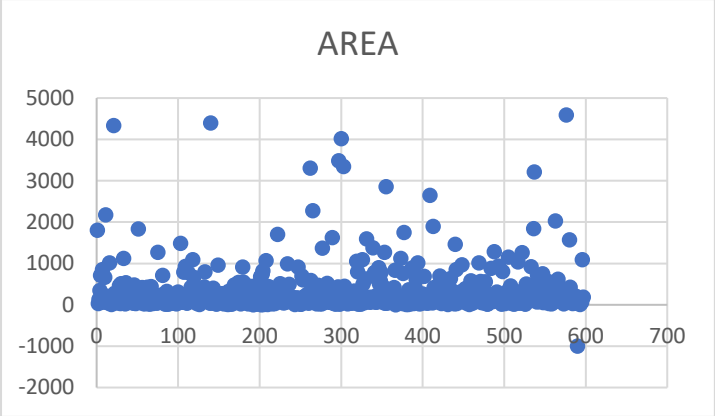
Part 1 – Data Pre-Processing

Before beginning the actual implementation of the Multi-layer Perceptron algorithm, I believed that data pre-processing should take precedence. So, I began by cleansing the data first, for this I needed to understand the data, so that I would know what seemed appropriate and what the possible outliners would be. This is important as data that contains outliers or errors, will produce incorrect trends and correlations, which means that when the neural network is provided new predictors, its prediction will be completely incorrect.

So, I began by comparing each predictor against the predictand using excel, this was used as it very producing graphs to show correlations and trends. I first compared the Area agaisnt the Index flood, doing this showed me that there was value that came out as -999, this was clearly an outlier as an area cannot negative. This fact was true for all predictors, so I knew I needed to look out for negative numbers. I created graphs for each predictor and discovered that they too contained negative numbers.



I decided to create an algorithm to perform the cleanse – this function was named the 'prep_data' function. I decided to create an automatic cleansing function, so that the whole system could be repurposed to be used on a different dataset without having to look at it properly. To do this, I brought dataset in as a text file, splitting each line into row of data to be appended into an array called the 'dataset'. Then for each row in the array I performed calculations in order to look out for negative numbers, if found the 'prep_data' function removed the entire row. When I first ran the code to cleanse the data, I was unable to process the input due to the function not being to parse through strings, which meant the data had more errors and outliers. So, I had a check for non-floats /integers into the system, if any were found once again the row was completely deleted. This meant that the 'prep_data' function was able to return a cleansed dataset, with no errors.

The next part was making sure the data points were standardised. This is another important step as this makes sure that data is internally consistent, and it also means reduces the size of the delta values when updating the weights, which means the resultant model will be more stable. To do this, I needed to get the maximum and minimum values of all the predictors and the predictand. So for each column, used the in built max() and min() functions to find the min and max values. Using these values, I calculate the standardised values for each data point. I decided to standardise my values between the ranges 0.1 and 0.9, using the formula below. This was done to help with the prediction of the model, so it would be able best produce values outside the scope of the training data predictors and would not be as subject to over training.

$$S_i = 0.8\left(\frac{R_i - Min}{Max - Min}\right) + 0.1$$

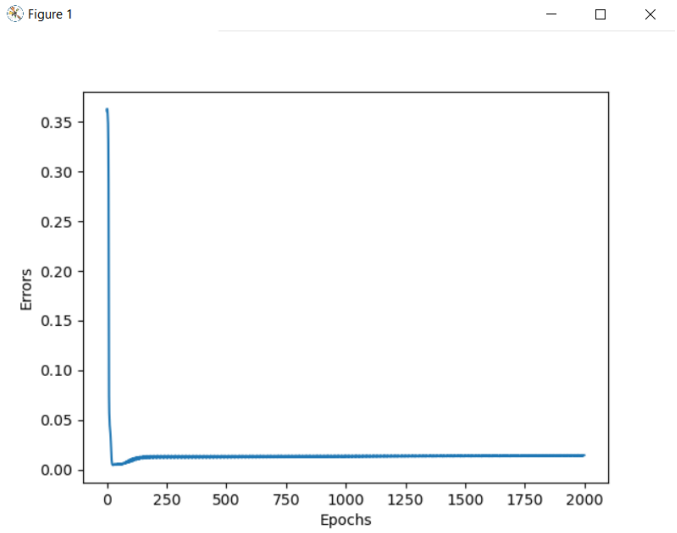
Finally, I split the dataset into three sets, the training set, the validation set and the testing set. This was decided by giving the training set 60% of the data, and the other two were given 20% each.

Part 2 – implementation of gradient descent

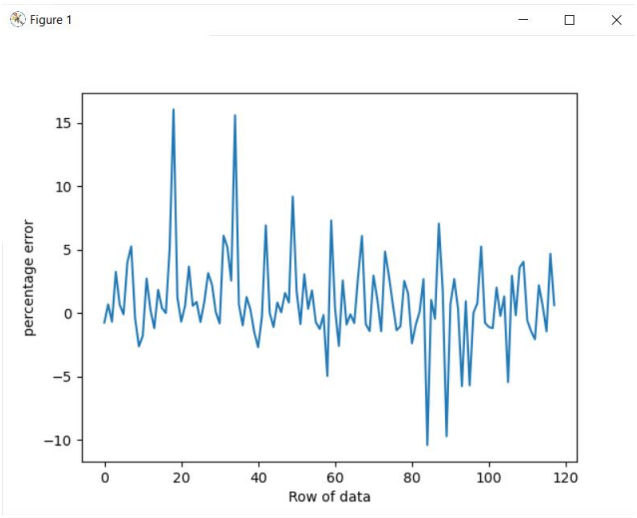
I decided to use a procedural approach when creating the system. So, I first began by creating a function to initialise a network model. The outcome of this function was to return all the weights of the model, all structured into individuals' layers and their own nodes. So, for this, the parameters the function needed were a number of inputs and a number of hidden nodes. So, for example with the inputs (8,4) the function produced a three-dimensional array with two layers, the first layer containing 4 node arrays, each with 9 weights associated to them – 8 for input and 1 for the bias value. The second layer then would contain one node, with weights from the hidden nodes to the output node including the bias for the output node. Each weight was initialised with random weights between -1 and 1.

