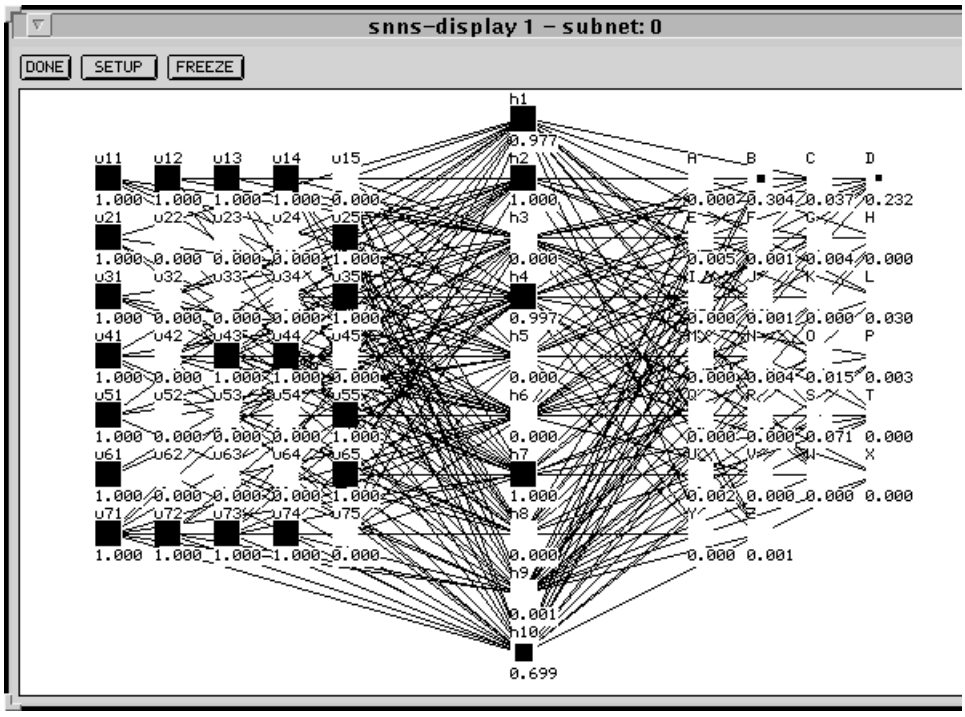

Figure 4.4: The input pattern B has been changed in just one pixel yet the effect on the network's response is fairly large. The network is responding maximally to the pattern as both B and D yet an observer could not be sure that it was seeing either

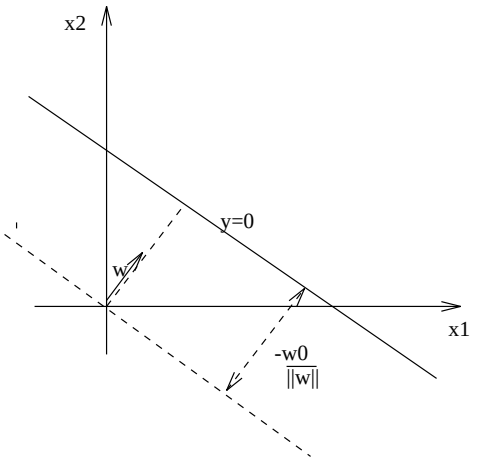

Figure 4.5: A linear discriminant line in a two dimensional space

4.7 Linear Discrimination

A linear feedforward network with a single output neuron is defined by the rule

$$y = \mathbf{w}^T\mathbf{x} + w_0 \tag{4.13}$$

The parameter w_0 is known as the bias (or threshold).

We can view this as a discriminant function if we assign a vector $\mathbf{x}$ to class C_1 when $y > 0$ and to class C_2 when $y < 0$. It can be shown that if we have 2 classes whose instances are drawn from Gaussian distributions and if the classes have equal covariance matrices (which implies in particular that their variances are equal) this function is the best we can get to discriminate between the classes and y=0 is that line(actually hyperplane) on one side of which we have class C_1 and on the other side of which we have class C_2.

