

Neural Networks Assignment 2

By Mike Blow

EASy MSc

10.03.04

House Price Prediction using a Multilayer Perceptron

Introduction

The problem is a real-world example to predict 8 house prices in the Boston area using a neural network. Approximately 500 training examples are given, consisting of 14 variables. The first 13 are various parameters that might affect the price and the 14th the price itself:

1	CRIM	per capita crime rate by town
2	ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
3	INDUS	proportion of non-retail business acres per town
4	CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5	NOX	nitric oxides concentration (parts per 10 million)
6	RM	average number of rooms per dwelling
7	AGE	proportion of owner-occupied units built prior to 1940
8	DIS	weighted distances to five Boston employment centres
9	RAD	index of accessibility to radial highways
10	TAX	full-value property-tax rate per \$10,000
11	PTRATIO	pupil-teacher ratio by town
12	B	$1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13	LSTAT	% lower status of the population
14	MEDV	Median value of owner-occupied homes in \$1000's

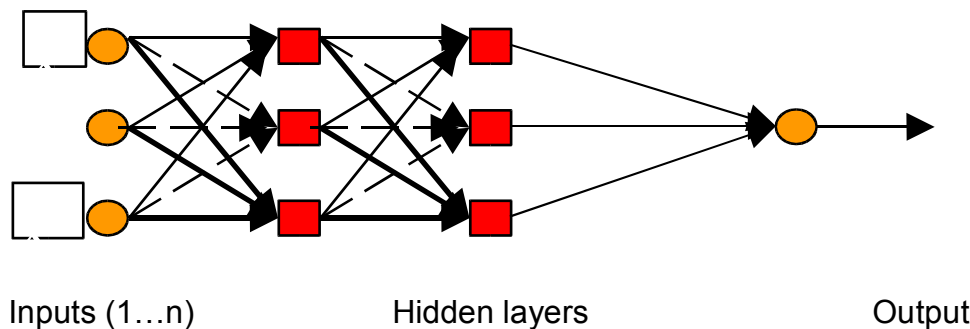
This is a sample of the actual data. Each row corresponds to one set of values:

```
0.02731 0.00 7.070 0 0.4690 6.4210 78.90 4.9671 2 242.0 17.80 396.90 9.14 21.60
0.02729 0.00 7.070 0 0.4690 7.1850 61.10 4.9671 2 242.0 17.80 392.83 4.03 34.70
0.03237 0.00 2.180 0 0.4580 6.9980 45.80 6.0622 3 222.0 18.70 394.63 2.94 33.40
0.06905 0.00 2.180 0 0.4580 7.1470 54.20 6.0622 3 222.0 18.70 396.90 5.33 36.20
0.02985 0.00 2.180 0 0.4580 6.4300 58.70 6.0622 3 222.0 18.70 394.12 5.21 28.70
```

The task is to train, validate and test a neural network and finally to use it to predict what the unknown house prices should be.

The Multilayer Perceptron

For this task I am using a multilayer perceptron (MLP). The MLP consists at minimum of one layer of neurons to which the inputs are applied; one layer that produces the output(s); and between them, one or more 'hidden layers':



Each connection between the layers has a variable weight which essentially alters the amount of effect that neuron's output has on the final result. It is by updating these weights that the MLP is trained to generalize the information in the data.

It is important that the transfer function of the hidden units is not a step function as this causes the MLP to behave like a single-layer perceptron, and as such cannot solve linearly non-separable problems i.e. ones that cannot be solved with a straight decision boundary. Instead a non-linear function is used, which allows the MLP to approximate any function. Example sigmoid transfer functions are:

$$y = 1/(1+\exp(-x)) : \text{outputs between } 0 \text{ and } 1$$

$$y = a \tanh(bx) : \text{outputs between } -1 \text{ and } 1$$

The transfer function of the output neuron can be sigmoid or linear depending on the required output.