After creating the network, the next task was to implement the forward propagation algorithm, so that inputs could be fed into the network in order to produce an output. This function – named 'forward_prop', needed the model of the network and a row of data as parameters. The main algorithm involved creating an output array which stored the results from each layer. And for each node in a layer, the function computed sum of all the weights for that node agaisnt the predictands. After running the first layer through the algorithm, it would produce all the outputs for the hidden nodes, using the inputs from the dataset. The second layer would produce a single value of the output in the output layer using the outputs from the hidden layer as inputs. I also created a 'node_activation' function which took in the outputs for each node and performed a sigmoid function on each so that values can be match to a predictand. The 'forward_prop' function would then later be implemented as part of the backpropagation algorithm aswell for predicting new index floods later for the validation set and testing set.

I then began to work on the backpropagation function, this function needed many parameters - these being the initial model of randomised weights, the training set, the learning rate, and the number of epochs. The initial way I implemented the backpropagation algorithm, was by taking each row of data and computing derivatives for each predicted output to each layer from the forward pass. Then I calculated the cost function for each node in the two layers. I then calculated new weights using the learning rate and the cost functions. The cost function was computed using the gradient descent method, which means calculating deltas values for each predicted output against the actual output. This meant for a training set of 100 data rows, weights would be updated every row of data per epoch. So already at this point I could see how the performance of the system may be impacted by many epochs. In this function I also computed a root mean squared error, so that I would be able to see the error for each epoch and monitor the performance. Then using the Matplotlib library I added each error per epoch to an error so I could plot the end results.



Another vital part of the system was a function that allowed me to test the network agaisnt the validation set and the testing set. This was a simple implementation; it combined the root mean error squared computation from the backpropagation algorithm aswell as the forward prop function in order to produce outputs using the new weights and biases from the improved network model. I also calculated a percentage error for each network, so that I two measures of model accuracy. For percentage error, the main goal was to find any model under 15% and for the root mean squared error, any value under 0.04 tended to be a hopeful model. This function also uses matplotlib in order to produce a graph of the percentage error between each correct output and the predicted value.



Finally, I created a model store function, that allowed me to save models that were below a certain threshold. This function stored the learning rate, number of hidden nodes, root mean squared error and the percentage error of the system. The threshold I had selected was a percentage error rate of 15 or less.

The graphs above were produced using a learning rate of 0.1 and 1000 epochs.

Part 3 – improvements

Whilst testing for improvements I had the model store threshold set to 40% so that I could see how the improvements changed over time. Aswell for each improvement, weights were kept constant throughout each iteration to make sure the weights did not produce anomalies and outliers. This was done by using a while loop around the validation function so the same initial network was passed through but values such as the learning rate, number of hidden nodes and epochs could be changed each time.

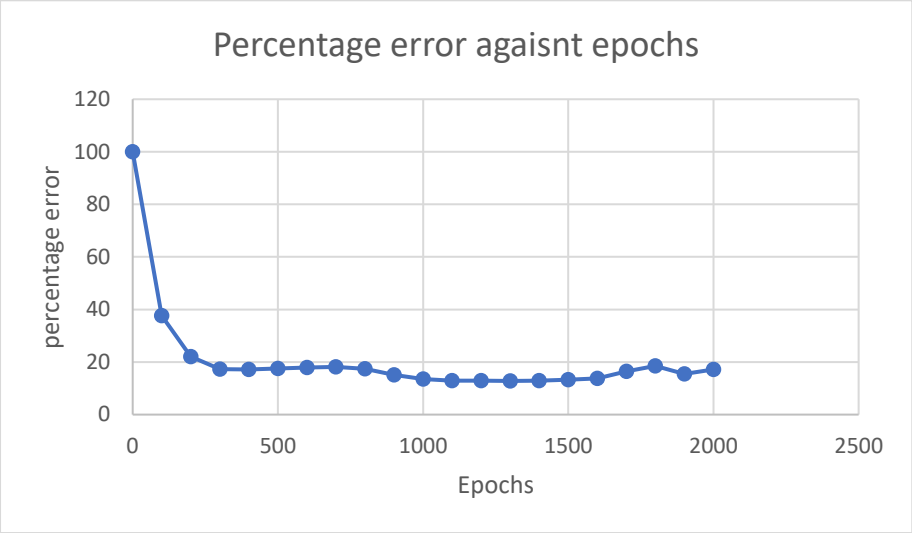
Changing number of epochs

The first improvement that I wanted to implement was how number of epochs affected the percentage error and improvement rate of the network. So, I kept a constant learning rate of 0.1 and 4 hidden nodes and began to check the performance of the network. I increase the epoch amount by 100 each time, I decided to pick a threshold of 2000 epochs as anymore would have a large runtime, with the current backpropagation algorithm.

I noticed instantly that a lower number of epochs impacted the actual performance of the network, a lower number of epochs, led to a quicker runtime than a higher number of epochs.