Then we have, on the discrimination line, $\mathbf{w}^T\mathbf{x} + w_0 = 0$. Now if $\mathbf{x}_A$ and $\mathbf{x}_B$ are any two points lying on the discrimination line, then $\mathbf{w}^T\mathbf{x}_A + w_0 = 0$ and $\mathbf{w}^T\mathbf{x}_B + w_0 = 0$ and so $\mathbf{w}^T(\mathbf{x}_A - \mathbf{x}_B) = 0$. So $\mathbf{w}$ is perpendicular to the discrimination line and so determines the orientation of the discrimination line. The w_0 parameter on the other hand determines the distance of the line from the origin (see Figure 4.5) since this distance, $\| \mathbf{x} \|$ is determined by

$$\begin{aligned} \| \mathbf{x} \| \| \mathbf{w} \| \cos\theta &= \mathbf{w}.\mathbf{x} = \mathbf{w}^T\mathbf{x} \\ \text{And so, } \| \mathbf{x} \| &= \frac{\mathbf{w}^T\mathbf{x}}{\| \mathbf{w} \|} = \frac{-w_0}{\| \mathbf{w} \|} \end{aligned}$$

So the line is completely determined by the weights. Alternatively, we can view the data as a set of 3 dimensional data comprising $(1, \mathbf{x})$ where the 1 is the bias. Now we have a two dimensional discrimination plane which passes through the origin in a 3 dimensional space (see Figure 4.6) and the classes are defined with respect to the intersection of the two planes as shown.

We can introduce several classes by having several output neurons each with its own weight vector. Then we have a set of boundaries (discrimination lines) between classes dependent on the weights and biases of the respective neurons. We use the simple rule that $\mathbf{x} \in C_i \iff i = arg\max_i y_i$.

It is readily shown that the regions formed are convex (i.e. if A and B are in a region, so is every point on the line AB). Consider Figure 4.7. Then let A be represented by the vector $\mathbf{x}_A$ and B be represented by $\mathbf{x}_B$. Then P can be represented by

$$\mathbf{x}_P = \alpha\mathbf{x}_A + (1 - \alpha)\mathbf{x}_B \tag{4.14}$$

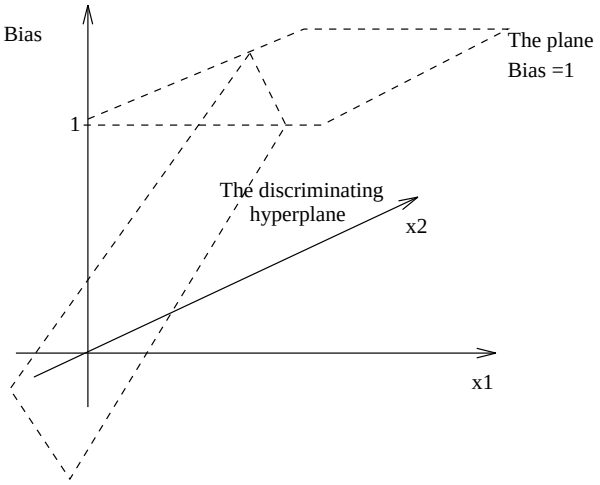

Figure 4.6: The slanting hyperplane discriminates between the two classes. We are interested in those (x_1, x_2) values which are on the plane Bias =1 which determines the classes

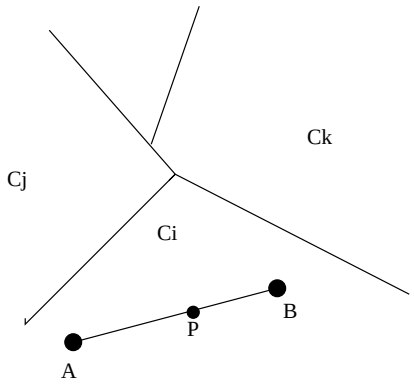

Figure 4.7: A representation of several classes which show convex regions

where $0 \leq \alpha \leq 1$. Since both A and B are in C_i, they satisfy $y_i(\mathbf{x}_A) > y_j(\mathbf{x}_A), \forall j$ and $y_i(\mathbf{x}_B) > y_j(\mathbf{x}_B), \forall j$. So

$$\begin{aligned} y_i(\mathbf{x}_P) &= \alpha y_i(\mathbf{x}_A) + (1-\alpha) y_i(\mathbf{x}_B) \\ &> \alpha y_j(\mathbf{x}_A) + (1-\alpha) y_j(\mathbf{x}_B), \forall j \\ &= y_j(\mathbf{x}_P), \forall j \end{aligned}$$

So P is also classified as being in Class i.

