MLP Neural Network

Philip Crabtree
BSc Computer Science & Cybernetics

1.1 Abstract

This report will demonstrate the workings of a Multi-layer-perceptron Neural Network that is capable of implementing momentum. The network is capable of learning trends over a series of inputs so that once the network has been fully trained a prediction of future results can be made. In this report the MLP will be used to learn the XOR problem and learn trends over a set of sample statistics for the growth in traffic over 5 years. The network can be used to predict traffic flow in future years for up to 5 different road types.

Contents

2 Introduction

2.1 What is a Neural Network

Neural networks are often used in businesses such as insurance companies, stock brokers and the Met office to predict possible future results (temperature, share prices etc) based on previous data. An MLP neural network will take a set of sample data, known as an epoch, and calculate an output. This output will be compared against the expected output and, based on this difference, the network will adjust its weights (discussed later) to get the actual output closer to the expected output. After a few thousand epochs an average neural network will be expected to give outputs that closely represent the expected output. Once the user is satisfied the network has learned to a reasonable level (note that this level can be detected by the network its self by using a sum-squared-error method) the training will end. Now new inputs can be entered that represent possible future values and an accurate prediction can be output.

Neural networks are not only used for learning logic problems, such as the XOR problem discussed in this report, and making predictions. They can be used for any problem where by a set of sample data can be provided along with expected outputs. This includes optical-character-recognition, sound analysis and many other uses. However there are limits to the problems that can be solved. Prediction for the stock market can be done but due to the almost limitless number of variables that have an effect on the final output, the fact that the price of shares for one company will affect the price of another companies shares creating a loop, and the fact that share prices are affected by unpredictable world events such as terrorism, natural disasters etc. These all make prediction of future stock prices inaccurate.

The MLP neural network consists of three node types – input nodes, hidden nodes and output nodes. These are arranged into layers consisting of one type of node each. These nodes are connected together as per the diagram below. The input nodes will take an input variable to the system. The number of nodes required depends on the number of input variables. The XOR problem has two inputs (0,0 for example) and so two input nodes are required. These nodes are (in a fully connected network) linked to each of the hidden nodes. These links carry weights which are used when calculating the output of the network. The number of hidden nodes in a hidden layer depends upon the task one is performing. Trial and error will show how many nodes are most appropriate. These hidden nodes have another weight relating connected to them known as the bias. The bias value is generally set to 1 and there is a weight associated with this link as well. The final layer is the output layer. This will generally consist of only one node as generally you will only need one output. The output node has a link to each of the hidden nodes which also carry weights. The output node has a bias in the same way that a hidden node does.

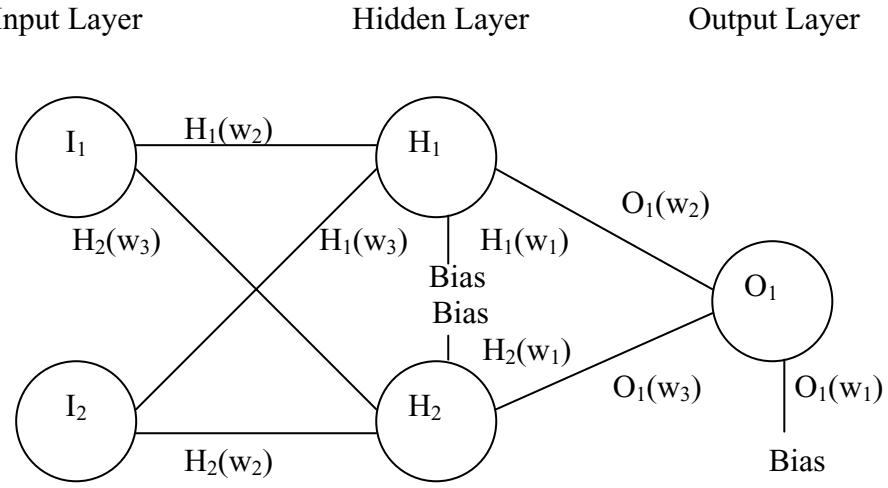

Figure 2.1. A typical MLP neural network as used to solve the XOR problem

2.2 How are the outputs calculated

The output of the network is calculated by firstly calculating the output from each node that precedes the output node (the inputs and hidden). As the input nodes only contain a single float value to output of this node is that float value. Calculating the output on the hidden layer becomes more complex. For the network above it can be calculated by

$$H_1 \text{ output} = \text{sum of (input} * \text{weight)}$$

Which is:

$$H_1 \text{ output} = (\text{bias} * H_1(w_1)) + (I_1 * H_1(w_2)) + (I_2 * H_1(w_3))$$

The same method is used to calculate the output from H_2.

This output can be linear or sigmoid. To have a sigmoid activated node the equation

$$1 / (1 + \exp(-\text{output}))$$

must be applied to the output. In the cases discussed in this report all the hidden nodes are sigmoid activated.

The output is calculated in the exact same way with the single exception that the inputs are the output from the hidden nodes. This forms the equation:

$$O_1 \text{ output} = (\text{bias} * O_1(w_1)) + (H_1 * O_1(w_2)) + (H_2 * O_1(w_3))$$

The output node can be either linear or sigmoid activated. The hard-

limiter function for sigmoid activation is the same as that shown above for a hidden node sigmoid activation. The linear activation is simply defined by ensuring that all values of output are between the values 0 and 1. The simple method of implementing this is:

a = input
b = output.

If a > 1 then b=1
If a < 0 then b = 0
If 0 < a < 1 then b = a;

This completes the forward pass of the network. If the network were fully trained then the output is expected to be approximately equal to the expected output. However, it is extremely unlikely that a network will be fully trained after just one forward pass. To correct the network the error in the output must be determined and the weights through out the network must be altered to bring the overall output closer to the expected output. The error in the network can be determined by taking the current output, multiplying it by one minus the current output and multiplying that by the desired output minus the current output. This forms the equation:

$$Delta = O_1 \text{ output} * (1 - O_1 \text{ output}) * (\text{Desired output} - O_1 \text{ output})$$

This however is only the error of the output node. To alter the weights of the whole network this has to be passed backwards (hence back propagation). It is worth noting that at this point that the delta produced by the equation above can be used to calculate the sum squared error of the network. This is a simple equation involving only squaring this output and adding this to the sum squared errors produced when using the other members of the training set.

The Delta of each hidden node can be defined by the equation:

Delta = ((HiddenNode output)(1 - (HiddenNode output)) * (Weight between HiddenNode and Output Node) * (OutputNode delta));*

All that remains now is to alter the weights according to the deltas found. The weights relating to the hidden nodes (the weights between the hidden node and the inputs (or bias). The general formula for this is as follows:

Weight change = learning rate input to HiddenNode(or bias) * HiddenNode delta*

This weight change, or dweight as it is often referred to as, is then added to the current weight. This is done for each weight (and hence input) relating to that node. The same is then done to any remaining hidden nodes in the network. The next stage is to alter the weights related to the output nodes using exactly the same formula as above with the exception of the fact that the input is the output from the hidden nodes.

That covers the forward pass and back propogation for the network. This

whole process is done for each set of training data. The whole process is then repeated again (including the alteration of training data) for either a set number of epochs or until a certain value of sum squared error is met. For example the above process is run four times to cover all of the training set (0,0 0,1 1,0 1,1). This is then repeated two thousand times for 2000 epochs. Therefore the whole process is actually called 4*2000 = 8000 times.

2.3 Improving the network

There are methods of improving the learning rate and hence the final sum squared error of the network. These methods include using momentum and simulated annealing.

Momentum is very useful in situations where the network falls into a local minimum. Under normal circumstances the network will think that it has reached the desired minimum (the so called global minimum). This is best explained by illustration.

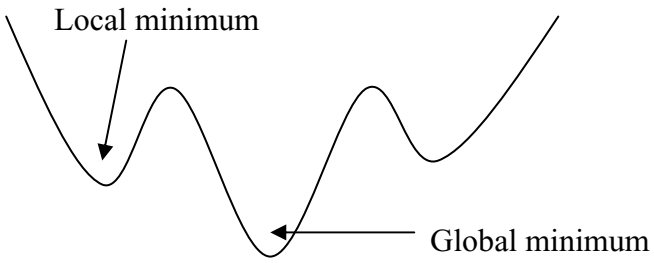

Figure 2.2. demonstrating local and global minimums

If a network falls into a local medium then under normal circumstances the network would not have the energy to push itself out of the 'valley' it is currently in. As shown in the diagram above, the network needs to push itself out of the current 'valley' and over the 'hill' into the global minimum 'valley'. The global minimum 'valley' is the deepest 'valley' that occurs in such a situation and so it is unlikely the network would be able to push itself out. So how does the network push out of the 'valley'?

Momentum works on the principal of taking some of the change in weight from the previous iteration of the back propogation, multiplying this by a user set variable (the momentum value) and adding this to the weight change that is being calculated for the current iteration.