While run different models of the network with different number of epochs, I discovered that on average a larger number of epochs led to greater reduction in percentage error. This was expected as more cycles through the data would allow for more adjustments to the weights and therefore would improve the network overall.

As you can see from the graph below an epoch rage of 1000 to 1500 seemed to have the greatest improvement on the network. The lowest percentage error being 12.8% at an epoch of 1300.

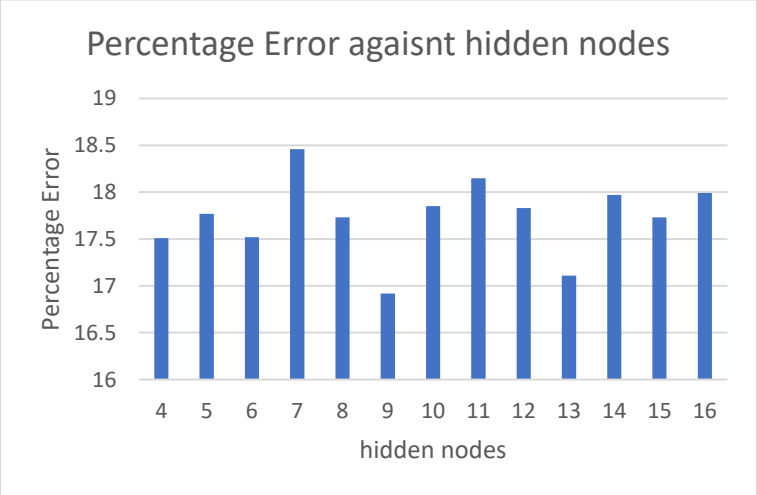


For this improvement, I did not keep the weights the same throughout, instead I took an average of percentage error at each epoch.

Changing the number of hidden nodes

For this improvement, I decided to use an epoch of 1300 as it was shown to provide the best percentage error. I also decided to test the hidden node range of 4 to 16.

For this improvement I expected to see a underfitting when there were too little hidden nodes and overfitting when the number of hidden nodes were too great and therefore an optimal number of hidden nodes that produced the lowest percentage errors. However, during my testing of the improvement, I discovered that the percentage error did not really budge too much. I believed that this was due to not having a large enough of epoch range, because increasing the number of hidden nodes means more updates need to be made so that all parameters work together.

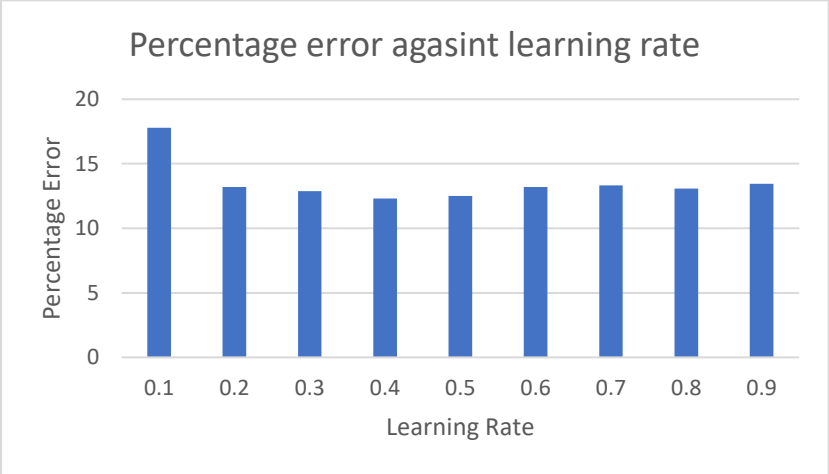


So, I picked three values for hidden nodes these being, 4, 8 and 16. I then used 2000 epochs and discovered the same trend, not much change occurred. I concluded that this could still be to do with the number of epochs, but with current performance of the algorithm and model it would be ineffective to test using 10000 epochs for each hidden node.

However, continuing from here the graph suggests a hidden node number of 9 would be good for finding the optimal value of each weight.

Changing the learning rate

I started enough with a learning rate of 0.1 and decided to see what a learning rate of 0.01 would do to the network, this was not a great result, the network produced worse results with a learning rate of 0.01. So, I concluded that to improve the model I would need to increase the learning rate. I then tried updating the