Training an MLP

Initially the data is pre-processed if necessary and split into three sets:

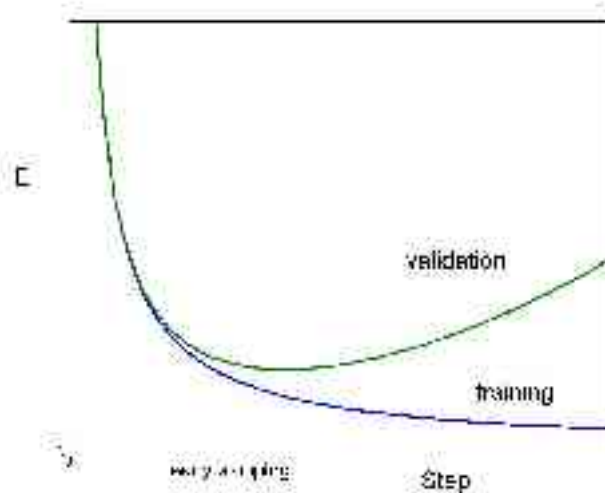
The **training set** is used to train the network and update the weights.

The **validation set** ensures that the network does not over-train and begin to fit only the training data – in other words it ensures good generalisation.

The **test set** is used once the network has been trained to assess how well it has learned the data.

The general training procedure for an MLP is as follows, where i is the input activation, h is the hidden layer activation and w is the weight associated with that connection:

1. Present a training data example to the input neurons.
 2. Calculate the hidden neuron inputs by summing the w^*i to each neuron.
 3. Calculate the hidden neuron activations by applying the transfer function.
 4. Calculate the value at the output neuron by summing w^*h to it.
 5. Evaluate the output layer error correction term by comparing the actual output with the target output, and the hidden layer error correction terms using the derivative of the transfer function.
 6. Update the output layer weights using output error * hidden activations.
 7. Update the hidden layer weights using hidden layer error * input activations.
8. Check the validation error at chosen epochs by applying a validation set through steps 1-4, and repeat steps 1-7 until the validation error starts to increase. This is the threshold between optimum generalisation and over-training. Beyond this 'early stopping' point the network will learn the training data better at the expense of general performance on other data (see figure below).



top line = validation error
bottom line = training error

9. Lastly present the network with a test set and calculate the error to give an idea of its accuracy.

Adding more hidden layers involves adding activation (using the outputs of the previous hidden layer) and error correction calculations (using the derivative of the transfer function) for each layer.

The Network Model

I have used an MLP with a one input corresponding to each data variable, a single hidden layer and one output neuron corresponding to the house price. The input and hidden layers also contained a bias node each feeding into the hidden and output layers respectively. The specific structure of the network was optimised using the procedures detailed below.

Weights were initialised using the formula

$$w = \text{rand}() * 2 * R - R$$

$$\text{where } R = \text{sqrt}(3/L)$$

and L is the number of inputs to the neuron the weight affects.

I used the non-linear tanh function on the hidden neurons:

$$y = a * \tanh(b * x)$$

and its derivative for updating the hidden layer weights:

$$y' = a * b - (b / a) * y^2$$

where $a = 1.715$ and $b = 2/3$. These numbers were suggested in the MLP overview and I did not vary them.

I designed the output neuron with a linear transfer function as I was concerned that a sigmoid would affect the outputs and make it harder to map the final answers to the example data.