*Dweights = (learning * Input (or bias) * Delta of current node) + (dweights from last iteration * momentum);*

An example of when momentum can be useful is ones where the sum squared error against epochs gives a graph such as

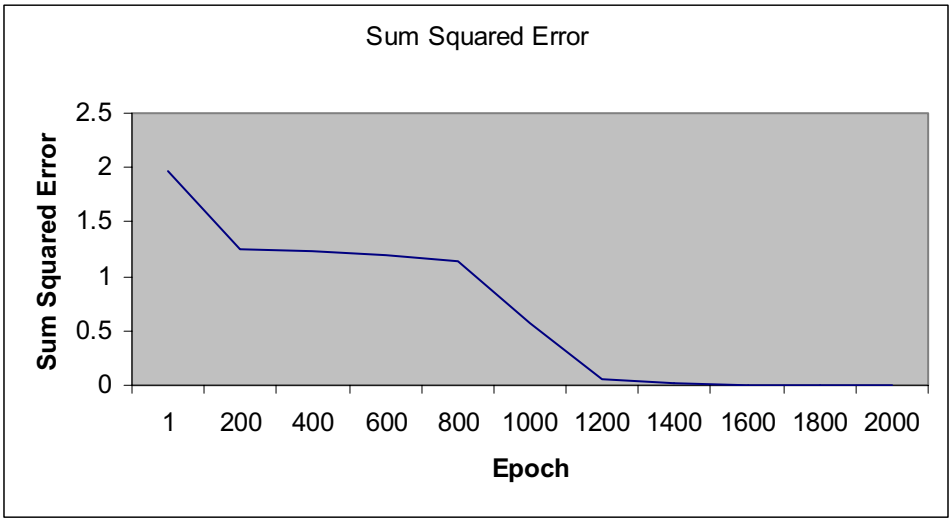

Figure 2.3. network in a local minimum

Between the 200th and 800th epoch it can be seen that there is very little alteration to the sum squared error in comparison to the change before and after this period. This suggests that there is a local minimum which the network has fallen into and is having difficulty escaping. Adding momentum to the network can, if set up correctly, totally remove this minimum and increase the learning rate by up to 600 epochs (the period where the network is stuck).

Simulated annealing is another popular method which works towards the same aim as momentum. This is discussed in greater detail during testing on both the XOR problem and the Traffic Flow Prediction problem.

3. Overview of the XOR problem

3.1 What is the XOR Problem?

The XOR problem is considered to be a 'Hard' problem when dealing with neural networks. This is because unlike an OR, or AND problem, the XOR problem is not linearly separable with just the input and output layers. To solve this problem a new layer can be introduced known as the hidden layer. This hidden layer effectively adds a new dimension to the problem allowing the XOR problem to be separated. These three separate layers are together known as a multi-layer-perceptron and they are capable of solving so called 'Hard' problems. Picton demonstrates the difference between the two types of problem using diagrams similar to those below.

(a) (b) **[1]**

Figure 3.1. Illustration of linear and non-linear problems. a) linear. b) non-linear

Imagine the triangles in the above diagram represent binary 1 and the circle represents binary 0. In figure (a) note how a straight line can be drawn to separate the 1's from the 0's. If such a line can be drawn then the problem is said to be linearly separable. These include the AND, NOT problems. Figure (b) can not have a line drawn in any way to dissect all the 1's from 0's. This problem is a non-linearly separable problem (such as the XOR problem).

The same delta rule that is used in single-layer-perceptrons can be used in a multi-layer-perceptron but only for the output layer. This is because we know what the expected output of the network should be but not what the expected output of the hidden nodes should be. To find the expected output a method known as back propagation is needed. Back propagation will pass the errors from the output layer back to the hidden layer to allow calculations to take place so that alterations to the appropriate weights can be made.

For an XOR network the nodes required are:

2 input nodes
n number of hidden nodes although 2 is adequate
1 output node

The weights used can either be random, specifically set by a user or Pictons initial weights can be used.

3.2 Network Performance

When the XOR problem is run using Picton's initial weights, a learning rate of 0.5, a momentum of 0, a sigmoid activated output node and hidden node and not using simulated annealing we get a learning curve as below

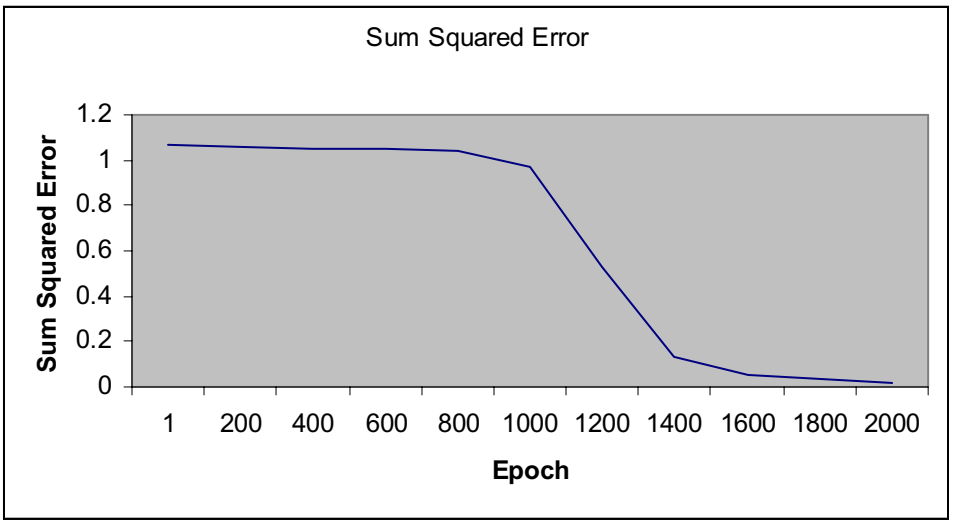

Figure 3.2. Sum squared error of the XOR problem against epoch number

It can be seen that until the 800th epoch there is very little change in the sum squared error. Between the 1000th and 1400th epoch the most dramatic changes occur before slowing down to give our final sum squared error a the 2000th epoch.

Perhaps it is possible to see a more dramatic change in sum squared error earlier on by using momentum or a larger learning rate. Hopefully, as the sum squared error is constantly decreasing, a low sum squared error can be achieved by the 2000th epoch. The following tests were done using the slightly varied settings of the above test.

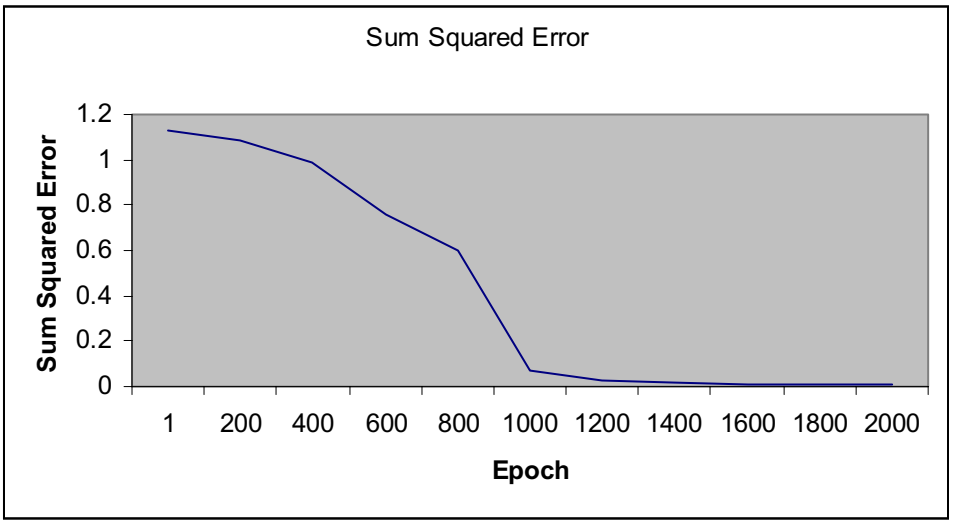

Figure 3.3. using learning rate of 0.9

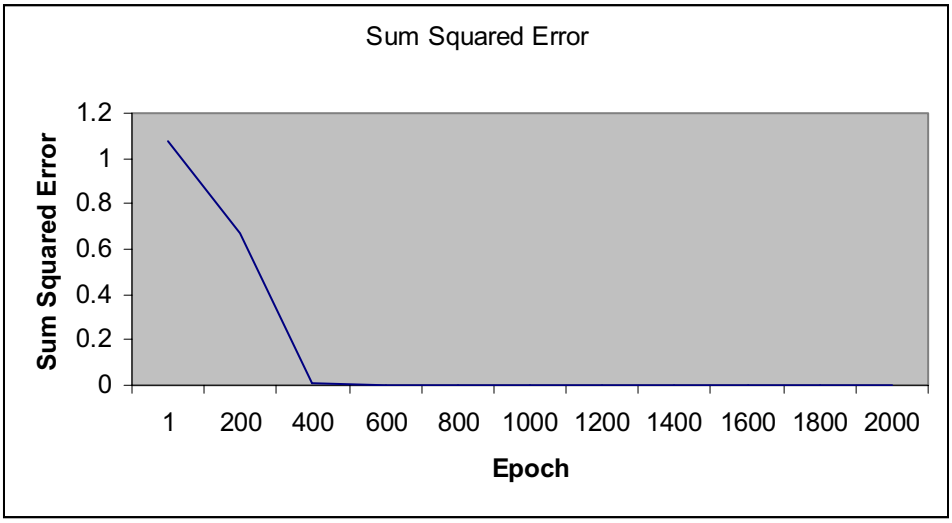

Figure 3.4. Using a learning rate of 0.5 and momentum of 0.9

From the above figures it is clear that altering the momentum and the learning rate have a dramatic improvement on the learning curve of the network. By increasing the momentum to 0.9 the learning rate is 5 times faster than that without momentum.

3.3 simulated Annealing

Another option that is available when attempting to speed up a networks learning abilities is the option of using a technique known as simulated annealing. This technique has been adopted from the method of heating metal, glass etc. The idea is that by heating up the material quickly and then allowing for a slow cooling the material can be formed easier due to the nature of the crystalline structures that form when cooling. When adapted to a neural network this technique is used in the same manner to that of momentum. If the network is stuck in a local minimum then energy will be required to 'jump' out of the current 'valley' and into the next. In some cases just momentum on its own is not enough for this. Simulated annealing provides the extra energy to do this (not that momentum is not required when using simulated annealing). The largest 'valley' is the global minimum, the minimum the network should settle at, for this reason it is important to not to be capable of jumping out of this 'valley' by providing to much energy.