Now the perceptron is guaranteed to converge to a line which will discriminate between classes if such a line can be found. The LMS rule can be shown to find the best linear discriminant between the classes under the conditions stated above. Notice that if we add a non-linearity into the network (such as the logistic function or tanh()) we still have a linear discriminant function (since these functions are monotonic). i.e. the network performing

$$y_i = f(\mathbf{w}_i^T \mathbf{x} + w_{i0}) \tag{4.15}$$

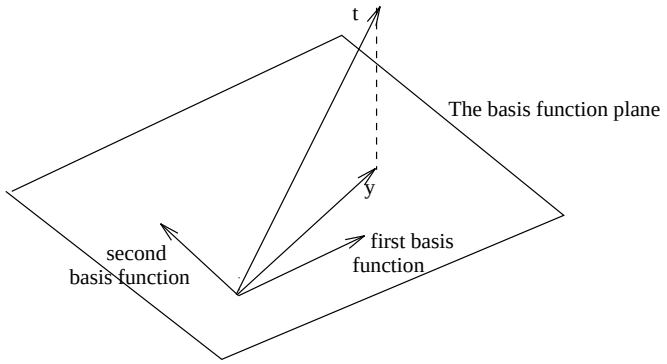

Figure 4.8: The target value, t, is outwith the basis function plane. Its best projection is y which can be learned by a simple LMS algorithm.

still gives a line (or hyperplane) between the classes where y=0 since

$$\begin{aligned} f(\mathbf{w}_i^T\mathbf{x}) &> f(\mathbf{w}_j^T\mathbf{x}) \\ \text{if and only if} & \\ \mathbf{w}_i^T\mathbf{x} &> \mathbf{w}_j^T\mathbf{x} \end{aligned}$$

though you will change the position of the discriminant function. This corresponds to the non-linear LMS rule from chapter 3.

4.7.1 Multilayered Perceptrons

Notice however that we can consider an alternative nonlinearity e.g.

$$y_i = \mathbf{w}_i^T\mathbf{f}(\mathbf{x}) \tag{4.16}$$

Here the function $\mathbf{f}()$ are also vectors. If we use different functions for each f() we call them basis functions and usually write for a single output neuron network

$$y = \sum_{j=0}^{M} w_j\phi_j(\mathbf{x}) \tag{4.17}$$

An example of this is the network defined by

$$y = \sum_{j=0}^{M} w_j \tanh(\mathbf{v}.\mathbf{x}) \tag{4.18}$$

Notice that this corresponds to a backpropagation network with weights $\mathbf{v}$ between the input and hidden neurons, a non-linear activation function at the hidden layer and an identity function for the output layer neurons. This type of backprop network can be shown to be capable of approximating any continuous function (or even a continuous function with a finite number of discontinuities). A geometric interpretation is shown in Figure 4.8. We can view the non-linearities in the hidden layer as forming a new basis (set of axes) for the data and then the feedforward through the network acts like a projection onto this new basis. Then the error between the output and the target values can be given by

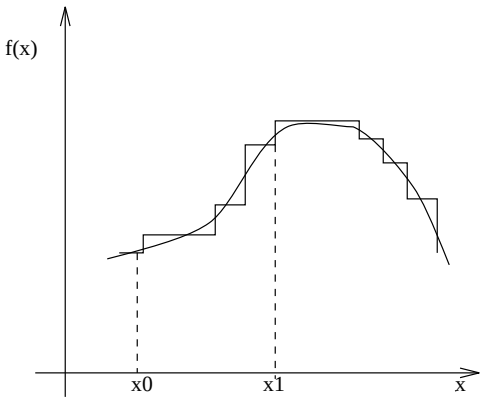

Figure 4.9: An approximation of a continuous function with a set of step functions.