Optimising the Network Structure

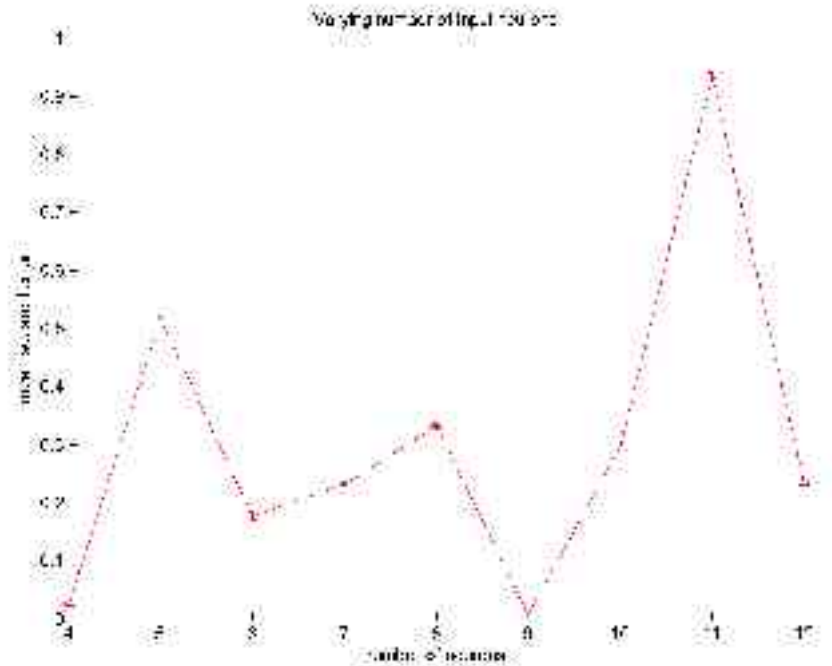
There are several variables affecting the performance of the network, and trials were undertaken to optimise them for accuracy using the pre-processed data. The procedure was to train the network on a subset of the training data (each time in a random order) and calculate the mean square error over a test set. Thirty results were averaged to give each value.

In all the experiments below the non-modified variables were constant at:

Number of input neurons: 7
Number of hidden neurons: 10
Learning rate: 0.0001
Size of data sets: 100 examples

Variable 1: Number of Input Neurons

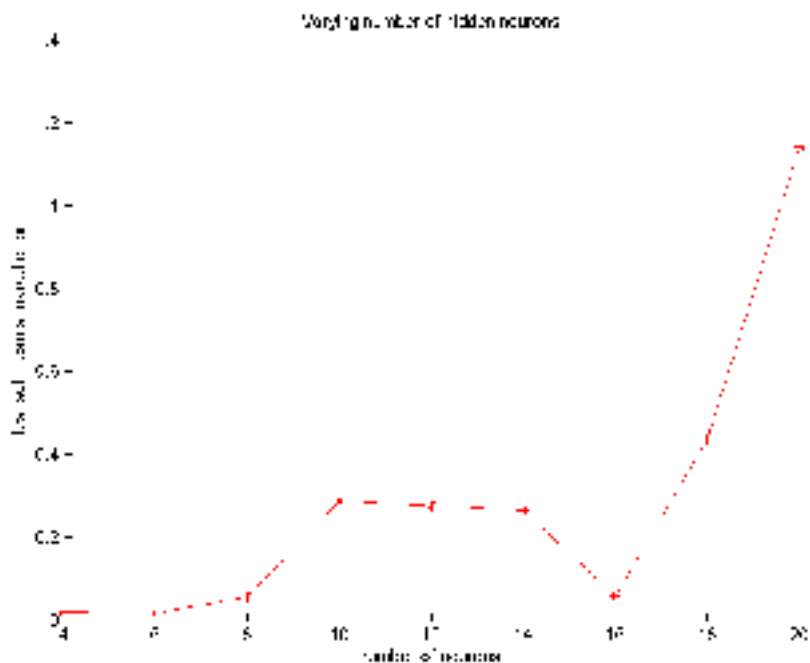
In this trial the number of input neurons was varied between 4 and 12.



This trial is not very conclusive. There is a definite low error point at 9 but the whole graph is quite noisy. Optimising the number of inputs can also be inferred from the processed data as I shall show in a later section.