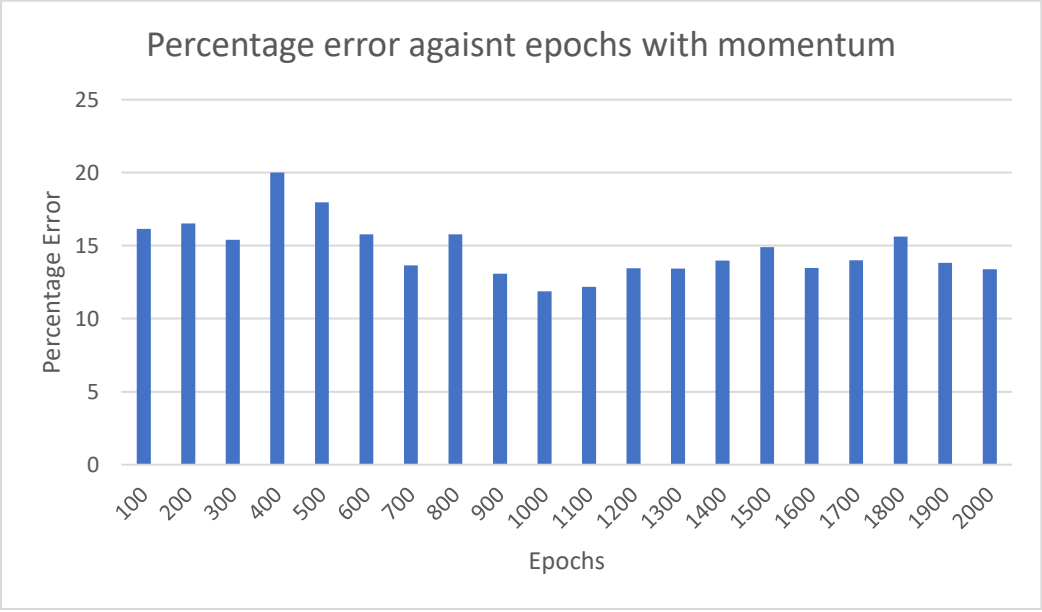


learning rate by 0.1 for each model. The results from this test showed that 0.1 was perhaps too low of a learning rate for optimal improvement of the model at 2000 epochs. 0.4 seemed to be the optimal value at 2000 epochs for producing the best model with a percentage error of 12.3.

The results do however also show that after about 0.5, the learning rate begins to negatively affect the model. This is due to the weight being changed by too great an amount, making it more difficult for the gradient descent to find the optimal weights.

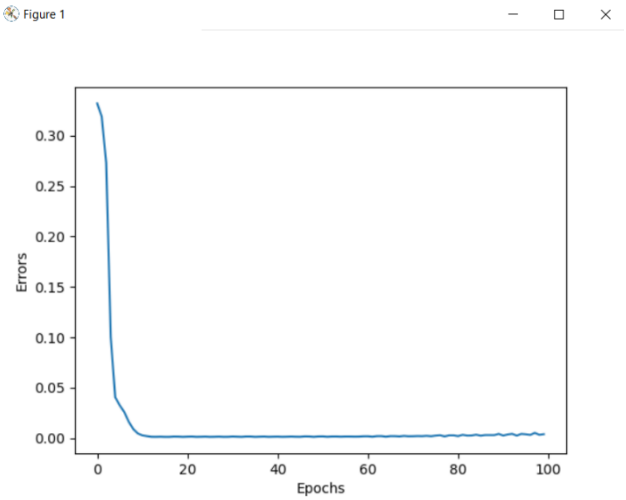
Momentum

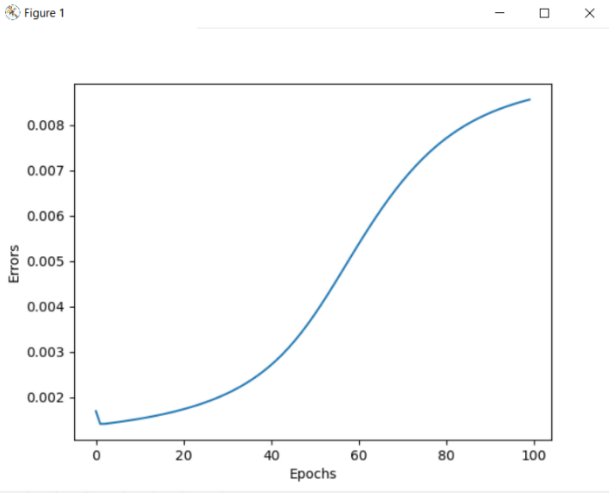
Momentum is a technique used to travel an extra step towards the optimum weight. It takes a percentage of the difference between the current weight and the previous weight adds that onto the new calculation of the weight. I decided I would use this improvement to speed up the runtime of the network. For this new tested I wanted to see what affect momentum had on the performance with varying epochs. I used the same learning rate throughout and the same number of hidden nodes. As I had discovered in the previous tests, the optimum number of hidden nodes was 9 and the optimum learning rate was 0.4 so I used those.



The results show that, momentum produces about the same accuracy as without momentum given the same number of epochs. However, momentum seems to stabilise the error in the network in fewer epochs. For example, below on the left is the root mean squared error with 100 epochs and momentum is being used. On the right is without momentum and it shows no stabilisation within 100 epochs and in fact the error is increasing slightly. So, it is clear to say that momentum has performance benefits and helps

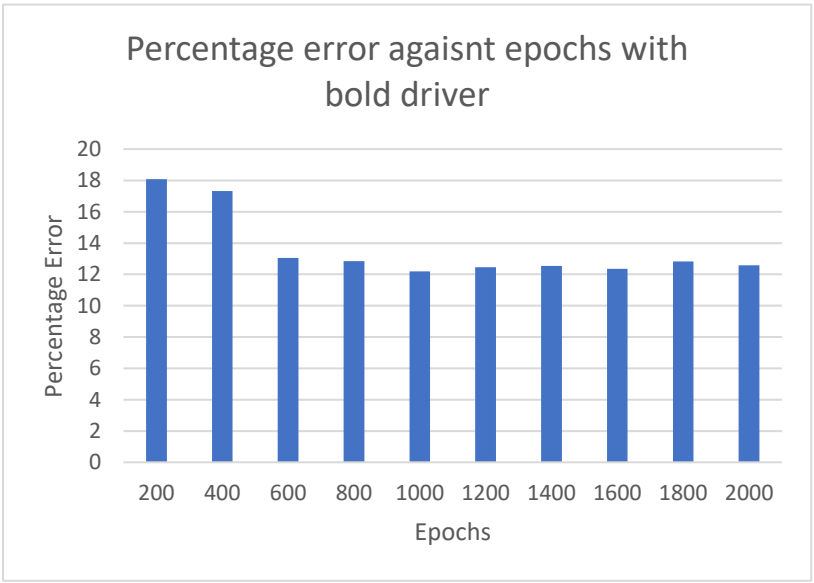
stabilise the weights more. However, it also shows some oscillation this could be due to taking too far an extra step in a gradient direction in order to minimise error. So maybe another improvement could added to this one in order to help smooth out the oscillations towards the end.