$$E = \frac{1}{2} \parallel \sum_{j=0}^{M} w_j \phi_j(\mathbf{x}) - \mathbf{t} \parallel^2 \tag{4.19}$$

As usual we can minimise this by error descent using

$$\frac{\partial E}{\partial w_i} = \phi_i(\mathbf{x})(y - t) \tag{4.20}$$

Now if E can be made 0, it can only be when $(\mathbf{y} - \mathbf{t}) = 0$ i.e. (see Figure) when the target lies precisely in the basis function's plane. If there is a residual error it is caused by the continuing existence of a perpendicular distance between the basis function plane and some of the targets.

For the multi-output neuron case we can have

$$\frac{\partial E}{\partial w_{ij}} = \phi_i^T(\mathbf{x})(\mathbf{y} - \mathbf{t}) = 0 \tag{4.21}$$

i.e. the learning process stops when the difference between $\mathbf{y}$ and $\mathbf{t}$ is perpendicular to the basis vectors. i.e. in 2 dimensions, when $\mathbf{y}$ is directly under $\mathbf{t}$.

4.8 Function Approximation

We can approximate any continuous function using sets of tanh() sigmoid functions. Since we can make the tanh() function arbitrarily close to a step function, we can see the outline of an approximate proof in Figure 4.9. Then

$$f(x) \approx f(x_0) + \sum_{j=0}^{N} (f(x_{i+1}) - f(x_i)) H(x - x_i) \tag{4.22}$$

where H() is the Heaviside function.

We can show the increasing accuracy of multi-layered perceptrons when we add more hidden neurons with the following small experiment: we use a noisy version of a simple trigonometric function and consider the set of points shown in Figure 8.4 which are drawn from $\sin(2\pi x)$ + noise. We train a multi-layered perceptron with linear output units and a tanh() nonlinearity in the hidden units. The results are shown in Figure 4.11.

Notice that in this case we were *required* to have biases on both the hidden neurons and the output neurons and so the nets in the Figure had 1, 3 and 5 hidden neurons *plus* a bias neuron in each case.

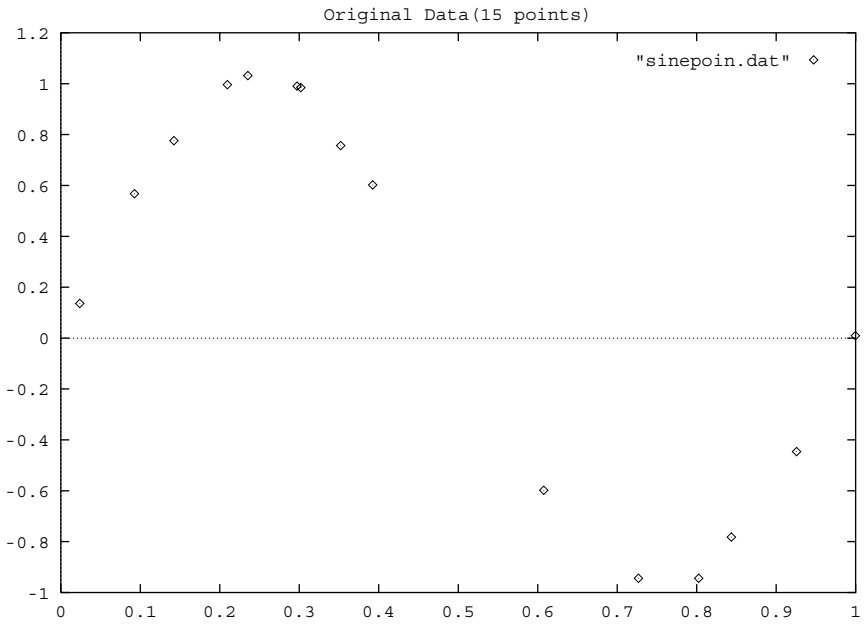

Figure 4.10: 15 data points drawn from a noisy version of $\sin(2\pi x)$.