Variable 2: Number of Hidden Neurons

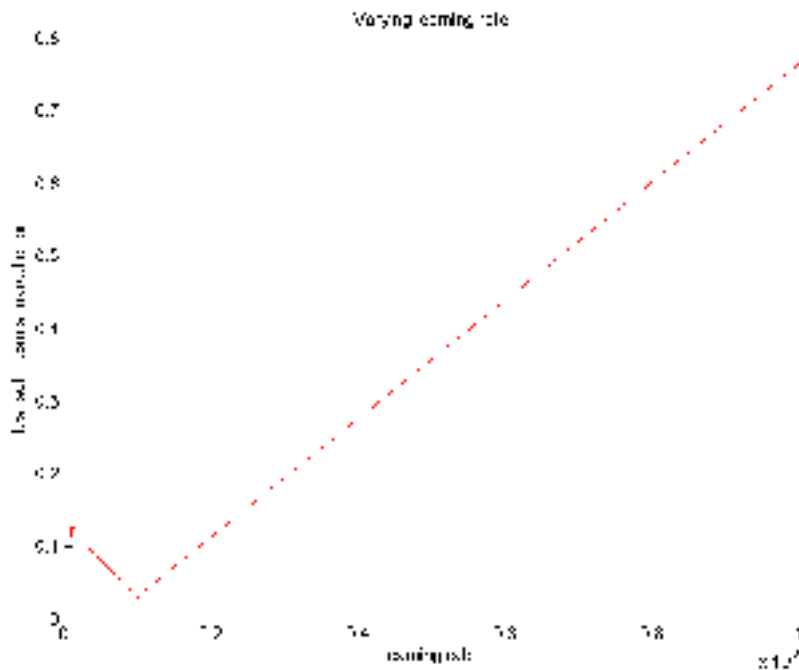
The number of hidden neurons was varied between 4 and 20 in steps of 2.



There is a general trend here towards greater error as the number of neurons increases. This is due to over-training (as discussed in a previous section), which tends to increase with network complexity.

Variable 3: Learning Rate

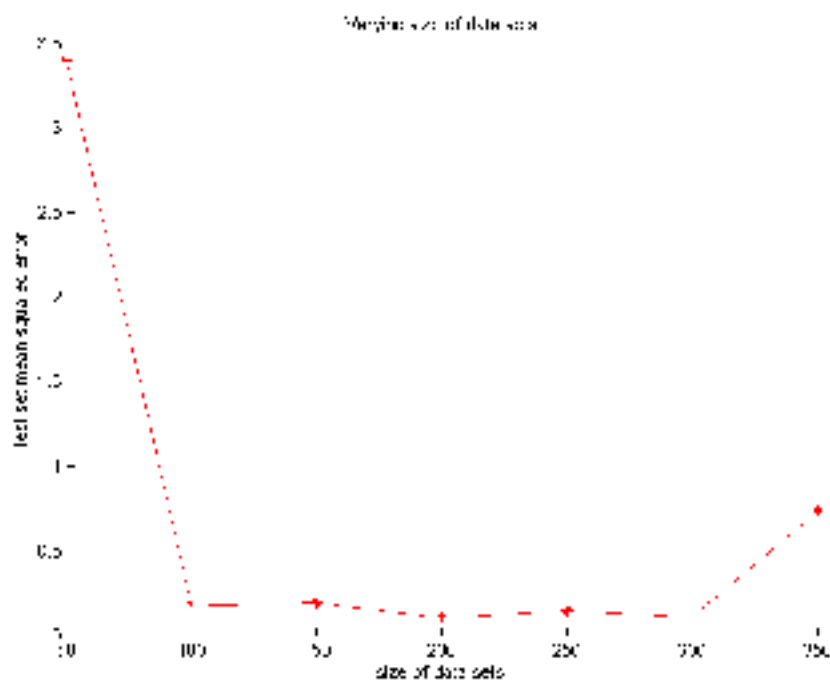
In this trial the learning was varied over three values: 0.001, 0.0001 and 0.00001.