Simulated annealing is applied in this program by using the formula:

$$Learning = e^{(original\ learning\ rate\ /\ (1+\ (epoch\ /\ Annealing\ constant)))}$$

This program will not assume that simulated annealing is desired. The user will be prompted to enter an annealing constant if required.

To see the effect simulated annealing has on the network one must use the same settings throughout the tests, changing only the annealing constant. The weights used will be Pictons initial weights, a learning rate of 0.5, a momentum of 0 and a sigmoid activated output.

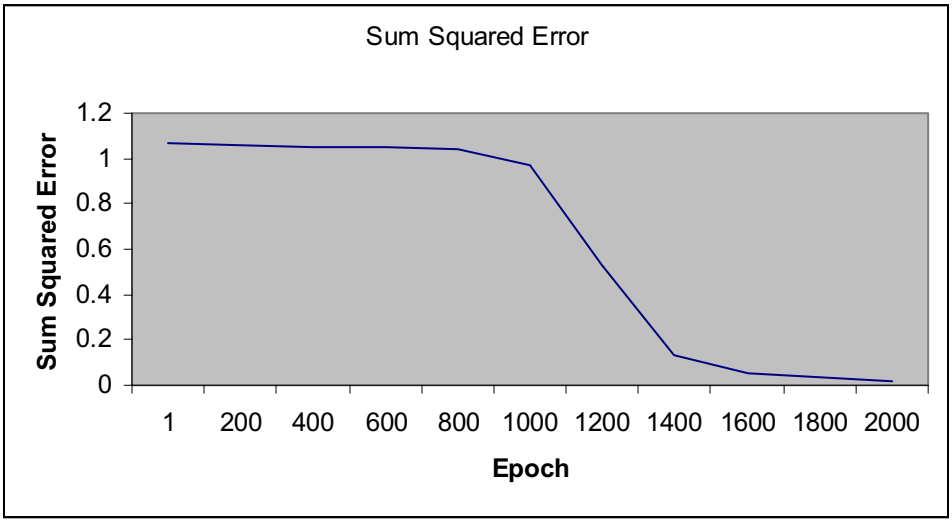

Figure 3.5. No simulated annealing

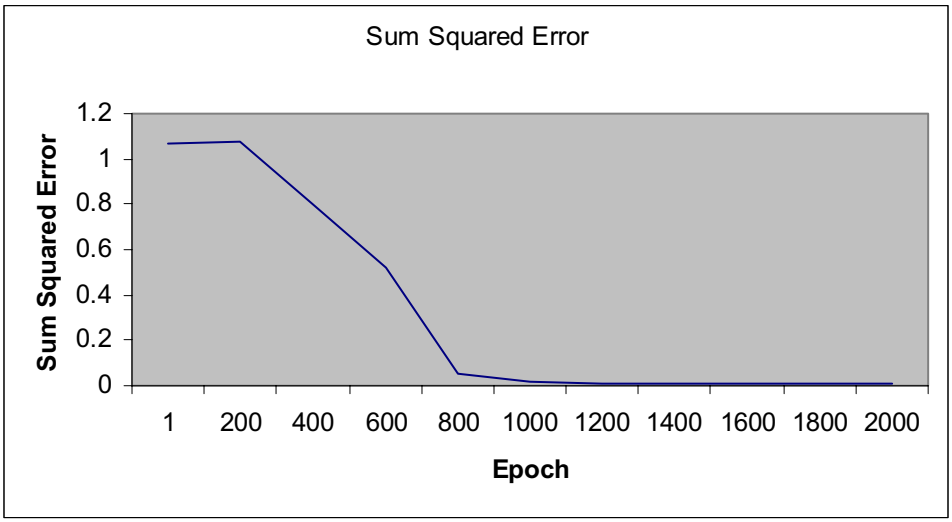

Figure 3.6. Annealing constant of 10.

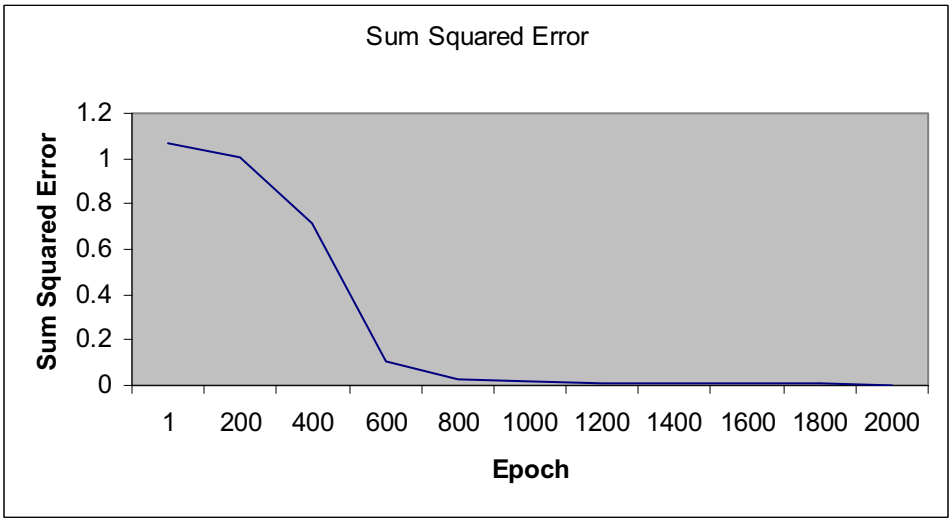

Figure 3.7. Annealing constant of 100

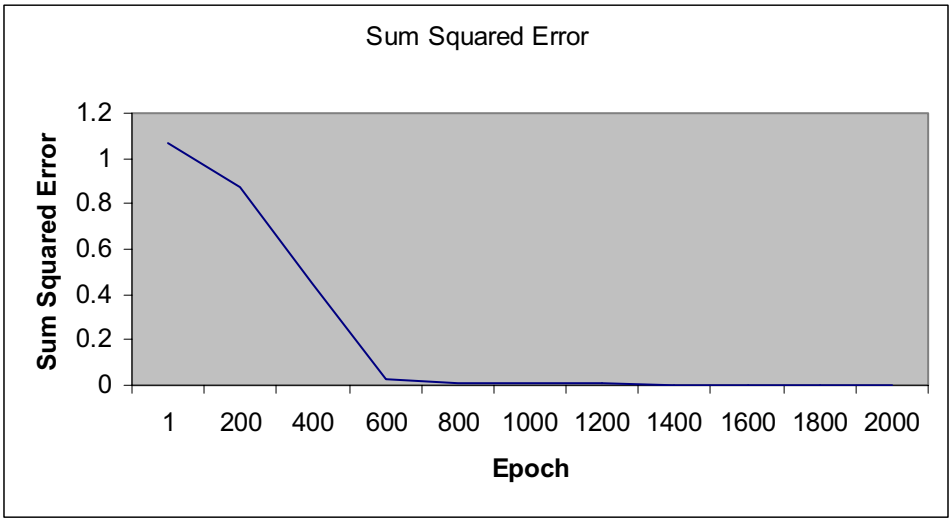

Figure 3.8. Annealing constant of 10000

From this point on the, raising the annealing constant makes little difference to the learning curve. Very slight alterations will be seen on the final sum squared error but these alterations are so small that in a network to solve a problem such as the XOR problem there is little need to extend to annealing constant past 100.

3.4 Improved Learning Curve

So far it has been seen what difference the momentum, the learning rate and simulated annealing have on the learning of the XOR network individually. If they are combined together is the result a very steep learning curve? If so then this can dramatically decrease processing time. If a network can learn after only two hundred epochs then it is a much more efficient network than those seen previously. The next test will be using the standard XOR network with Pictons initial weights, a learning rate of 0.6, a momentum of 0.6 and an annealing constant of 100.

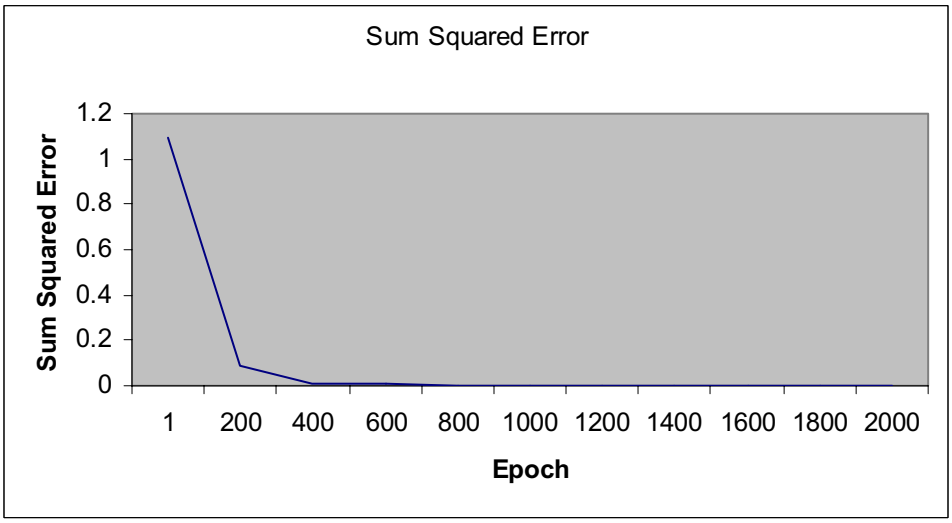

Figure 3.9. Combining a learning rate of 0.6, momentum of 0.6 and an annealing constant of 100

It can now be seen that by combining all the improved learning techniques

discussed above one is able to create a network that is capable of learning the XOR problem in just 1/7th of the time required by conventional methods.