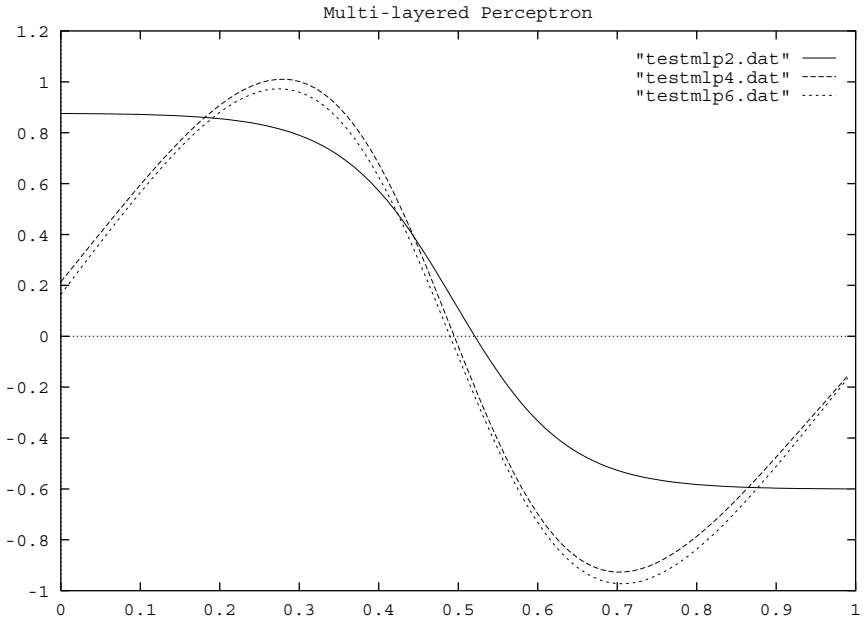

Figure 4.11: A comparison of the network convergence using multilayered perceptrons on the trigonometric data.

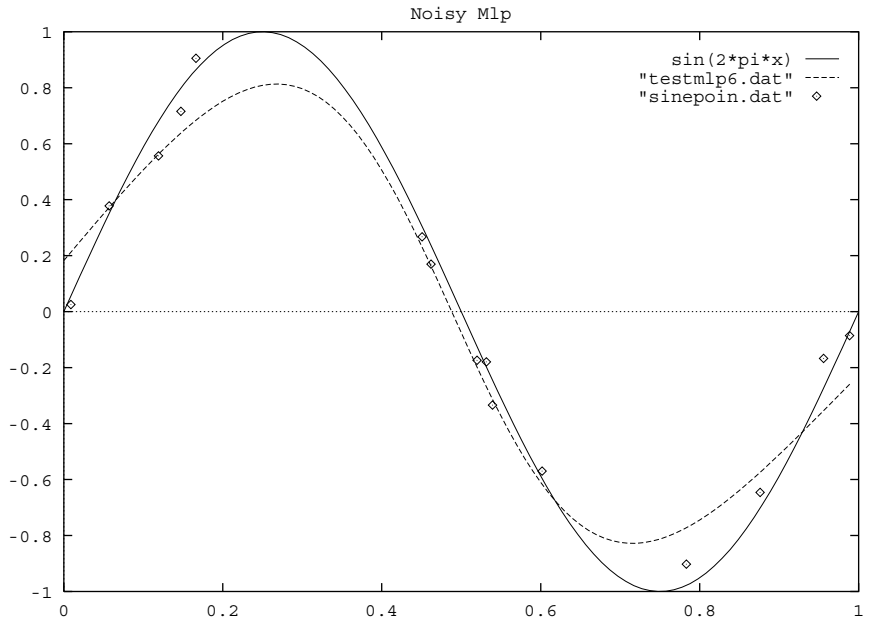

Figure 4.12: An MLP with 15 hidden neurons will also model the noise rather than the underlying signal.