Bold driver

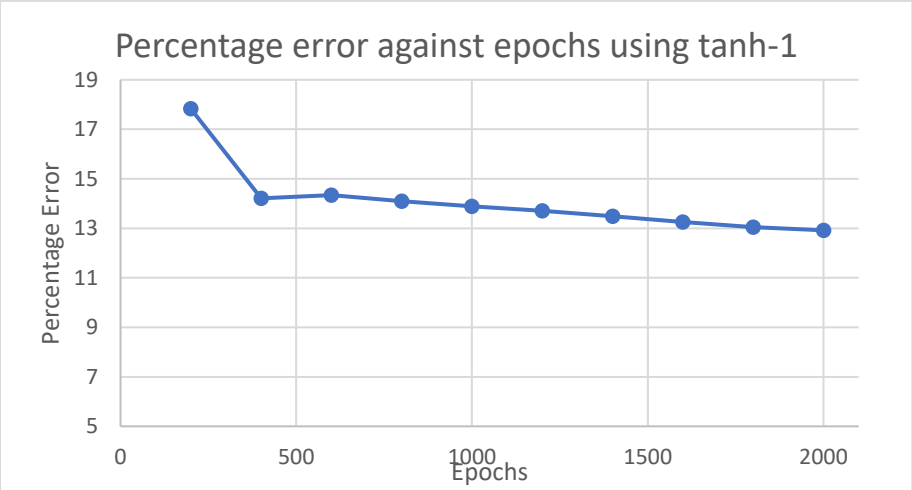
Bold driver is a technique used to adjust the learning rate, so as to speed up the performance of the network, for example if the error per epoch is larger than the previous error then the learning rate is decreased by 30% so as to make smaller changes to find the optimal weights. Otherwise, the learning rate is increased in order to make a greater leap towards the optimal weights.



The results from the bold driver improvement show similar results to the percentage error against epochs without any improvements. However bold driver is seen here to stabilise the error more there is less fluctuation, and the stabilisation appears faster than normal as after 600 epochs there isn't much change. In the graph without any improvements, stabilisation occurs at about 1000 epochs and then overtraining begins to occur and error increases slightly. Bold driver changes the learning rate as to not increase the risk of over and undertraining.

Using tanh-1 activation function

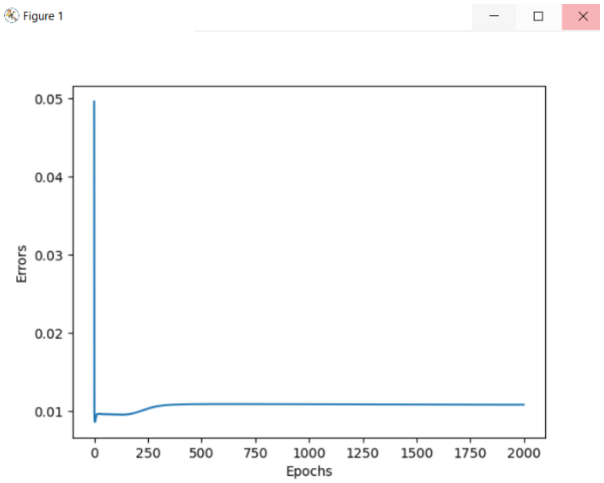
Another improvement I thought may be beneficial to improving the performance of the network was trying to change the activation function. For this I tried using the tanh-1 activation function this involved changing the range of the random weights from (-1,1) to (0,1), aswell as changing the derivative output equation. The benefit of tanh-1 is that tanh-1 derivatives are greater than sigmoid derivatives, which means that the global minima can be found a lot faster as greater error is reduced each training row/ epoch. In order to test this, I conducted another percentage against epoch test, this time with the activation function being tanh-1. To make the test fair I decided to use the same number of hidden nodes and learning rate as the initial percentage error agaisnt epoch test.



The results from tanh-1 improvement show less fluctuation than in the original percentage error agaisnt epoch graphs, as well as lower percentage errors for the same number of epochs as the original graph, this means that with less epochs the network now performs as well if not better than the original network. This all shows that tanh-1 improves the performance of the network due to having larger derivatives and therefore removing larger errors from the weights each epoch. So, from this I was to conclude that the tanh-1 activation function works better than the sigmoid activation function.

Simulated Annealing

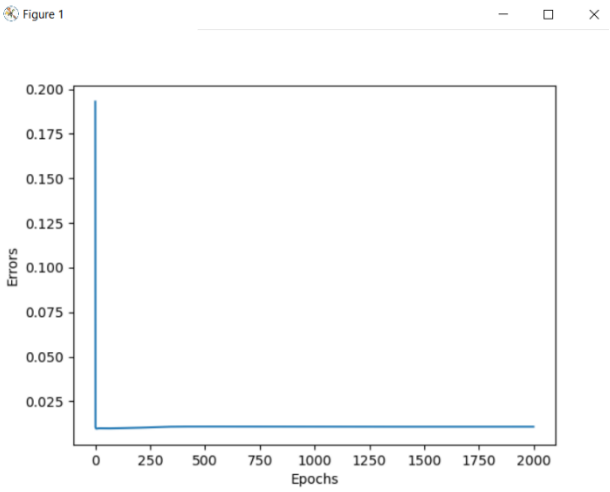
Simulated annealing is a technique used to change the learning rate every epoch so as to increase the performance of the network. This because at the start of neural network, a larger learning rate is needed to help quickly get close to the global minima, then as you continue to go through epochs this higher epoch can often cause oscillation when as the delta values/ cost function values produced are too great. So therefore, annealing helps reduce the learning rate over time to find the global minima. For this test, I kept the improvement of using the tanh-1 for the activation function, used the same number of hidden nodes throughout each epoch this being 9 and I was looking to see what affect on error rate per epoch, for a total epoch of 2000.



