ANN Implementation Report

Corben Sayer (F026006)
AI Methods (21COB107)

Table of Contents

Data pre-processing 2
Cleaning the data 2
Identifying predictors 3
Splitting the data 4
Standardising the data 4
Implementation of the MLP algorithm 5
Introduction 5
Structure 5
Layer class 5
ANN class 5
Improvements 6
Momentum 6
Annealing 6
Weight Decay 7
Limitations 7
Training and network selection 8
Finding the best version of the training algorithm 8
Finding the best configuration 9
Evaluation of final model 11
Comparison with another data driven model 12
Appendix 14
Setup.java 14
Layer.java 15
ANN.java 15
Basic (no improvements) 15
All improvements (momentum, annealing and weight decay) 19
Final version (momentum and weight decay) 23

Data pre-processing

Cleaning the data

With my data set, the first thing I set about doing was to clean it. What I wanted to look for was any spurious or inaccurate values that may have been lurking in the data set. To locate these, I mainly did two things. The first thing I would do is plot a scatter graph of the values for each row in the data set, so I could see if there were any obvious outliers among the data. By looking at figure 1, you can immediately see that this has helped to detect some outliers. The second thing I would do, is work out what values in each column fell outside of 3 standard deviations of the mean. If there were any values that fell outside of this range, I would highlight them red so I could easily find them in the data set. I then would manually look through each red value and determine if was truly an outlier or not. How I would decide this, was based on how far outside the range the value fell, the values in the column on the previous and following days, and the other data in the same row. For example, looking at figure 2, although there are numerous values that are highlighted in red, just by looking at the other values around them, I determined that none of these were true outliers, and that they would be useful data to help train the ANN. Some data in red however were obviously spurious such as a '#' or a '-999' which you can spot in figure 1. Once I had then found all these values that needed to be cleaned, I decided to replace each one using interpolation by taking the average of the previous two days and the following two days. Finally, once this would all be done, I would plot a scatter graph of the new cleaned values and would be able to determine if the data had been sufficiently cleaned or not. For example, looking at figure 3, I can see no significant outliers and the data seems to follow a trend of peaking at regular intervals, likely due to seasonal differences in the weather. Therefore, I deemed it to be sufficiently cleaned. This process was followed until all the data was cleaned.

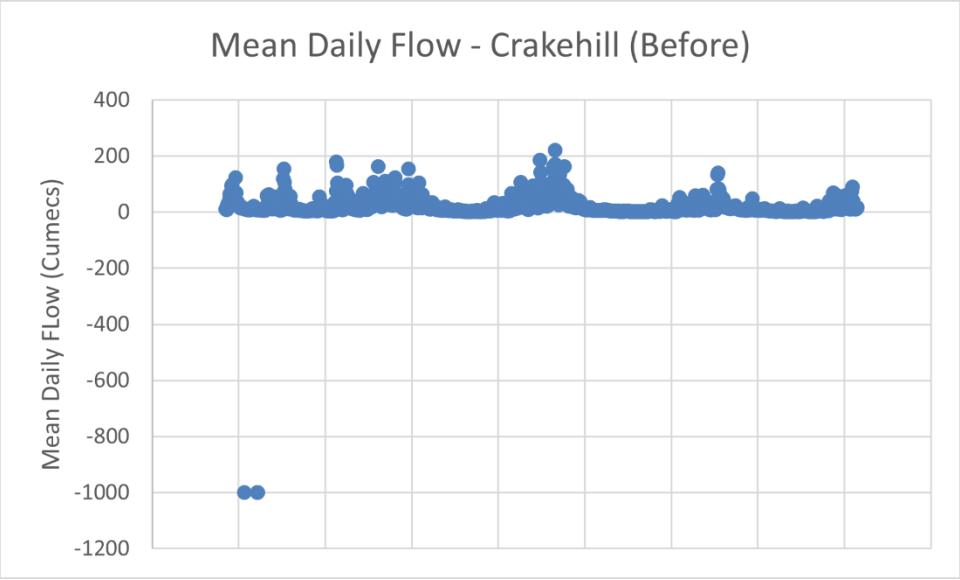


Figure 1 – the mean daily flow at Crakehill for each data point before any data cleaning