3.5 Linear Activation

An area which has so far remained untouched is the ability to have a linear output node. Using the same values as used in principle tests above one can see how a linear output node will alter the networks performance when compared against a sigmoid activated output node.

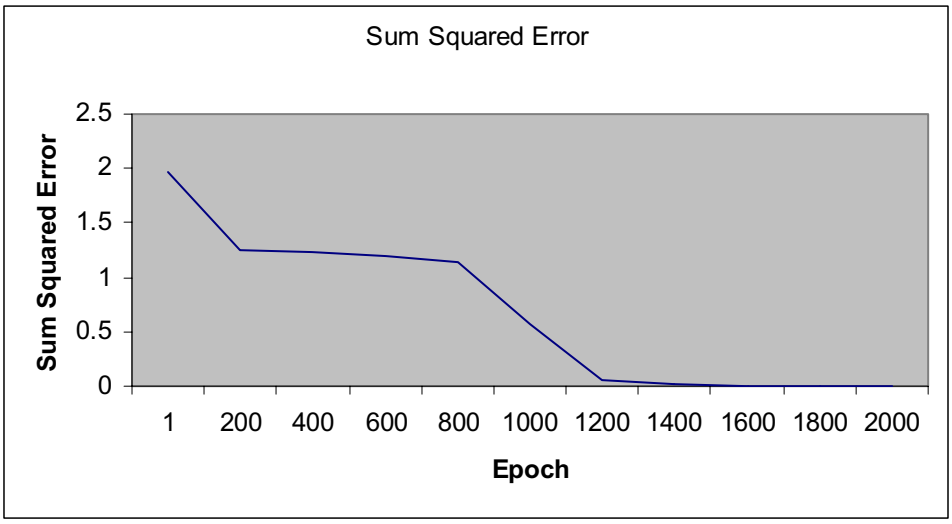

Figure 3.10. linear activate output node

This produces a result which is slightly better than that produced when using a sigmoid activated output node. It does however exhibit a behaviour that suggests the network was getting stuck in a local minimum between the 200th and 800th epoch. The introduction of momentum may improve the learning.

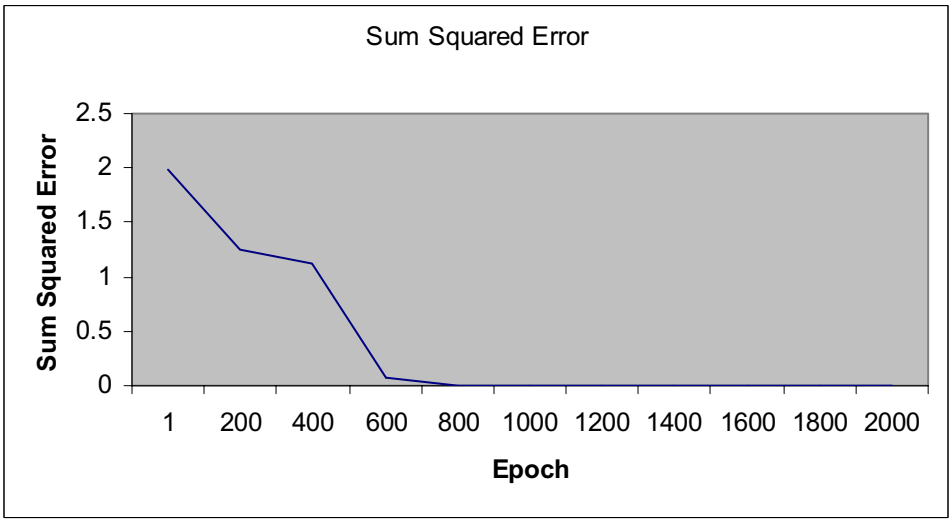

Figure 3.11. Linear output with momentum of 0.6

The introduction of momentum clearly reduces the time spent in a local

minimum as expected. This, in turn, decreases the over all time taken to learn to a sufficient standard by around 600epochs. Total removal of the local minimum (by increasing momentum or introducing simulated annealing) with further decrease the time taken to sufficiently learn by around another 200 epochs.

4. Description of the 'Traffic Flow Prediction' program and its Results

4.1 Design, applicability and potential

The second function of the software produced is one that will allow a user to predict the growth of traffic on 5 main road types. These are

- Motorways
- All major urban roads
- All major rural roads
- All minor roads
- All roads

For the inputs of the network these will be assigned the numbers 0.1 through 0.5 respectively at intervals of 0.1. Tests were carried out on assigning at intervals of 0.2 but it was found that the results were not improved so it has been decided that leaving spare input values will allow for future expansion.

So far only one input to the network has been covered. The other input is the year. This allows for a large range of test data (over a number of years) and for future prediction. The data used is data supplied by the department of transport for the years 1993 through to 1998. If the year 1998 is excluded from the training set so that it may be used for testing the networks prediction ability then the remaining data will from the year 1993 through to 1997. These will be assigned the input values 0.3 through to 0.7 respectively at intervals of 0.1. Note that experiments were carried out on using an integer value of 1993, 1994…etc but as the input is required to be pre-processed to provide a float value between –1 and 1 the value differences became too small to make an effective network. (i.e. 0.1993 and 0.1994 have a difference of only 0.0001 which is too small a difference to perform well).

When the program displays its inputs and outputs note that they have been multiplied to show an integer value. This is only used on the screen output, not in any of the calculations. Note that the output are representing billion vehicle killometers.

The inputs are generated by a simple algorithm within the code. The desired output are too random to produce an algorithm and too numerous to enter manually on each run. To solve this problem the desired outputs are read in from an MSDOS formatted text file called 'data.txt'. Each desired output is held in its own line and so the data can be pulled in a line at a time and stored in array. An alternative method would be to use delimiters such as commas to extract the relevant data. The main future advantage of reading in data from a file is the ease of updating the desired output should more data be added in the future. The desired output has been pre-processed in the text file so that all values are between the values 0 and 1. On output to the screen this is multiplied to show the data as it was originally supplied.

Whilst this program only uses two inputs it is possible to see how the number of inputs can be extended to improve prediction or to predict on a more specific level. For example, the years could be altered to represent days, the road type can represent time and a third input could be added for, say, the weather. This would be useful if

you narrowed down all the data to just one road (the M1 for example). The user would then be able to take some data over the period of a week or more for the training data and then use the network to predict the traffic flow along the M1 for any time of day, any day of the week depending upon the weather. This type of information can be useful for road construction/maintenance companies that which to work on the road. They would be able to find a time and day where the road would be suitably quiet for the works to be carried out. Note that this is the actual intention of the network but due to a lack of adequate data for such prediction it has been made to work on a more general level.

4.2 Results of Tests

4.3 How many Hidden Nodes?

One of the first steps that must be taken when making a prediction using a multi-layer-perceptron neural network is to find a suitable number of hidden nodes. The best way to do this is by trial and error. Obviously using random weights will, by its very nature, produced random results that may not correlate to the number of hidden nodes used. When a possible suitable number of hidden nodes has been found, several more tests with random numbers will be carried out on the settings to ensure that it is a suitable setup.

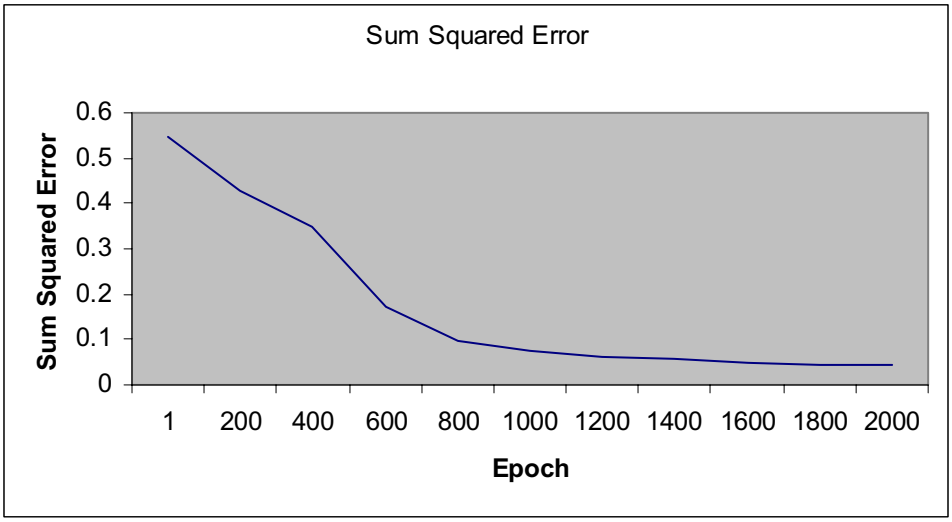

Figure 4.1. Using two hidden nodes.