This is the graph of root mean squared error per epoch without annealing, with a learning rate of 0.1, the lowest error rate the network gets to whilst training is 0.005 and then it begins to rise again this being due to oscillating between the global minima. Then afterwards it gets stuck in a local minimum. Below are the values of the network agaisnt the validation set.

root mean squared error: 0.0374

Percentage Error: 13.97



This is the graph of root mean squared error per epoch with annealing, with a initial learning rate of 0.1 that decreased down to 0.01 by the end of the 2000 epochs, here it can be seen that there is lower errors per epoch and there is no oscillation. Below are the values of the network agaisnt the validation set.

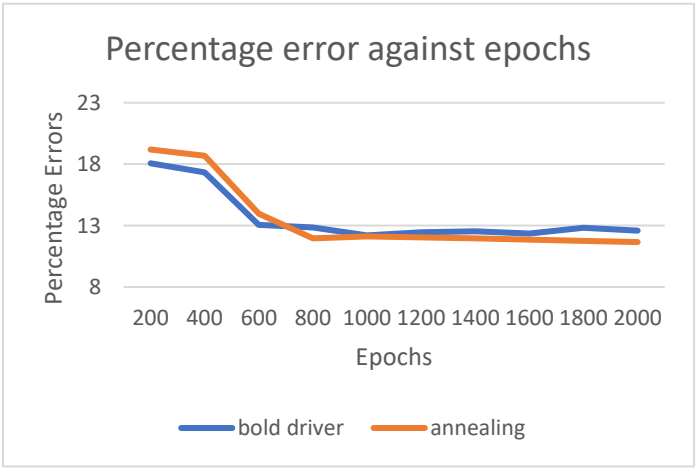
root mean squared error: 0.0397

Percentage Error: 13.2

Part 4 – Comparisons

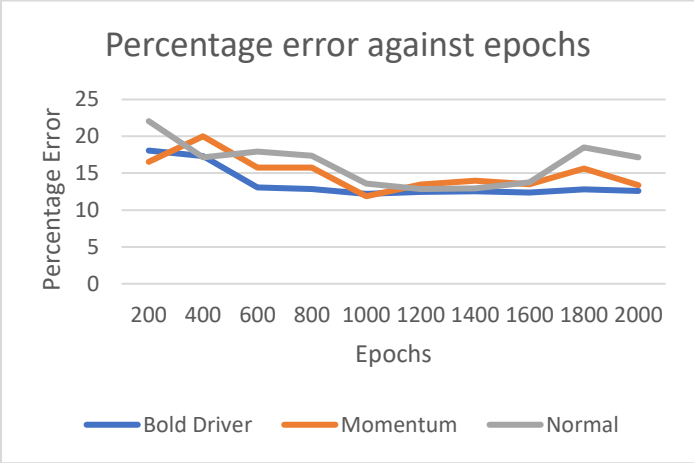
I tried to do some combinations between some improvement and see what affect they had on the percentage error rate, over 2000 epochs.

Bold driver vs annealing



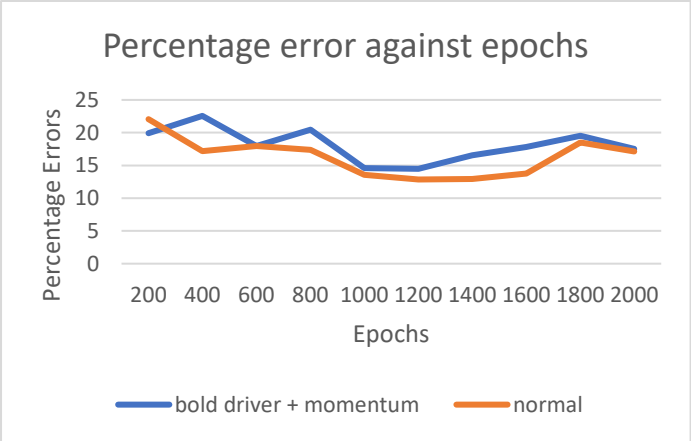
I ran a test to see which improvement would perform better after changing the activation function from sigmoid to tanh-1. As you can see from the graph at lower epochs bold driver has a greater affect on the performance, arriving near the global minima fastest, however after 700 epochs, annealing begins to become more effective than bold driver. Although both are still quite similar in percentage error. Using these results, I would conclude that that annealing would be a better improvement than bold driver. Due to annealing taking precedence in the algorithm, it would not be able to use both in unison.

Momentum vs normal vs bold driver



I also wanted to compare the difference between momentum and bold driver, so I used the normal unimproved model as a comparison. With low epochs 0 – 200 momentum seems to have the greatest impact, by finding the global minima in the quickest number of epochs. However later on bold driver seems to outperform both the normal model and the momentum model it also does not oscillate like the momentum and normal model, due to the reduction in learning rate as the error decreases, the idea is to make smaller changes to the weight the closer you get to the optimal weight.