Clearly the network with a single sigmoid hidden neuron is not capable of modelling the sine function adequately whereas the nets with more hidden neurons are more successful. This might suggest that our best tactic is simply to throw lots of simple processing power (i.e. lots of hidden neurons) at a problem. So now we show the MLP with 15 hidden neurons (plus the bias) also responding to the data in Figure 4.12. It can be seen that the MLP is responding to the noise as much as the underlying signal in the data. It has sufficient degrees of freedom (the network weights) to "memorise" the data; it must be made to work harder in order to extract the underlying signal from the data and the way to do this is to cut down on the number of hidden neurons in the network.

4.8.1 A Prediction Problem

We are going to use a network such as shown in Figure 4.13 to predict the outcome of the next day's trading on the Stock Market. Our inputs to the network correspond to the closing prices of day (t-1), day (t-2), day (t-3) etc and we wish to output the closing price on day t. We have arbitrarily chosen to input 5 days information here and only used a training set of 100 days. (We will not make our fortune out of this network).

For this problem we choose a network which has a nonlinearity (actually tanh()) at the hidden layer and linear output neurons. We chose to make the output neuron linear since we wish it to take values in the range e.g. 0 - 3800 or whatever value the stock market might achieve. It would have been possible to have sigmoids in the output layer but we would then have had to scale the target values accordingly so that the network outputs could approximate the target values.

In practice, we find it easiest to take out the trend information from the raw data before feeding it to a network and so we adopt a very simple procedure of subtracting the previous days' data from the current days' data. Similarly with the outputs and targets. So what we are actually predicting is whether the market will go up or down and by how much.

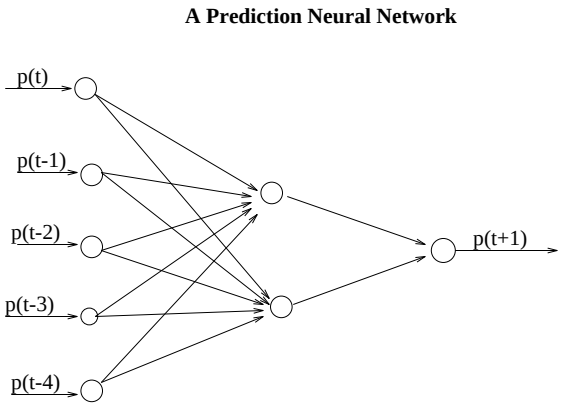

Figure 4.13: The inputs to the neural network comprise the last 5 day's closing prices and the network must predict the next day's closing prices.

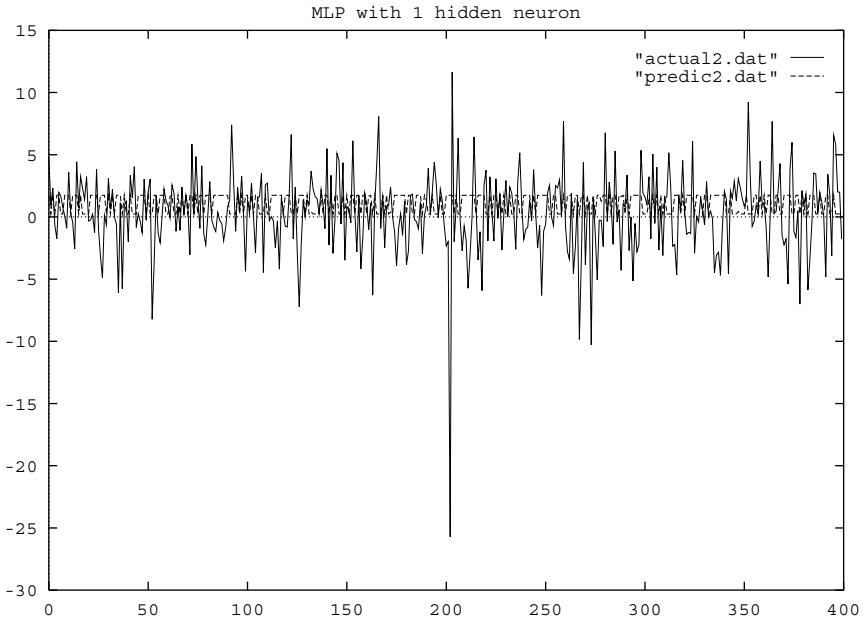

Figure 4.14: An MLP with a single non-linear hidden neuron is attempting to predict the closing price of the stock market on 100 days.