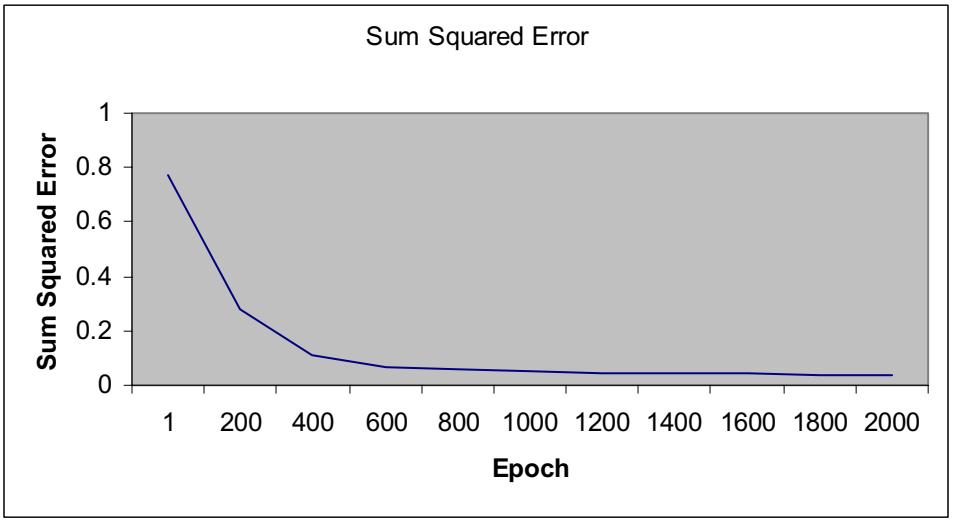

Figure 4.2. Using four hidden nodes.

The above two tests show quite a variation. It is clear however, that the test with four hidden nodes does not only provide a faster learning curve but it also provides a lower sum squared error at the end of 2000epochs. For a network such as this one, where the output values differ in fairly small amounts when compared to outputs such as those produced by the XOR problem, it is of high importance to get as little an error as possible so as to increase the validity of any predictions that will be made. At this point it is sensible to continue to increase the number of hidden nodes to see if any particular value out-performs others by a noticeable amount.

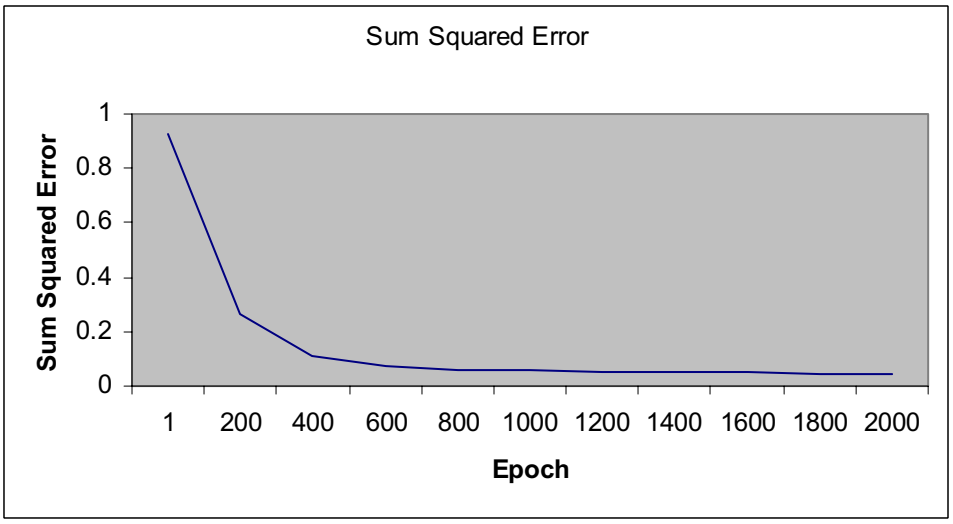

Figure 4.3. Using 6 hidden neurons.

This graph shows a lower performance than seen previously. These tests conclude that 4 hidden nodes, on average, produce a final sum squared error that is lower than results from networks with other values of hidden nodes.

4.4 Reducing the final sum squared error

The next phase is to ensure that the final sum squared error is suitably low such that an accurate prediction can be made. One way of achieving such a result is to train the network using the final weights from a previous run using the same settings. This will ensure that training is continued from where the previous network stopped due to the 2000 epoch limit. However, referring back to the above graph showing a network with four hidden nodes, it can be seen that all the learning is done within the first 600epochs and it is only fine tuning that occurs there onwards. This suggests that training for a further 2000 epochs would have little effect on the final sum squared error.

Looking back at the various techniques used to improve learning in the XOR problem it can be seen that the final sum squared error was reduced when the learning rate was increased, momentum was introduced and when simulated annealing was introduced. These factors should have a similar effect when dealing with this problem. Firstly experiments on the effect of the increase of the learning rate can be carried out. Note that for all the following tests the standard is a 4 hidden node network using random weights, a learning rate of 0.5, a momentum of 0 and a sigmoid activated output node unless said otherwise.

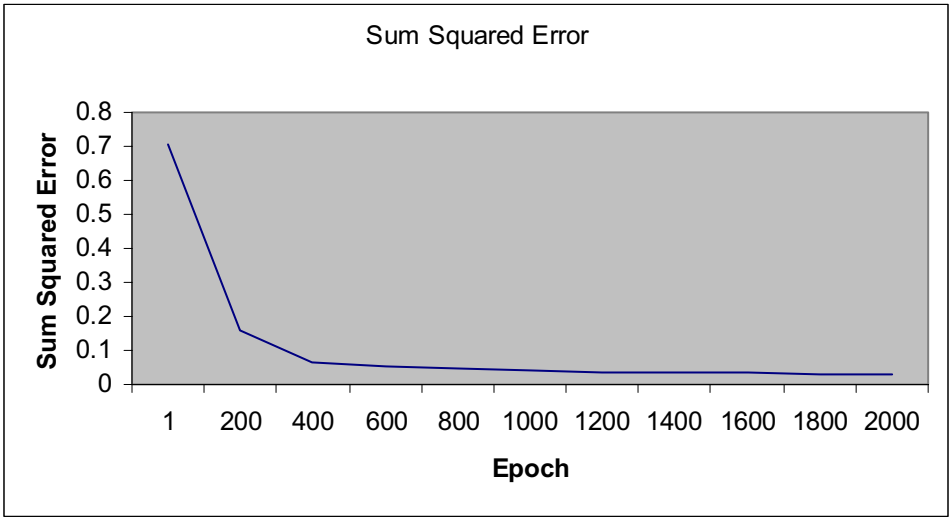

Figure 4.4. learning rate of 0.8

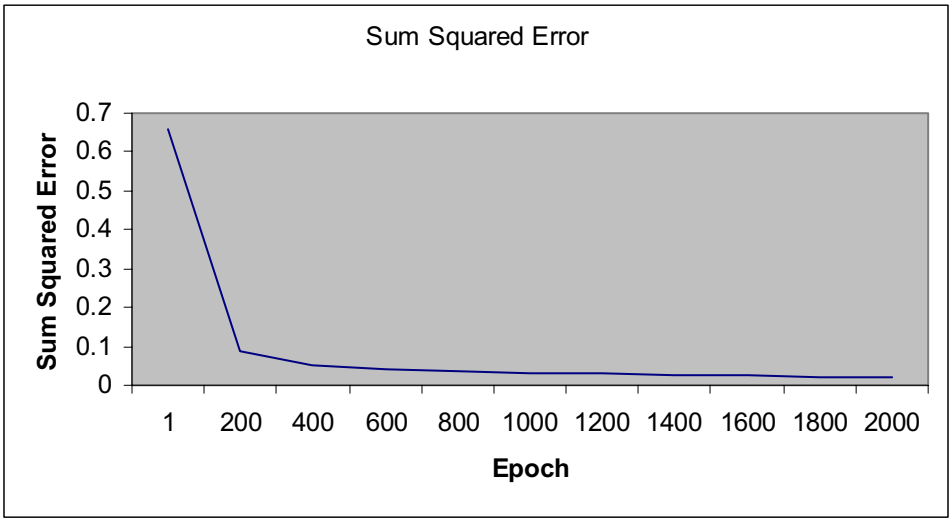

Figure 4.5. momentum of 0.6

As far, the change in momentum has a greater effect in reducing the sum squared error than a change in the learning rate. The next step would be to use simulated annealing. Previous experience (from the XOR problem) tells us that the optimum value for the annealing constant is 100 so this shall be used the first test.

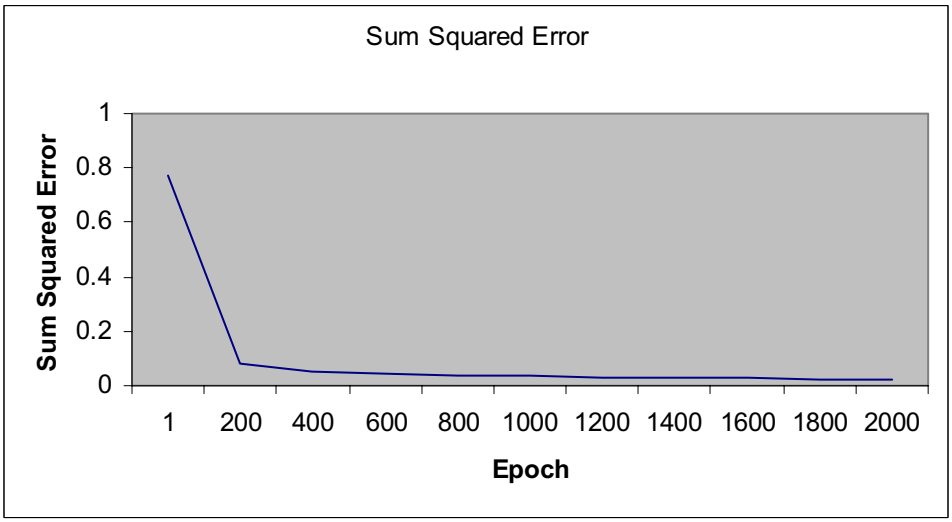

Figure 4.6. Simulated annealing constant of 100.