Momentum + bold driver vs normal



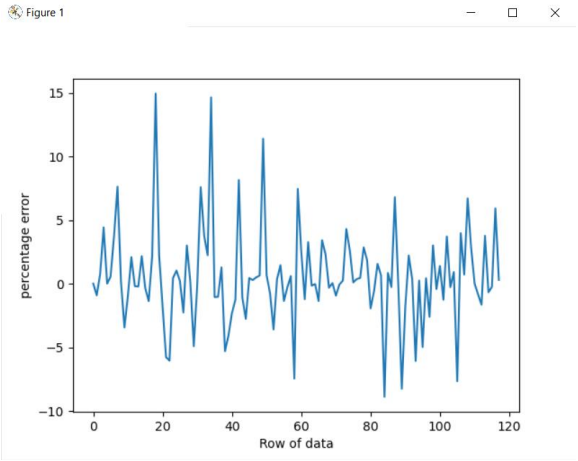
The final condition I wished to try was a combination of momentum and bold driver. I tried this because I thought bold driver could help smooth out some of the oscillations that adding momentum caused on the network. The results however the opposite, a combination of bold driver and momentum for a epoch range of 0 to 2000 actually produced models that were weaker than the a model that had not been improved at all. I think this may be down to momentum having to great of an impact on the weights. Aswell as maybe having too low of an epoch value. I did not think the results would be enough to discourage the use of both improvements at the same time, so later on I still ran a test with them both at 15000 epochs.

Finding the best Model

After trying these combinations, I took each of the best improvements and ran them for 15000 epochs to see the best network model that they could produce, in order to find one network that could be tested against the testing set.

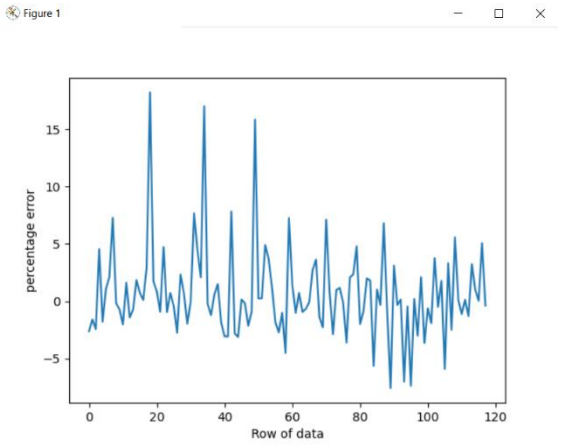
Annealing

After 15000 epochs annealing produce a network model, with a percentage error of 13.75% and error rate of 0.0381 – calculated using root mean squared error. The learning rate was set to an initial rate of 0.1 and it finished at 0.01. I also used 9 hidden nodes in this model, as this was the value concluded earlier to provide the best results.

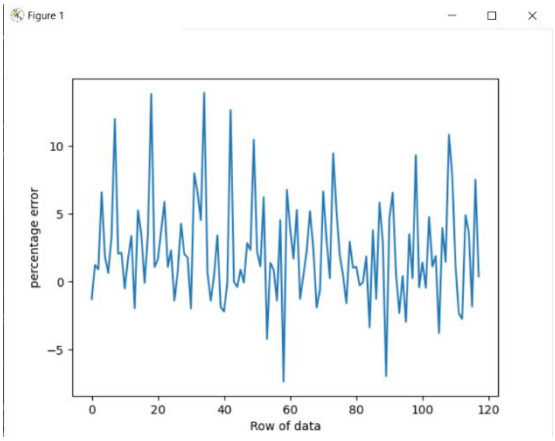


Momentum

For this final evaluation of momentum on the production of models, I used a learning rate of 0.1 and 9 hidden nodes. This produced a percentage error of 15.28 and a root mean squared error of 0.0404.

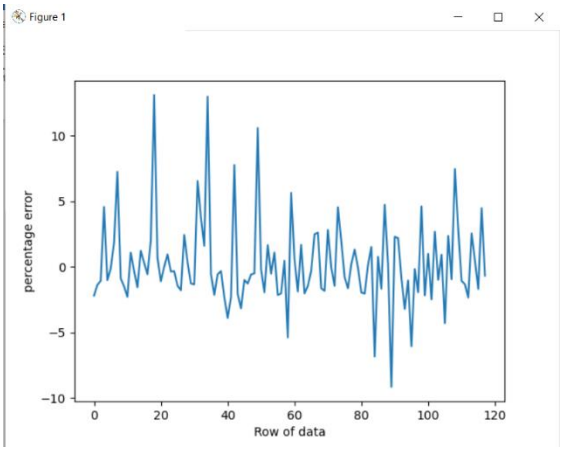


Bold driver



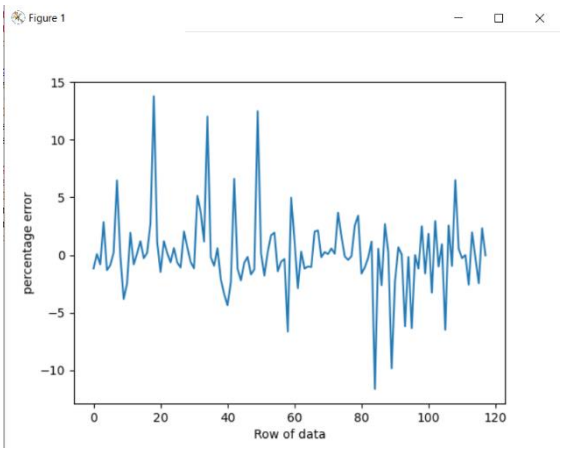
For this final evaluation of bold driver on the production of models, I used a learning rate of 0.1 and 9 hidden nodes. This produced a percentage error of 19.14% and a root mean squared error of 0.0443.