The optimal result was provided by 0.0001. If the learning rate is too high the MLP will oscillate and may never find the minimum error. If it is too small the training will be very slow, and in the case of my code the error seemed hardly to reduce at all over many epochs.

Variable 4: **Size of the Data Sets**

In this trial the data set size was varied between 50 and 350.

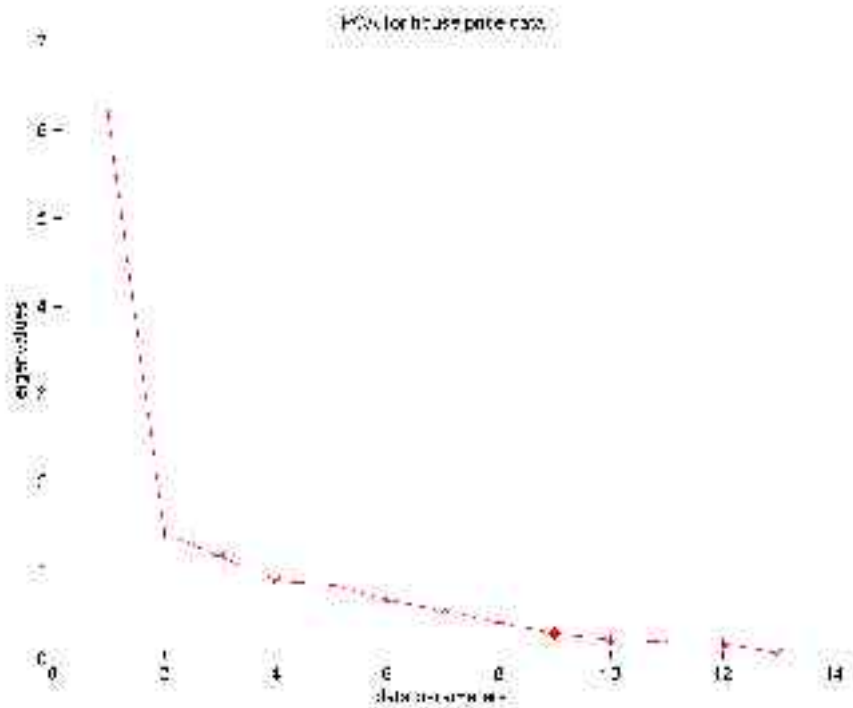


As can be seen the optimal size for the data set is around 200. Too small and there are not enough examples for the network to learn. If it is too large the training takes a lot longer and all 3 data sets comprise principally the same data which may lead to overtraining.

Data Processing

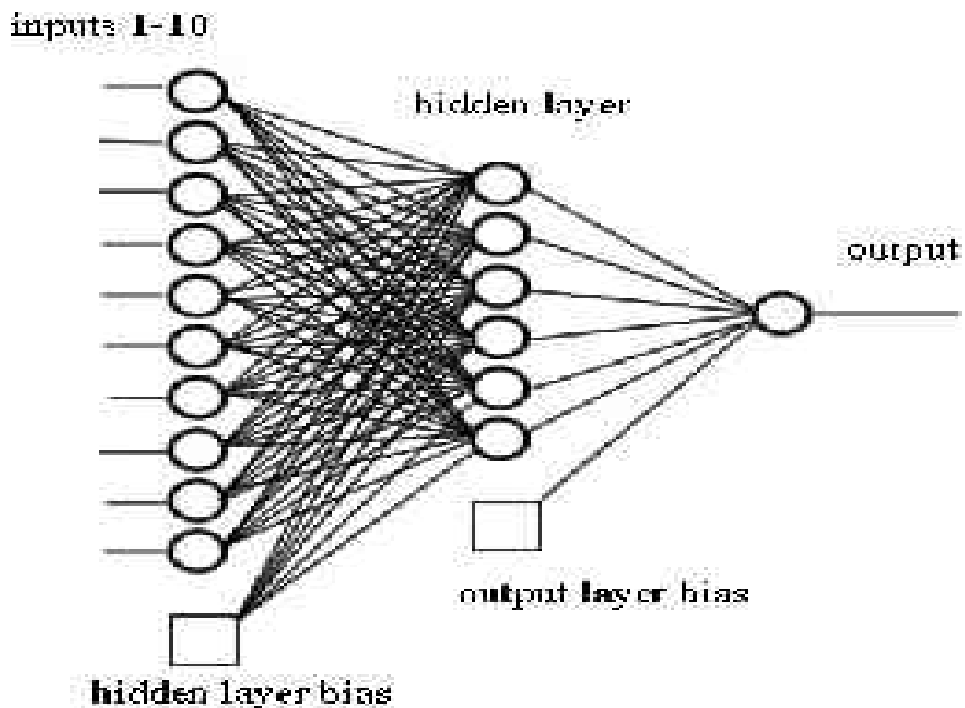
Before the data can be used to train the network it should be assessed as it may need preprocessing. I undertook three steps in preprocessing the data:

- 1) removing erroneous data. The house prices seemed to be capped at 50, and there were several entries where this was the case. I decided that if, as seems probable, the price in these cases was not a function of the input data but had been artificially capped, this would adversely affect the training of the network. Consequently I removed these entries.
- 2) Normalisation. In order to get the MLP to work I found I had to normalise the data to have a mean of 0 and standard deviation of 1, using MATLAB's `prestd` command. Without this preprocessing the network would produce NaN values in MATLAB. Normalisation ensures all data is at the same scale and produces positive and negative inputs centered around 0 which suited the tanh transfer function I chose to use (with a $-1 : 1$ range). After the net was trained I postprocessed the house price answers using `poststd` to recover the actual values.
- 3) PCA. Principal Component Analysis is used to reduce the number of inputs whilst retaining the information in the data. This is useful because of the 'curse of dimensionality' – the fact that the number of examples needed for training rises exponentially with the number of inputs. The variance over each input parameter is assessed and only those with a large variance are used to train the network by projecting them onto the principal component. An important point is that PCA is a tradeoff – the fewer inputs you use, the faster and less complex the network will be but less data means lower accuracy. The graph below shows the eigenvalues (corresponding to the relative amount of variance) for each of the input parameters over the whole data set



In this case you could obviously reduce the amount of data. If accuracy was the priority you might choose the first 10 inputs, but if speed was the issue the first 6 might contain enough data for acceptable training.

Using these results I optimised my network for accuracy at 10 input neurons and 6 hidden neurons. I set the learning rate to 0.0001 and used data sets of 200 examples each. Below is a picture of the final network topography.