Date	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
11/09/1993	19.7	4.538	27.108	61.8	8	0	24	12.4
12/09/1993	12.2	3.333	15.395	34.91	5.6	0	7.2	11.2
13/09/1993	75.6	39.174	158.423	129	225.2	55.6	208.8	196.8
14/09/1993	180	70.723	153.231	317.4	103.2	74.4	30.4	22.4
15/09/1993	167	80.244	129.352	337.2	80	36.8	15.2	16.8
16/09/1993	105	51.684	54.93	258.9	12	15.2	0	0
17/09/1993	50	27.4	34.089	152.9	0	0	0	0

Figure 2 – a week of values from the data set with those that fell outside of 3x standard deviations of the mean highlighted in red

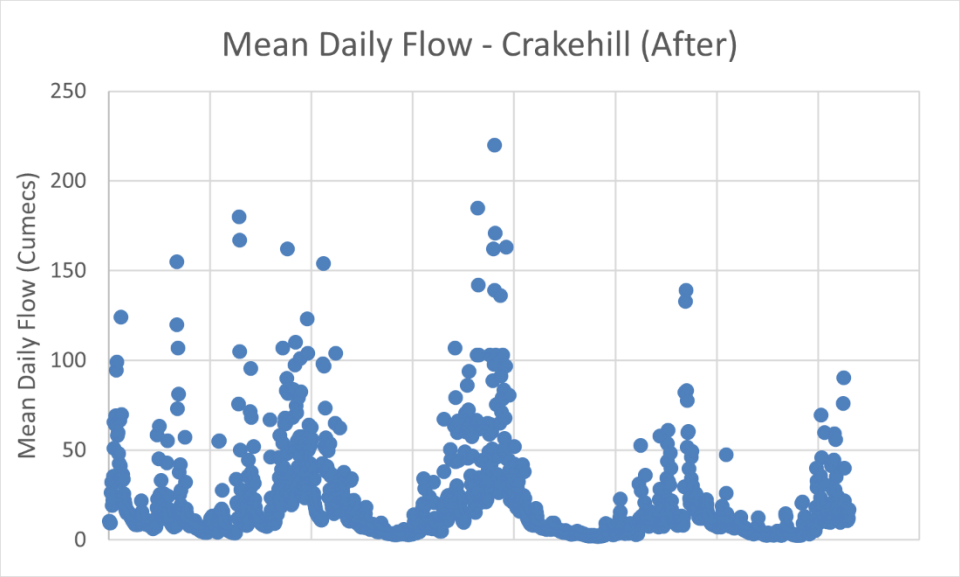


Figure 3 – the mean daily flow at Crakehill for each data point after data cleansing.

Identifying predictors

To identify the best predictors, I decided to test lagged values for each river flow column. For this, I tried lagged values of 1, 2 and 3 days. To determine which lagged measure was best, I would work out the correlation between the lagged measure and the river flow at Skelton. To work out correlation I used the CORREL function in excel. In each case, for river flows at Crakehill, Skip Bridge and Westwick, the lagged measure with the correlation closest to 1 was a lagged measure of 1 day. Looking at the river flow at Skelton, I decided that the previous days value might be a good predictor too, so I additionally lagged the river flow at Skelton and checked which lagged measure had the correlation closest to 1. In this case, a lagged measure of 1 day (i.e., yesterday) was also the best.

Next, looking at the rainfall data, I decided that using a weighted moving average for each rainfall column would be a good predictor. This is as total rainfall on its own as a predictor can be quite volatile, some days may have lots of rain, others not and thus smoothing it out might help it to act as a better predictor. Additionally, the reason I chose to do a weighted moving average, is because it's likely that yesterdays rainfall is going to have a bigger impact than the day before and so on. In the

end I ended up using a 3-day moving average as after 3 days I felt that the impact the rainfall that far back would have on the river flow would start to have less of an effect. The formula I used to calculate the 3-day moving average was $WMA_3 = \frac{0.5R_{t-1} + 0.3R_{t-2} + 0.2R_{t-3}}{3}$.