Normal with a tanh-1 activation function



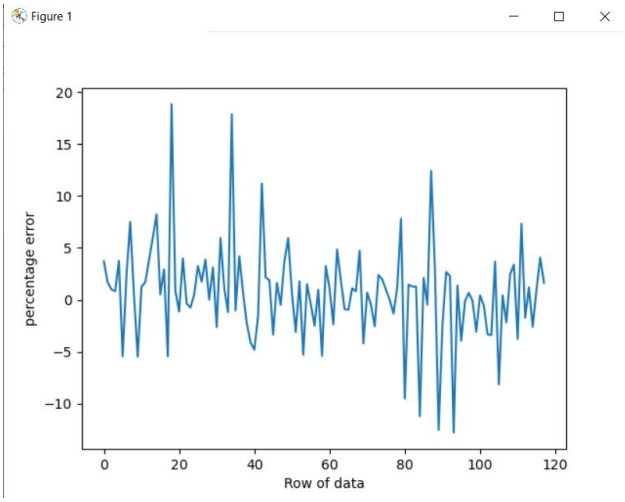
I tried the tanh-1 activation function, with 9 hidden nodes and 0.1 for the learning rate at 15000 epochs. This produced a model with a percentage error rate of 12.91% and a root mean squared error of 0.0330.

Normal with a sigmoid activation function



For this evaluation of the network, I tested the optimal number of hidden nodes and learning rate for an epoch of 15000. These being 9 and 0.4 respectively. This produced a model with a percentage error rate of 10.98% and a root mean squared error of 0.0339.

Momentum + bold driver + 0.1 learning rate



I combined momentum and bold driver in this final evaluation, this time I used a earning rate of 0.1 and 9 for the number of hidden nodes. The percentage error produced was 19.35, while the root mean squared error was 0.0469.

Part 5 – conclusion

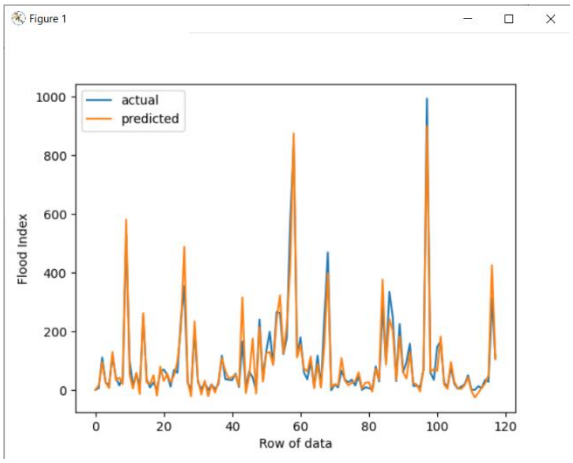
The table shows the comparisons between the 6 best improvements that I had found. From the table below the best percentage error found was within the model an increased learning rate using the sigmoid activation function. While the model with the smallest root mean squared error was the Normal Tanh-1 model.

	Momentum	Bold Driver	Normal + Tanh-1	Normal + sigmoid	Momentum + Bold Driver	Annealing
Percentage Error	15.28	19.14	12.91	10.98	19.35	13.75
Root mean squared error	0.0404	0.0443	0.0330	0.0339	0.0469	0.0381
Learning rate	0.1	0.1, but varied throughout	0.1	0.4	0.1, but varied throughout	0.1-0.01
Hidden nodes	9	9	9	9	9	9

With the sigmoid model not being too far away in error rate. I expected the other model improvements to have the biggest impact on percentage error and error rate due to them being able to find the global minima the fastest. However, these results indicate that perhaps a little over training began to occur during the large number of epochs.

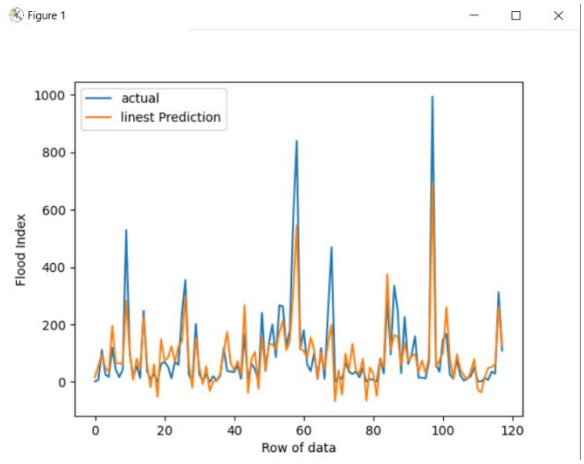
Using the results, that I have gathered, I concluded that the best model is the sigmoid model with a learning rate of 0.4 and 9 hidden nodes.

Best model – Comparison with a Data driven model.



Using the best possible model, I had created I ran the model against the unseen test data.

The overall percentage error on the training data was 10.18% with a root mean squared error of 0.0322. The graph on the left shows the results of the predicted values agaisnt the actual values of the testing data set, all the values appear to be close together and so shows the model to have good accuracy.



I also compared this to a simple data driven model, in this case this model was LINEST which I did using Excel. I took all the forecasted results of LINEST and the graph on the left shows how the forecasted values compared to the actual values. It is clear to see from the graph that percentage error exceeds that of the neural network model. Due to the greater variation between prediction and actual outcome. I also calculated the root mean squared error which came out as 0.0558. So, this shows the success and power of a neural network as it is able to provide better accuracy than that of a data driven model.

Short comings

In my report I was unable to test the benefits of batch programming, an improvement that I did have include in the actual written up code. However, having ran out of time, I decided I would not be able to include the batch testing in the report, given more time I would have like to see the outcome. I believe this would have had the greatest effect on performance compared to the other improvements.

I also would've like to change the momentum value from 0.9 to a lower value to see how the rate of momentum affect the performance of a system. So perhaps adding an option for user input for the value of alpha, the momentum.

Program Listing

This can be found in the MLP – Code listing pdf.