We show in Figure 4.14 the results of using a single hidden neuron (plus a bias term at the hidden layer) on the financial data. The results are clearly not good since the network is only finding out that in general the market is going up.

As we might expect the results are very much dependent on the size of network chosen and so we show in Figure 4.15 a network with 5 hidden neurons. The results are clearly much better but again we must take care about willy-nilly adding neurons since with 15 neurons the results (Figure 4.15) are very much worse (cf the sin() graph problem).

4.8.2 Practical Issues

You have been asked to create an artificial neural network to perform prediction on the FTSE index. What factors must you take into account? Some of the obvious ones are:

- Number of Inputs: we cannot specify in advance the number of days of inputs which are valuable in training the network optimally. So we would have to try different lengths of input vectors under the criteria below and find the best.
- Number of Hidden neurons: the smaller the better. Depends on the number of inputs etc.. We will use the least possible number of neurons which gives us a good predictor since if we add too many hidden neurons (== too many degrees of freedom) we will have an over-powerful neural network which will model the noise as well as any underlying series. Probably stick with a single hidden layer.
- Number of outputs is not an issue since we only wish one step look ahead.
- Output Activation Function will probably be linear since financial data is up to 4000 etc. Other possibility is to normalise the data between 0 and 1 and use logistic function.
- Learning rate: trial and error but almost certainly small (¡0.1). Maybe annealed to 0 during the course of the experiment which seems to give better accuracy.
- Split historical data into test set/training set. Division under a number of different regimes i.e. not just train on first 1000 samples, test on last 100. Stopping criterion: probably the least mean square error or least mean absolute error on the test set.
- Momentum: test a momentum term to see if we get better results.

4.9 Data Compression

We have stated that a network with no non-linear activation functions is no more use than a single layer linear network. There is one situation in which this is not quite true: when we have an auto-association network which has a hidden layer which is of smaller dimensionality than the input and output layers. In this specific case the hidden layer can be shown to form the optimal linear representation (the one which throws away least information) of the data. i.e. we are performing the optimal linear compression of the data.

We say that the network is performing a Principal Component Analysis. We will meet this in more detail in the next Chapter. For the time being consider a network which has been trained to autoassociate on a data set. Further let us consider the situation in which both we and a remote site have a copy of the network (or the data on which to train a network). Then if we wish to transmit that data with as few bits as possible losing as little information as possible we can feed the data to a network, extract the hidden neurons' activations and merely transmit them. When they are received at the remote site these activations can be inserted to the hidden nodes of *its* neural network and out pops the best (least-lossy) representation of the data.