With my weighted averages, I then decided to lag them to see if a certain measure worked best. I found for all cases that a lagged measure of 1 day boasted the greatest correlation to river flow at Skelton. However, I noticed that the moving average rainfall at East Cowton had a correlation around 0.4. Although this still indicated a moderate correlation, I decided to omit this as a predictor as it wasn't as good as the other predictors, which all scored correlations above 0.6.

Thus, I settled on 7 predictors those being the lagged measures of 1 day of river flow at Crakehill, Skip Bridge, Westwick and Skelton and the lagged by 1 day, 3-day weighted moving average of rainfall at Arkengarthdale, Malham Tarn and Snaizeholme.

Splitting the data

I randomised the order of the values in excel and then split up the data into two parts. The first part containing 80% of the dataset had the first 75% (60% overall) for training data and the remaining 25% (20% overall) to be validation data. The second part containing 20% of the overall dataset was wholly for the testing data. How I randomised the order of the dataset in excel was by adding a column to it by using the rand() function and then sorting the dataset by this rand value.

Standardising the data

Next, I standardised the data. I decided to standardise the data in the range 0.1,0.9 to allow for the ANN to be able to predict values outside of the data set. For this I used the formula $S = 0.8 * (\frac{R-min}{Max-min}) + 0.1$. I decided to do this all in excel rather than a program as it was considerably easy in excel. If I wanted to de-standardise a value, I would use the formula $R = \left(\frac{S-0.1}{0.8}\right)(Max - \min) + \min$. All values were standardised using the max and min of the combined training and validation dataset, excluding the testing set.

Implementation of the MLP algorithm

Introduction

The language I used for my implementation of the MLP algorithm was Java. I decided to use Java it is a language I am familiar with, and I wanted to implement my ANN using an object-oriented approach. The only library I used was the Java.io library. I implemented my MLP using the following approach: There is a Layer class and an ANN class. All code can be found in the appendix. With the comments, it should hopefully be relatively easy to follow.

Structure

Layer class

A layer class represents either a hidden or output layer of an MLP. It contains information such as the number of nodes that layer contains, an array of the weights for those nodes and an array of the biases for those nodes. A layer is initialised by passing the following variables, the number of weights, the number of biases, the number of nodes and the number of inputs (i.e., the previous layer's number of nodes). The weights and biases are automatically given random values of between 2 divided by the number of inputs and -2 divided by the number of inputs.

ANN class

An ANN class represents the MLP itself. The ANN is made up of an array of layers and contains all the methods involving training the MLP. The ANN is initialised by passing an integer array representing the number of nodes in each layer. For example, to create an ANN with 2 inputs, 2 hidden nodes and 1 output node the integer array would be [2,2,1]. If you wanted to create an MLP with additional hidden layers, then you would just add more integers into the array, so long as the number of inputs was the first integer, and the number of outputs was the last integer. Inside of the ANN class there are the following public methods: train and predict. There are the following private methods: forwardPass, backwardPass, updateVariables and test.

The train method is passed the following parameters: an integer representing the number of epochs to run for, a double representing the learning parameter, 2D double array containing the training set, a 2D double array containing the validation set, a 2D double array containing the test set and a string array of the dates of the test set (the nth element of the test set should correspond to the nth element of the test set dates). Each element of a 2D array should be of the format the first n-1 elements are predictors, the nth element is the observed value. The train method contains a for loop which runs for the number of epochs given. Inside of this for loop, there is an additional for loop which loops through each element of the training set that was given. In this loop the forwardPass, backwardPass and updateVariables methods are called in this respective order. Once this loop is finished, the RMSE of the training set is calculated and this information is stored to a text file, along with what number epoch the training is currently at. Every 1000 epochs, the RMSE of the validation set is calculated and if the validation RMSE has increased compared to last time it was checked, but the training RMSE has kept going down, then the training will terminate as it would seem like the ANN has begun to become overtrained. Additionally, if the RMSE of the training set starts to barely go down every 1000 epochs, then training is terminated as it is likely the ANN has begun to plateau. Once training has been terminated, or the for loop for the number of epochs finishes, the test method is called, and the train method has finished.