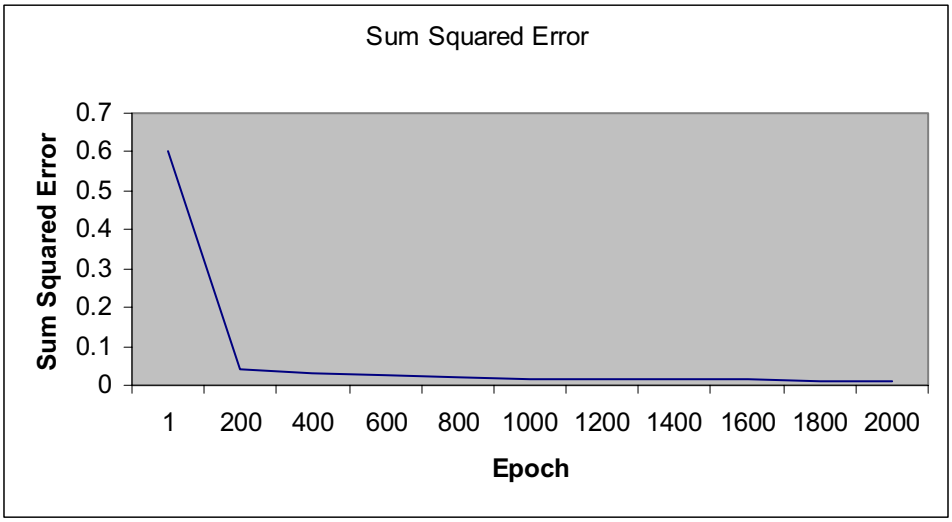

Figure 4.7. Learning rate of 0.8, momentum of 0.6 and a simulated annealing constant of 100.

The combination immediately above has, as far, given the best results in terms of the minimum sum squared error. This could be considered enough but if a comparison is drawn between the actual output and the desired output it can be seen how some values still differ greatly. The only way to reduce the sum squared error now is to take the final weights of the above test and plug them in to the network on a new run (note that the new run will keep the same settings as the previous run).

The final weights as produced by the above test are:

Table 4.1. Final weights from a network with LR=0.8, Momentum=0.6, Annealing constant=100

	Bias weight	W1	W2	W3	W4
Hidden Node 1	-5.79661	8.09753	0.00413181	-	-
Hidden Node 2	-1.00973	0.793696	-0.628611	-	-
Hidden Node 3	-1.02018	0.795653	-0.60704	-	-
Hidden Node 4	-6.24723	10.0421	0.14494	-	-
Output Node	-2.26118	5.1246	-0.743526	-0.717493	6.57331

As mentioned previously, it is expected that there will no drastic improvement on the final sum squared error, however, it is expected that the change will be enough to create a more accurate prediction than was possible previously. After plugging in the above values and using the same settings as were used to create the above table the network has trained as below: (note that this is effectively the same as continuing the training for 4000epochs on the original run.

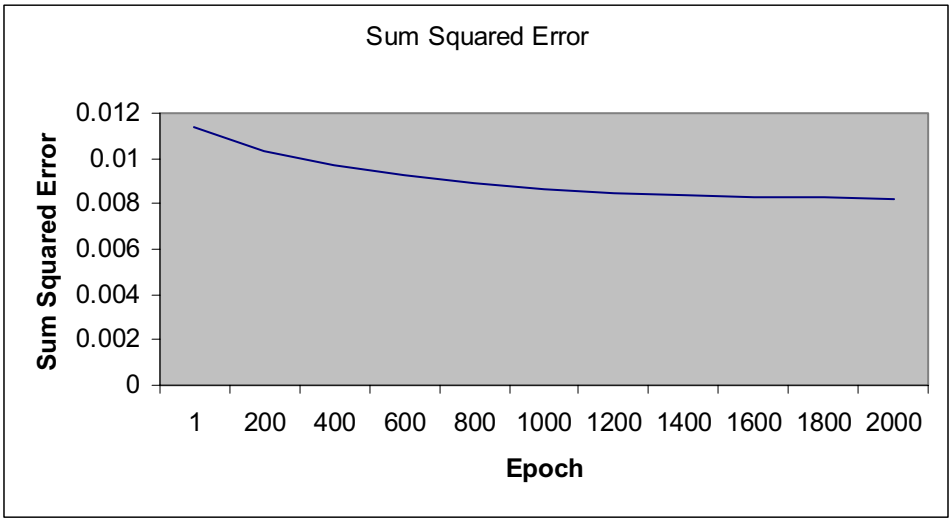

Figure 4.8. continuation from previous run

As expected, the changes are little but significant. These small changes will allow for a greater accuracy when predicting.

4.5 Final Predictions

Using this trained network a reasonable accurate prediction can be made for future years road traffic flow. A table follows with the results of predictions for the year 1999.

Table X. Predicted future road traffic flow for year 1999 in billion vehicle miles

Road Type	Predicted	Actual	Absolute Difference
1	95.2	88.7	6.5
2	132.1	130.7	1.4
3	79.5	81.5	2
4	171.9	166.6	5.3
5	512	466.5	45.5

5. Discussion of results

It is clear to see that the prediction is not 100 percent accurate. It does however give a figure that is in the right region but in some cases (such as for road type 5) the predicted flow is way off. This could be compensated for by further learning in the same manner as carried out earlier. The introduction of more training data would also help to increase the prediction accuracy as would introducing a third or fourth input to the system.

If this network were to be adapted to take results on a daily basis as discussed earlier then the number of inputs would increase as would the set of training data. If this data were then process to provide data for say, each month of the year then the network would be able to recognise trends with more ease (as the month would become an input).

An area that has not been explored in this report is the possibility of having several hidden layers each containing several hidden nodes. This is done using the same principle as the methods used here but should provide a higher level of accuracy due to the extra number of weights, each of which can be fine tuned to provide a low final sum squared error. Picton states that in theory any problem can be solved by having the correct number of layers with the correct number of nodes in each layer. [1]

Linear activated output nodes have not been used in this problem for the primary reason that the results are far too inadequate to consider for accurate prediction.

6. Conclusion

It has been shown that a multi-layer-perceptron is capable of learning the XOR problem with relative ease using standard back-propagation techniques. To extend this it has also been shown that, by introducing methods such as increased learning rates, momentum and simulated annealing one can increase the rate of learning by up to 7 times the original speed whilst maintaining (and in some cases reducing) the final sum squared error and hence increasing the accuracy of the final outputs.

This shows that it is not so much the problem one is trying to solve, but the method one uses to solve the problem. For example, in a situation where a fast learning rate was required, standard techniques would not be acceptable. This is when momentum and simulated annealing are introduced. In a case where speed is not so important one can quickly reduce the sum squared error to an acceptable level and then slowly reduce the sum squared error to produce a much more accurate system than previously. This is not so useful when dealing with the XOR problem although it is ideal for demonstrating the effectiveness of these techniques. The real use for these techniques are demonstrated in the second part of the program, the 'Traffic Flow Prediction' section.

When making a prediction, such as that made in the second part of this program, it is important that the sum squared error is as low as possible such that an accurate prediction can be made. It has been shown that an error similar the that produced by the XOR network whilst giving satisfactory results, if still too large for this type of prediction. By combining the techniques mentioned earlier it is possible to reduce the sum squared error down to less than 0.01. This still gives slightly inadequate results. There seems to be no real way to reduce the error even further without implementing further hidden layers and so one can draw the conclusion that the manner of prediction is some what random. In the real world one can not say for sure what will happen tomorrow but one can make an educated guess. This is all the neural network is attempting to do but based on very limited data.

7. References

Books
[1] Pictons ‘Perceptrons’, in ‘Introduction to Neural Networks, 1, Macmillan, 1994

8. Bibliography

Websites
http://www.generation5.org/content/2002/bp.asp
http://www.generation5.org/content/2001/xornet.asp
http://www.generation5.org/content/1999/perceptron.asp
http://www.msdn.microsoft.com/visualc/

Note that all websites were valid on access dates and unchanged through to time of writing (28/04/04)

Notes
Notes on neural networks supplied by Dr R Mitchell
Notes on C, C++ supplied by Dr V Ruiz

9. Appendix I

9.1 Design Methodology

Like all programs, this one starts with a basic algorithm as listed in the introduction of this report. An obvious translation of this would be to produce one function for each stage of the algorithm. This in theory works fine. However, once classes are introduced certain elements can be taken away from the functions and hard coded into the classes. These are things such as initialising arrays for weights, formulas for calculating the output, deltas etc. These will all be discussed in greater detail in the 'Descriptions of the Classes' section of this report.

The programs should all be accessible from one executable. This means implementing a menu structure for, initially, choosing which program to run and from then on which options you wish to use e.g. Pictons weights or random weights. The interface should always prompt the user to answer any questions such as "Do you wish to use simulate annealing?" and depending upon the answer, take a variable. The menu responses should be kept constant through out. As most menus will be in the form of a case statement it makes sense to expect an integer value (1 for the first option, 2 for the second etc). This can be done through out the program even when a question requires only a yes/no answer. Yes can be answered by entering 1, no by entering 0.

The programs that will be developed are ones that both require only two inputs. However, this does not mean to say that that is all that can be input. The second problem in particular has scope for future expansion and for such a reason the network is able to cope with any number of input nodes as is that case with hidden nodes. However, it is unlikely that there will ever be a need to more than one output. For this reason the network has been coded to only work with one output node. This was done for efficiency reasons.

Regardless of which program has been chosen to run (the XOR or the Traffic flow prediction) the same basic algorithm is required to run once information such as the number of input nodes, hidden nodes etc has been entered by the user. This suggests that there is one basic function for building the network and displaying any outputs. Variables can be carried through functions to allow each function to recognise if it is running for the XOR problem or the Traffic flow prediction problem.

To split the code up into logical chucks the main .cpp file will be split to create a function .cpp file where all the functions are carried out and a header file where all the classes and their associated functions are stored.