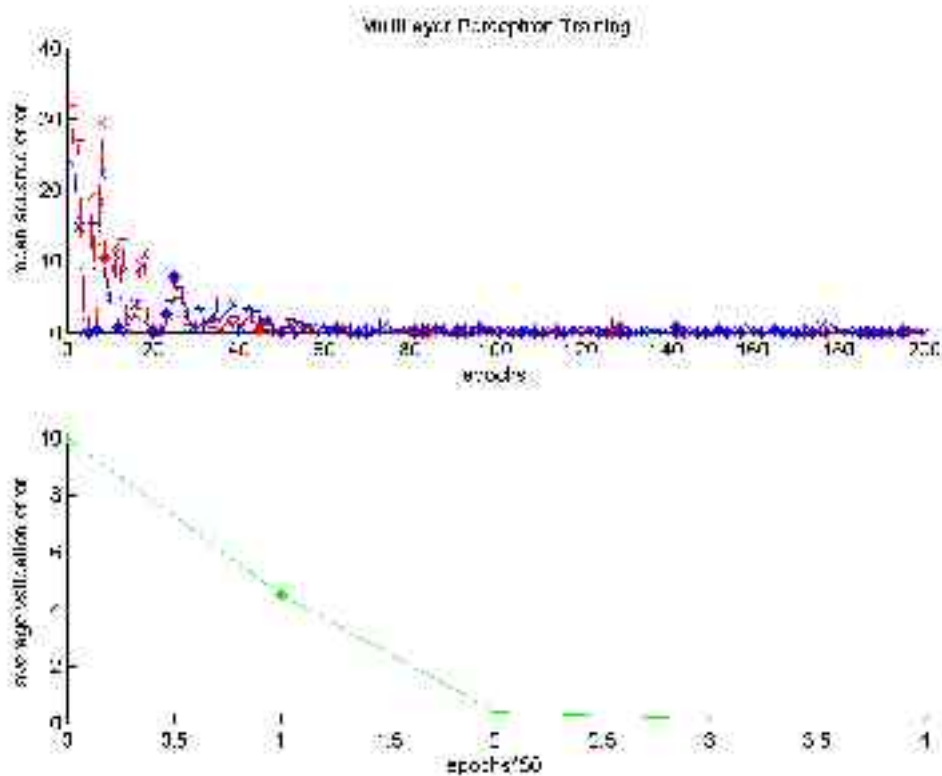


Training the Network

The network was optimised as detailed above and then trained using epochs of 200 randomly selected data examples. Validation error was averaged over 50 epochs and training was stopped when the error started to rise, indicating the onset of over-training.

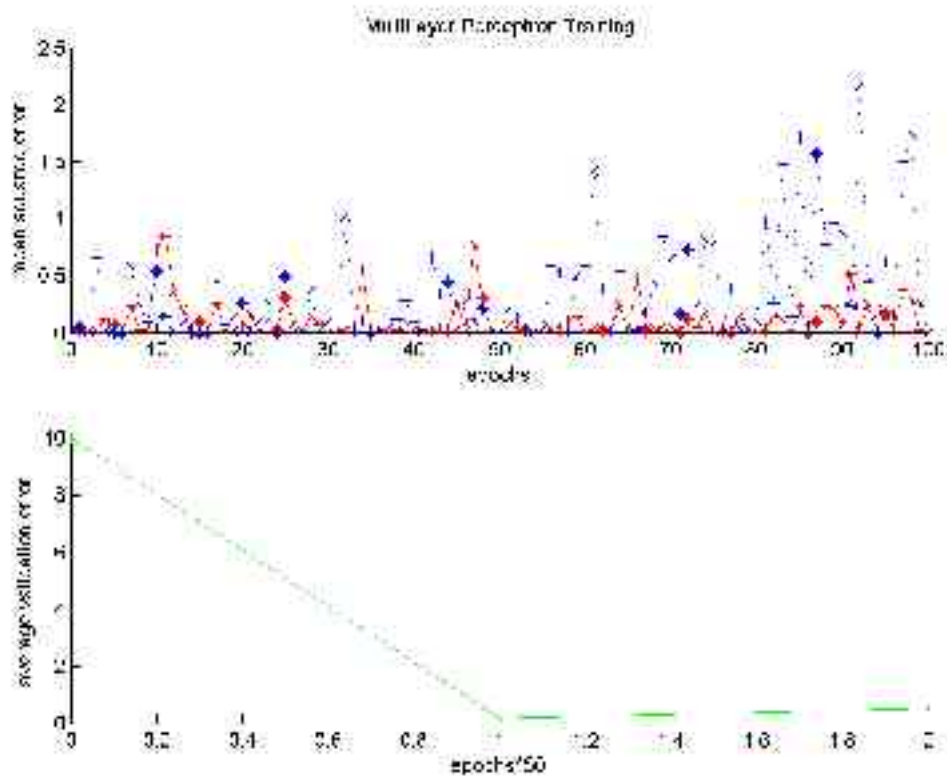
Below are examples of correct training, under training and over-training. In each the top graph shows the training error (red) and the validation error (blue) in each epoch. The lower graph shows the validation error averaged over 50 epochs:

Correct training:



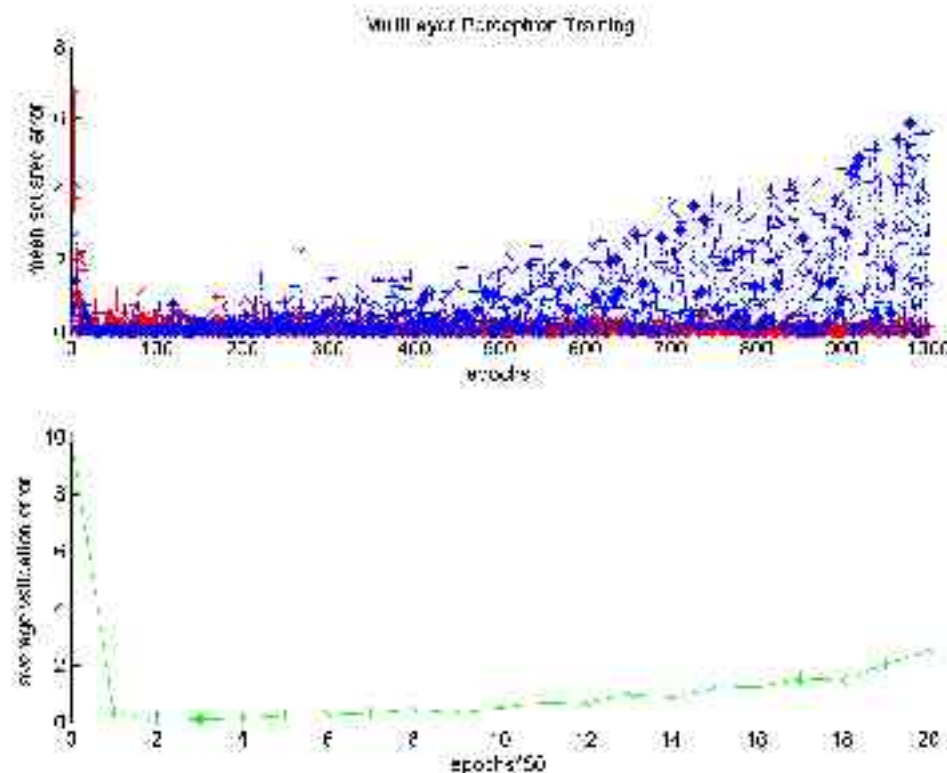
The error quickly reduces until epoch 200 where it has reached optimum generalisation.

Under Training:



In under training not enough data examples have been assessed to provide accurate generalisation. The network has started with a small training error so validation error has started to rise early and stopped the training prematurely.

Over Training:



In this example the network was run for 1000 epochs with the number of hidden neurons set to 12. It can clearly be seen that after 200 epochs the network starts to over-train. It is fitting too closely to the training data set information and as a consequence generalisation suffers and the validation error grows larger and larger.

Results

These are the house prices produced by the network after post processing, averaged over 100 runs:

house 1: 26.24
house 2: 10.99
house 3: 26.48
house 4: 17.89
house 5: 17.94
house 6: 25.44
house 7: 30.66
house 8: 22.63

Discussion

My MLP worked but in a 'noisy' way, with a large variance in the output errors each epoch. Nevertheless I have shown that a neural net can be trained on known data and used to predict unknown variables. I have justified my choice of network by showing how several of the parameters were optimised. I also have demonstrated the training algorithm, the danger of over training and the use of pre and postprocessing of the data.

It is apparent that there are so many variables in constructing a neural network that optimisation and data processing are essential parts of the process. There are many other enhancements that I did not have time to test; a second hidden layer; a more outlier-resistant error function and so on. From the hidden neuron optimisation trial, the fact that the number of hidden neurons should apparently be small for best performance would indicate that this is a relatively non-complex problem, which makes one wonder how much optimisation would be required for a more difficult task.