The predict method is passed an array of predictors and will return a double of the value it has modelled.

The forwardPass method returns a 2D double array of the activation values at each layer for each node and is passed a double array of predictors as a parameter. First, a 2D array called allActivations is initialised which will store the activation value for each node in each layer. Then there is a for loop which loops through each layer of the ANN. Inside of this for loop there is an additional for loop which loops through each node in that layer. Here the sum of the bias and the weights of each node multiplied by the previous layer's values are calculated. This sum is then run through the sigmoid activation function and added to the activation array. Once the for loops are finished, the 2D array containing all the activation functions are returned.

The backwardPass method returns a 2D double array of the delta values at each layer for each node and is passed the following parameters: an observed value and a 2D array of activation values. First, a 2D array called allDeltas is initialised which will store the delta value for each node in each layer. Then there is a for loop which loops through each layer of the ANN backwards. If the layer is the output layer, then the delta is calculated using the delta equation for outputs nodes, otherwise, the delta is calculated using the delta equation for a hidden layer node. Once the delta value is calculated, it is added to the allDeltas array. Once all the nodes delta values have been calculated, the 2D array containing all the delta values is returned.

The updateVariables method is passed the following parameters: a double representing the learning parameter, a 2D array containing all the activation values, and a 2D array containing all the delta values. The code loops through each layer and updates all the weights and biases using the appropriate activation and delta values.

The test method is passed the following parameters: a 2D double array containing the test set and a string array containing the dates for each element in the test array. It loops through each element in the test set and saves to a text file the observed value, the modelled value, and the date. This method is used to help evaluate the performance of the trained ANN model.

Improvements

I added the following improvements to my MLP algorithm: momentum, annealing and weight decay. Below I will explain how I added them, but later in the section on training and network selection I will explain what improvements I decided to keep and why.

Momentum

To add support for momentum, when the ANN is initialised, an array containing the previous weight changes and an array containing the previous bias changes are created. When the weight or bias is updated, it is additionally updated by 0.9 multiplied by the previous change. Every time a weight or bias is updated the appropriate value in the appropriate array is updated to represent the value the weight was changed by so it can be used for next time.

Annealing

To add support for annealing, at the beginning of the train method, before the epoch loop, I would define the start learning parameter, being the value passed into the train method, and the end learning parameter which is set to 0.01. Then at the start of the epoch loop, I would update the learning parameter using the following equation:

$$lp = endLp + (startLp - endLp)(1 - \frac{1}{1+e^{10-\frac{20*currentEpochNum}{\max Epochs}}})$$

Weight Decay

To add support for weight decay, I created a method called getOmega. When called, this would return a double representing the omega value from this equation:

$$omega = \frac{1}{2(number\ of\ weights\ and\ biases)}(sum\ of\ all\ the\ weights\ and\ biases\ squared)$$

Then, before calling the backward pass, I would calculate the upsilon value using the following equation: $upsilon = \frac{1}{lp * currentNumEpochs}$. Then, when calculating the delta values for an output node, I would use the following equation: $\big((observedValue - activationValue) + getOmega * upsilon\big)\big(activationValue * (1 - activationValue)\big)$.