The network will be built on the principal that it is to be fully connected. As mentioned before, the number of input and hidden nodes can be set by the user at run time and so having a network that will not be fully connected requires the user to enter a large amount of data, concerning which nodes are connected to which, on each run. This could also leads to many errors due to the general way in which the formulas are calculated throughout the neural network.

The network can be split into 3 different basic class types, the input nodes, the output nodes and the hidden nodes. The output and hidden nodes are perfectly valid class types but having an input node (storing only one float variable) as a class seems a little over the top. For this reason the input will just be a float value that is stored else where. If there are no input node classes then it makes sense to store the weights that connect them to the hidden nodes in the hidden node class. This is also useful for when connecting the bias to the hidden node using a weight. It would be possible to have the bias set up as a user defined variable but as in 99% of all cases the bias is set to 1 it seems un worth while. So if all the weights between the hidden nodes and input nodes are stored within the hidden node class it only follows to store all the weights linking the hidden nodes to the output nodes in the output node class. Again the bias for the output node will be set to 1 within the code.

We now effectively have two basic node types plus a float value for each input. These nodes and inputs will need to be stored through out the program in a manner such that easy access for retrieval and storage is possible. The answer is to have another class which is designed to store arrays of input nodes (or floats), hidden nodes and output nodes. Once weights for example have been set within a hidden node, that hidden node and all the data relating to it needs to be stored in a safe place for later retrieval. The way to do this is to copy the information from the hidden node into an array of hidden nodes. The original node can now be destroyed to save on memory without loosing all the data. The node containing all the node arrays can be passed through functions as a normal variable could be making for easy access, retrieval and storage of data. This class will only need to be defined once at the very beginning of the program (depending upon user inputs etc) and destroyed only once at the end of the program. For more information about the classes see the ‘Class Design’ section of this report.

When dealing with the second problem, any inputs made by the user when making a prediction or outputs made by the system will be in the form of an integer. i.e. the year will be displayed as 3 instead of 0.3, the raod as 5 instead of 0.5 and the traffic flow 534.23 instead of 0.534.23. This is purely aesthetic and is only done when piping out to screen for in from the user. All calculations are performed in float values between 0 and 1.

10. Appendix II

10.1 Description of classes.

The neural network will essentially consist of three different node types. The input nodes, the hidden nodes and the output nodes. It is easy to see how each one of these nodes can be converted to an individual class containing the required data. Note that two output node classes are required. One for the sigmoid activated output node and the other for a linear activated output node.
Below are three tables listing the information that would be required by each basic node.

Table 10.1. Input node variables

Input Node	
Required Information	**Of type**
Inputs	Float value

From looking at the above table it is obvious that from a memory and processing point of view it is inefficient to have an Input node. Instead, input values can be stored as an array of floats.

Table 10.2. Hidden Node variables

Hidden Node	
Required Information	**Of type**
Inputs	Float values (as many as there are inputs)
Weights	Float values (as many as there are inputs + 1 for the bias)
Dweights	Float value (as many as there are inputs plus one for the bias)
Delta	Float value
Output	Float value.

Table 10.3. Output Node variables

Output Node	
Required Information	**Of type**
Inputs	Hidden Nodes (as many as there are hidden nodes)
Weights	Float values (as many as there are hidden nodes + 1 for the bias)
Dweights	Float value (as many as there are hidden nodes plus one for the bias)
Delta	Float value
Output	Float value.

The above two tables show almost 100% similarity. The difference lies in the inputs to each node. The Hidden Node requires float values whereas the Output Node requires a pointer to a Hidden Node. Due to this similarity it is possible to build a Base class from which the Hidden Node and the Output Node will be derived. As

there is no need to actually access a Base Node as an individual object we can specify all its functions to be pure virtual.

The table below defines what information is required by a Base Node.

Table 10.4. Bass node variables

Base Node	
Required Information	**Of type**
Weights	Float values (as many as there are hidden nodes + 1 for the bias)
Dweights	Float value (as many as there are hidden nodes plus one for the bias)
Delta	Float value
Output	Float value.

Notice how the inputs have not been included. This is because the type of input differs between the derived classes and so functions and variable must be defined in each class as required.

A decision must now be made about the base node. As it will be inherited by multiple classes we are left with a choice between making the class functions virtual or pure virtual. A virtual function is specified in the prototype of a function in the base class:

virtual void FunctionName(void)

These functions can be defined in the base class so the whole function is inherited. For example the following is valid

```
class BaseNode
{
        int a;
public:
        virtual void FunctionName(void);
}

void BaseNode::FuntionName(void)
{
        a=10;
};
```

Inherited classes can then access this function. A class called ‘Inherited’ can, provided it has been given the right access to the base class, use the function above using the following code.

```
Inherited ob;
ob.FunctionName();
```

A pure virtual function is the same in principle. It is defined by adding a ‘ =

0;' to the end of a function prototype. A pure virtual function only has the prototype defined in the bass class and not the actual function. The function its self is defined in the inherited classes. This is useful if different procedures are needed to set inherited variables in the inherited class. An example of a pure virtual function is shown below.

```
class BaseNode
{
        int a;
public:
        virtual void FunctionName(void) = 0;
}

class Inherited : public BaseNode
{
public:
        void FunctionName(void);
}

void Inherited::FuntionName(void)
{
        a=10;
};
```

The way to decide which type of virtual function is required is to look at what functions are needed within the inherited classes (the hidden node class and the output node class)

The setting of weights can be done for both the HiddenNode and OutputNode in the same manner with the exclusion of setting Pictons initial weights. This is because they are pre-defined weights and passing that variable into a class function that has been set up to work in both the hidden node and output node becomes messy. The setting of random weights and user set weights can however, be a standard function common to both classes.

The output node will require the ability to take different inputs for both the hidden node and the output node and perform separate equations up on them. This is the same for both calculating the delta and the dweights.

From what is stated above it seems that the only option available is to have virtual functions in the base node and define the actual functions in the inherited classes.

The BaseNode will contain a majority of the variables needed in the class structure. The only variables that are held in the Inherited classes are variables that are only used by that particular class (e.g. only the output node has a need for an array of pointers to the hidden nodes as an input). These variables can all be protected.

Table 10.5. variables set in the BaseNode

Type	Variable name
float	output
float	*weights
float	delta
float	*dweights

Note that variable with a name preceded by a * denotes a variable that will be assigned as a dynamic array in the Inherited class constructors.

The pure virtual function prototypes must now be decided. One must bare in mind that any virtual function prototype must be accessed using the same parameters as defined in the virtual function prototype.

To set an output and deltas one only requires a float value to run through the hard limiter function.

To set weights and dweights one will require a float (the actual weight) and an integer (where to store the float in the array). To retrieve the float value from the array one requires an integer value to specify where to look in the array for that value

To set random weights as initial weighs one requires two integers. One for the seed (to ensure different random numbers can be generated on each run) and the other to specify how many random numbers need to be generated. To set initial user weights only one integer is required to specify how many vales need to be requested and stored.

All the prototypes can be set as public within the BaseNode class.

Table 10.6. function prototypes in the BaseNode

virtual void set_output(float) = 0;
virtual float get_output(void) = 0;
virtual void set_weights(float a, int) = 0;
virtual float get_weights(int) = 0;
virtual void set_dweights(float, int) = 0;
virtual float get_dweights(int) = 0;
virtual void set_delta(float) = 0;
virtual float get_delta(void) = 0;
virtual void set_random(int, int) = 0;
virtual void set_user(int) = 0;

Once derived we will have a structure that can be illustrated as below:

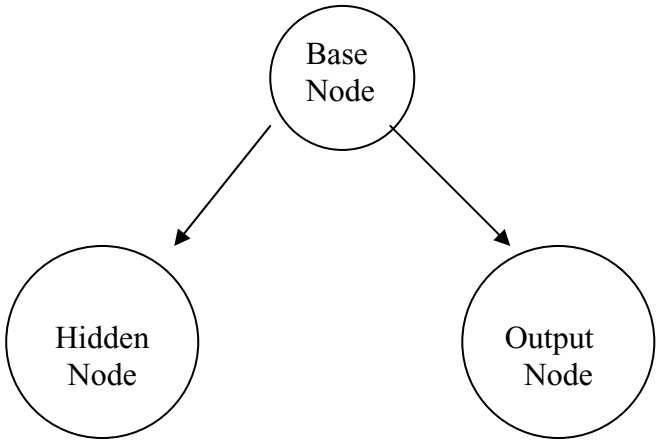

Figure 10.1. virtual class structure

The next step is to create prototypes for the inherited classes. (note that the inherited classes will be set using the line *class Inherited : public BaseNode* such that all public members are derived also. This leaves the variables as protected but they can be accessed through the derived classes.

The Hidden Node will require a set of floats as its inputs where as the Output node will require a set of pointers to the hidden nodes. These two variables can be defined as

*float *innodes*

*HiddenNode **innodes*

Respectivly. The HiddenNode function prototype will require an integer and a float value for the same purposes as those discussed when dealing with the weight functions. The OutputNode will need the same function prototype with the type float replaced with the type HiddenNode. Constructors and destructors must also be defined individually in each class. Both classes require only an integer to be passed to a constructor (the integer will represent the size of the arrays that need to be initialised). Destructors need no parameters. It is worth noting that by setting the destructors to 'virtual' within the derived classes one can dramatically reduce the memory leaks that may occur in the program.