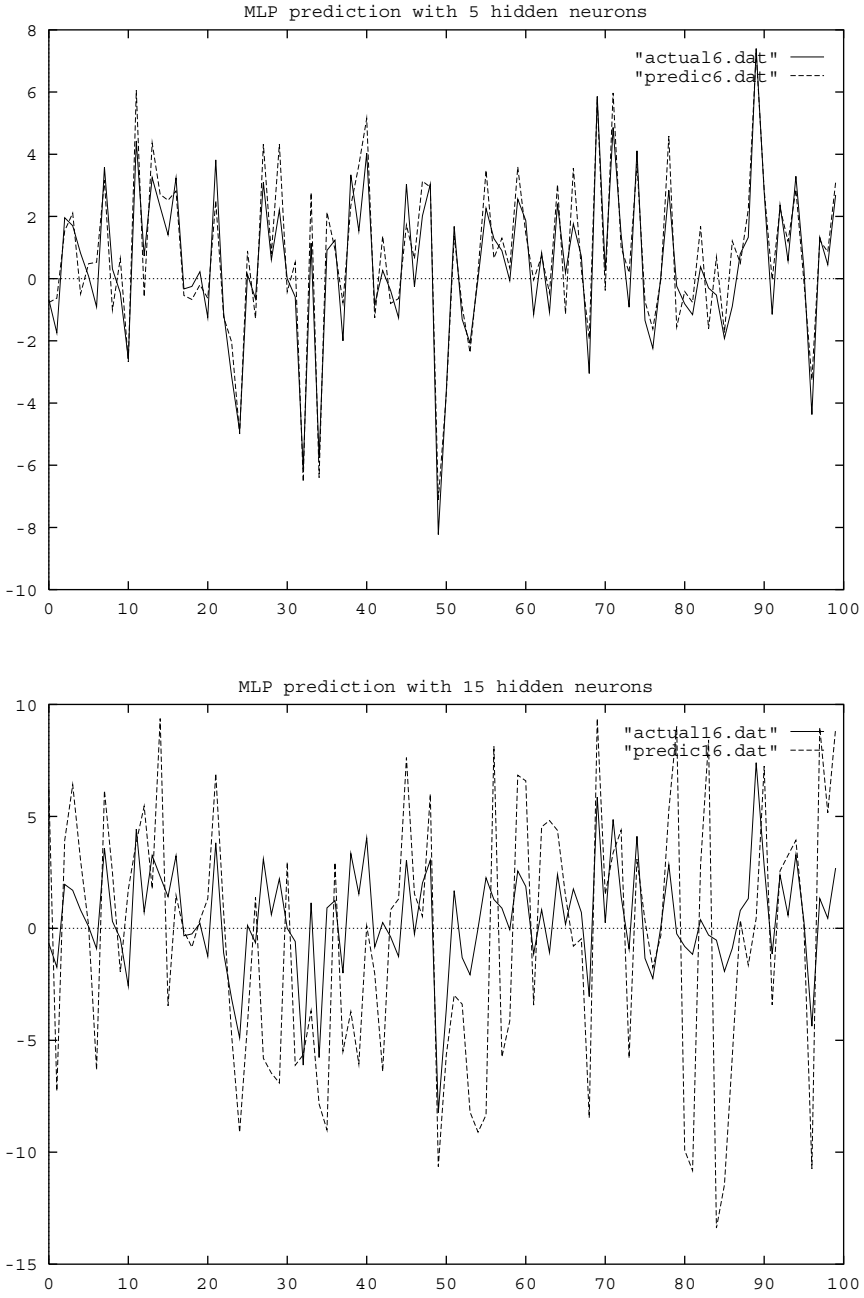

Figure 4.15: With 5 hidden neurons (plus a bias) the prediction properties of the network have vastly improved but with 15 hidden neurons the ntwork's powers of prediction have deteriorated again

4.10 Exercises

1. Consider the problem of data compression. We wish to create an 8-3-8 network which performs an encoding of the 8-dimensional input patterns into $\log_2(8) = 3-$dimensional hidden layer and which will then be decoded to an 8-dimensional output pattern. Use an autoassociation backpropagation network associating (1,0,0,0,0,0,0,0) with (1,0,0,0,0,0,0,0),
(0,1,0,0,0,0,0,0) with (0,1,0,0,0,0,0,0) etc.
Show a set of possible activations of the hidden values for each input pattern. (Objectives 4).

2. (Jagota,1995) Hidden units in multi-layer feedforward networks can be viewed as playing one or more of the following roles:

   (a) They serve as a set of basis functions

   (b) They serve as a means of transforming the input space into a more suitable space

   (c) Sigmoidal hidden units serve as approximations to linear threshold functions

   (d) Sigmoidal hidden units serve as feature detectors with soft boundaries

   Explain the benefits of the individual roles in helping us understand the theory and applications of multi-layer networks. Which roles help understand the theory and which roles ar insightful for which applications? (Objectives 1,3b, 3f).

3. Which of these two-layer feedforward networks is more powerful: one whose hidden units are sigmoidal or one whose hidden units are linear? The output units are linear in both cases. Explain your answer. (Objectives 1, 3b).