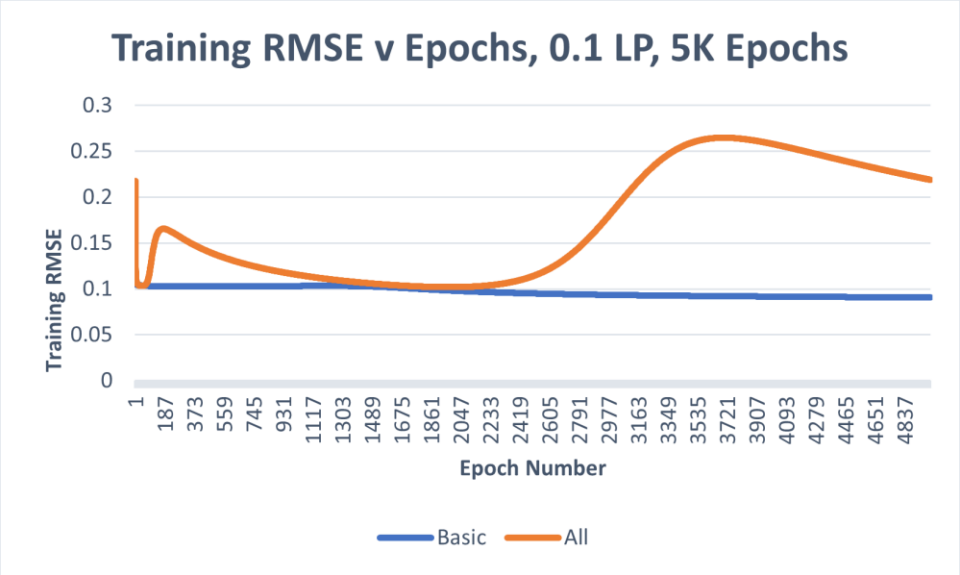
Limitations

There way I decided to implement my code means there are no limitations on the number of inputs you use or the number of hidden layers (and their number of hidden nodes). However, there is a limit on the amount of output nodes you decide to have. The MLP wouldn't work as intended if you tried to have multiple output nodes, as every piece of training data is treated as if there is just one output and when calculating deltas, it is assumed there is only one output node. This is something I would work on adding support for, given more time, but wasn't a priority as my model only needed to have 1 output node. Furthermore, currently the test method is passed a string array containing the dates, if you weren't using time series data, you wouldn't want to have this, but it is relatively easy to remove.

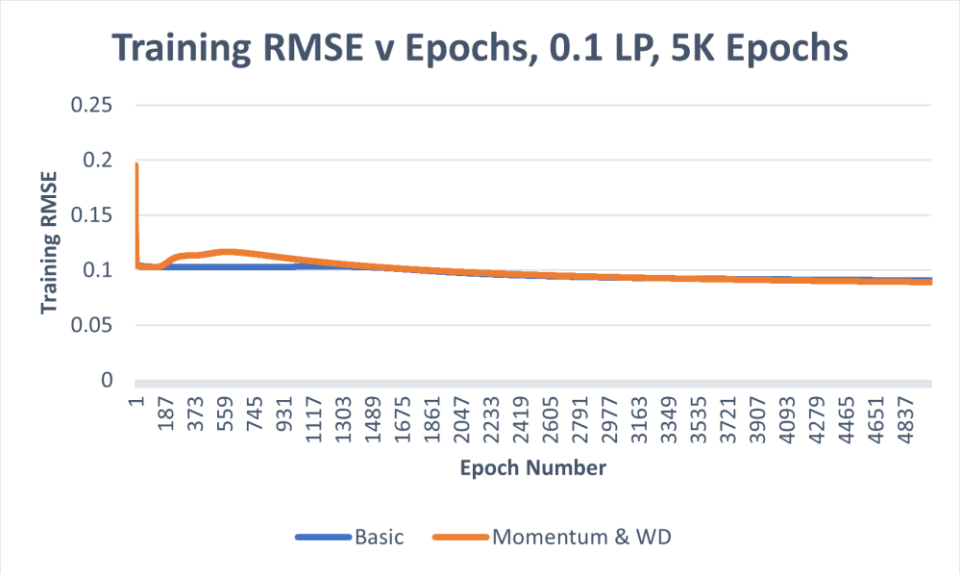
Training and network selection

Finding the best version of the training algorithm

Once I had added the improvements, I compared how the training RMSE decreased as more epochs passed for the basic algorithm and all 3 improvements (momentum, annealing and weight decay). In both cases I used 7 inputs, 4 hidden nodes and 1 output with a learning parameter of 0.1 and training for 5,000 epochs.