Table 10.7. Class specific function prototypes

Hidden Node	Output node
HiddenNode(int);	OutputNode(int);
virtual ~HiddenNode(void);	virtual ~OutputNode(void);
void set_innodes(float, int);	void set_innodes(HiddenNode, int);

Both the hidden node and the output node here will be ones for sigmoid activation. However the specification requires that an Output Node exists that has linear activation. As this node will be that same as a sigmoid activation Output Node (in terms of required data) it can be derived from the Output Node. The structure now becomes:

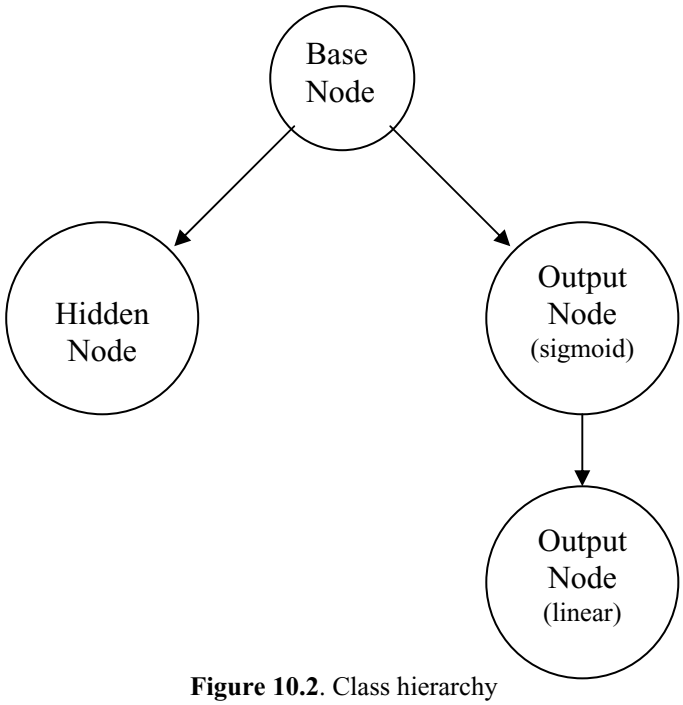

Figure 10.2. Class hierarchy

All access to the sigmoid activated output node can be done through the linear activated output node. If all functions are defined in the base output node (the sigmoid activated node) then all that is required to be defined in the inherited output node (the linear activated node) is the specific function for setting the output value when dealing with linear activation.

Note that now that the Output class has been derived we will refer to the Base OututNode as 'SigOutputNode' and the derived OutputNode as 'OutputNode'. Due to the derivation of the SigOutputNode there is no longer a need to have the constructor or destructor specified within the class. This is now specified in the OutputNode as shown below. The table below also includes the function prototype for the linear activation output node hard-limiter.

Table 10.8. function prototypes for derived Output node

OutputNode(int);
virtual ~OutputNode(void);
void set_linoutput(float);

Further to this structure another class is required. This further class is a class containing all the information about the Network such as the values for the inputs, the hidden nodes and the output nodes. This class is just a single class on its own. It is too set apart from the classes mentioned previously to be derived and there is no need for

a class to be derived from this new class. This new class will from this point on be known as the MLP class.

The MLP class will not only hold the nodes in the network but it will also hold other useful information such as the array of inputs and the array of desired outputs. The MLP will need its own constructor and destructor. The constructor will need a number of integers as inputs. These include the number of input nodes requires, the number of hidden nodes, the number of output nodes, the number of desired outputs and the number of individual inputs. Note that the inputs will be stored as a 1D array. This is because a simple algorithm can be run to extract the correct number of inputs or desired outputs to plug into equations. This method also allows for easy expansion of training sets.

Table 10.9. function prototypes for MLP class

MLP(int, int, int, int, int);
~MLP();
Void set_Input(float, int);
Float get_Input(int);
Void set_HNode(HiddenNode, int);
HiddenNode get_HNode(int);
Void set_ONode(OutputNode, int);
OutputNode get_ONode(int);
Void set_desired(bool);
Float get_desired(int);
Void set_Ins(bool);
Float get_Ins(int);
Void set_initialinputs(int, bool);

11. Appendix III

11.1 How the Classes were Tested

The first phase of testing simply involved building the BaseNode class and ensuring that all compiled correctly. As the functions for the BaseNode class are all pure virtual it was not possible to test them until the BaseNode class had been derived by another class. The HiddenNode class was then constructed and derived from the BaseNode class. Testing was then carried out to ensure that simple functions could be used to pass variables. E.g. the following was constructed.

```
class BaseNode
{
        int a;
public:
        virtual void FunctionName(void) = 0;
}

class Inherited : public BaseNode
{
public:
        void FunctionName(void);
}

void Inherited::FuntionName(void)
{
        a=10;
};
```

This ensured that the HiddenNode class is capable of setting the variable a to 10. The next step was to replace the function prototype and header with function that took an integer. Another function was then added to return an integer. The code now looked as follows:

```
class BaseNode
{
        int a;
public:
        virtual void FunctionName(int) = 0;
        virtual int FunctionName(void) = 0;
}

class Inherited : public BaseNode
{
public:
        void FunctionName(int);
        int FunctionName(void);
}
```

```
void Inherited::FuntionName(int x)
{
        a=x;
};

int Inherited::FunctionName(void)
{
        return(a);
};
```

This code ensured that variables could be passed into the BaseNode class and retrieved. This is a key feature in the classes required by the neural network program. An example of when this is needed is when a delta is needed to be passed.

This has proven that simple functions can be done with a similar setup to that above. The next step is to ensure that the class can cope with arrays such as those required when setting and retrieving the weights. Obviously an array is required. This is set by adding the line below into the private section of the BaseNode class.

```
float *weights
```

This will allow for a dynamic set up of weights. However, the size of the array will depend on the number of nodes in the network. What is required is a function that will allow an integer to be passed so that when a class is constructed the size of the array can be set. In c++ these are known as class constructors and they are capable of receiving variables such that arrays and so forth can be initialised. The following constructor must be added to the Inherited class in the public section.

```
Inherited(int);
```

From this prototype a function can be formed.

```
Inherited::Inherited(int a)
{
        weights =new float[a];
}
```

The size of the array of float values for the variable ‘weight’ can now be set. Setting and returning a float value from the array is done in a similar manner to that used when accessing a standard variable

```
void Inherited::set_weights(int a, float b)
{
        weights[a]=b;
}

float Inherited::get_weights(int a)
{
```

```
        return(weights[a]);
}
```

The above functions were tested to prove that float values can be passed in to the array and returned as expected. This presented no problems provided that one was not trying the set or return a value from a member of the array that does not exist. (e.g. trying the return the fourth element of the array when only 3 elements exist). At this point it is worth noting that setting an array size of three will result in there being elements 0, 1 and 2. Three values but starting at 0. This proves to be important when setting for returning values whilst in a loop.

The above has covered all likely events required in the HiddenNode class as this will only require passing of floats, integers etc. However, the OutputNode class will need to have a set of pointers to the HiddenNodes. This can be done by putting the line below in the private section of the OutputNode class:

```
HiddenNode **innodes;
```

As with the weights, the array of 'innodes' will need to be set using a constructor. In addition to this each element of the 'innodes' array will need to be initialised else memory allocation errors can occur. This is also done in the constructor and uses a simple for loop:

```
OutputNode::OutputNode(int a)
{
        int b;
        innodes = new HiddenNode*[a];
        for (b=0; b<a; b++)
        {
                innodes[b] = new HiddenNode(0);
        }

}
```

note *that the HiddenNode has parameters associated with its constructor.*

The tests then carried out involved constructing a HiddenNode and inputting a certain known value into each element of the weight array. This Output Node was then told to point to this hidden node. A further HiddenNode was constructed and more values were inserted into its array of weights. The output node was then told to store a pointer to this node in its innodes array aswell. By simply calling a function that would extract each weight from each hidden node as it was pointed to and display it on the screen showed that this method worked correctly. The next stage was to build the MLP class and have the appropriate nodes stored in there. This used exactly the same method as that described above and again no errors were experienced.

On these basic tests described in this chapter suitable classes and derived classes were built and tested. If any problems did occur then it was just a case of adding in a 'cout <<' statement to pipe whatever variable was being passed or read to screen so that one could ensure the correct values were being stored or read out. A

prime example of when this is necessary is when defining how the desired outputs are read in from the file “data.txt”. Error’s were occurring due to a incorrect constant being set. Whilst there were only 25 lines to read from the program was attempting to read 26 lines. By piping all the read in values to the screen one could easily see how the last ‘read in’ value was unexpected and so the problem was quickly solved.

12. Appendix IV

Table 12.1. Data as supplied by the Department of Transport

						Billion vehicle kilometres
	1993	**1994**	**1995**	**1996**	**1997**	**1998**
Motorways	**68.2**	**70.7**	**73.9**	**78.3**	**82.1**	**86.3**
All rural major roads	**113.3**	**116.5**	**119.5**	**123.5**	**126.7**	**129.1**
All urban major roads	**77.3**	**78.5**	**80.1**	**80.9**	**80.9**	**81.4**
All minor roads	**153.4**	**155.6**	**156.1**	**158.4**	**160.6**	**162.8**
All Roads	**412.2**	**421.5**	**429.7**	**441.1**	**450.3**	**459.6**

This data was then extracted a column at a time and pre processed such that all values were between 0 and 1. This gave a list suitable for reading in by the software. The list is as follows.

0.06819
0.11335
0.07732
0.15335
0.41221
0.07075
0.11654
0.07853
0.15561
0.42153
0.07390
0.11955
0.08007
0.15611
0.42972

0.07826
0.12351
0.08091
0.15840
0.44107
0.08210
0.12666
0.08089
0.16064
0.45030

13. Appendix V

13.1 The Code – the class definitions and associated functions

* Code Removed *
Please contact my directly through iphil.co.uk if you would like to see the source code