Looking at the training RMSE for the basic implementation vs the implementation with all 3 improvements, I noticed at around half the epochs, when the annealing starts to change the learning parameter a lot, the training RMSE increases significantly for the version with the improvements. As such, I decided to test the basic implementation vs just momentum & weight decay. A reason as to why the implementation with all 3 improvements might be more all over the place is because the algorithm might be looking more actively over the whole weight space.



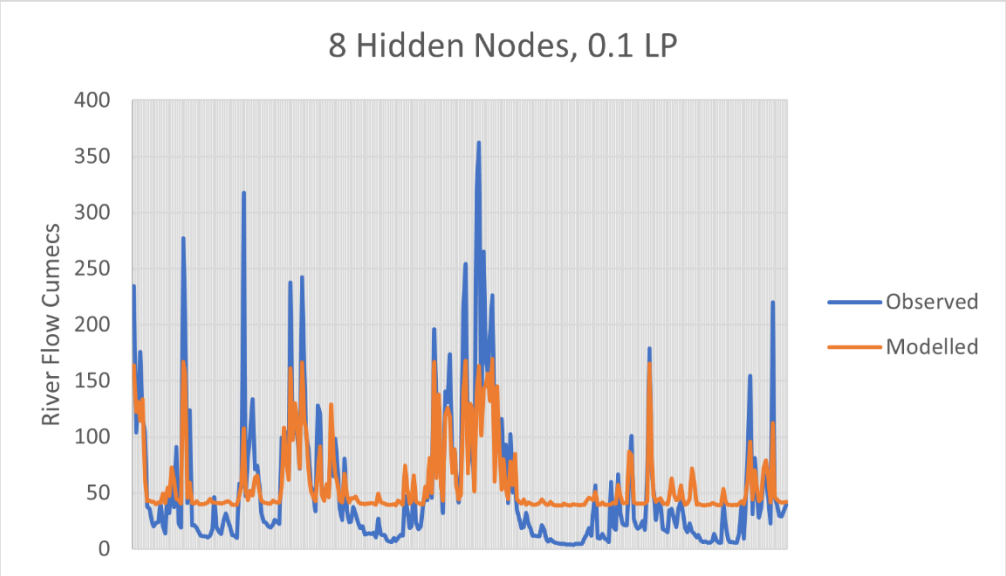
Here, you can instantly see that the implementation without annealing produces a much more desirable RMSE value, and as such, I decided to use this version for testing the best network configuration for my ANN as I believe it to be the fastest version at learning.

Finding the best configuration

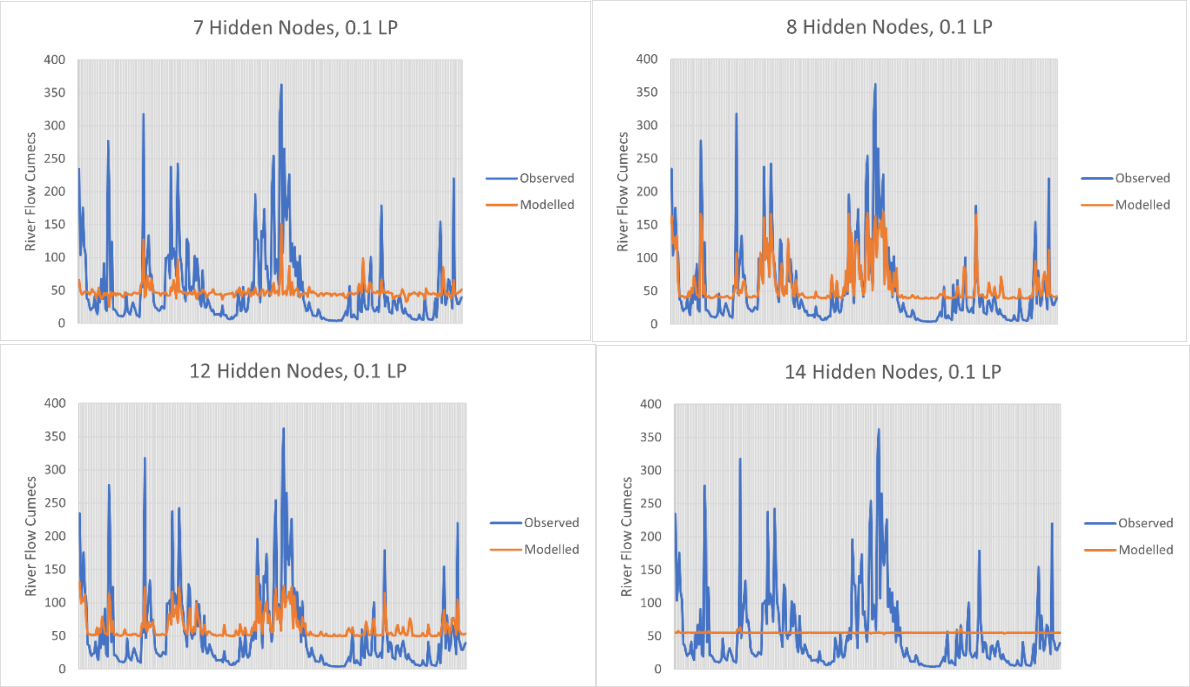
These are the values I had on my test set after training my ANN on the following different configurations running for 20,000 epochs or until training is terminated. On each run, the weights and biases are randomised. Red indicates the best value for that performance metric.

Learning Parameter	Hidden Nodes	RMSE	MSRE	CE	R-Squared
0.1	3	60.929	7.480	-0.0124	0.287
0.1	4	55.605	9.0365	0.157	0.730
0.1	5	53.081	16.0393	0.232	0.767
0.1	6	59.129	10.690	0.0465	0.745
0.1	7	55.599	7.834	0.157	0.543
0.1	8	**35.815**	**5.992**	**0.650**	0.886
0.1	9	47.061	10.447	0.396	0.808
0.1	10	49.362	12.111	0.336	0.879
0.1	11	49.439	11.154	0.333	0.876
0.1	12	45.430	10.678	0.437	**0.912**
0.1	13	51.324	13.318	0.282	0.865
0.1	14	60.521	12.955	0.00109	0.129

Looking at these results, we can determine a few things. The fewer hidden nodes there are, the harder the MLP finds it to model accurate values. This is clearly shown by looking at the CE (coefficient of efficiency) score. The value for 3 hidden nodes was even negative, suggesting that that model was worse than a no knowledge model. Similarly, we find that when there starts to become too many hidden nodes the CE decreases significantly as well (look at 14 hidden nodes), suggesting a too complex model finds it difficult to accurately model results as well. The model which performed best seemed to be the model with 8 hidden nodes, which interestingly had 1 more hidden node than there were inputs, perhaps suggesting a better model has slightly more hidden nodes than inputs. Additionally, the 8 hidden node model also performed best in both the RMSE and MSRE measures, suggesting it was one of the best models at getting close to the actual results. Looking at the graph of modelled values and observed values, we can clearly see that it has done a decent job.



When looking at the graph for example at 12 hidden nodes compared to the graph for 7 hidden nodes, we can clearly see the difference that as is also displayed by the r-squared metric. It seems that models with more hidden nodes are better at modelling the shape of the actual results. However, as with the coefficient of efficiency, with around 14 hidden nodes, it seems to become too much and the model retreats to being no where near as accurate as it's predecessors. In fact, it seems to be nothing more than a straight line, perhaps settling on always guessing a mean value. An important thing to note though, is having an r-squared value doesn't seem to mean that the model would be a better at predicting a flood, because it is merely matching the shape, but seems to be lacking the highs and lows that the river flow might experience.

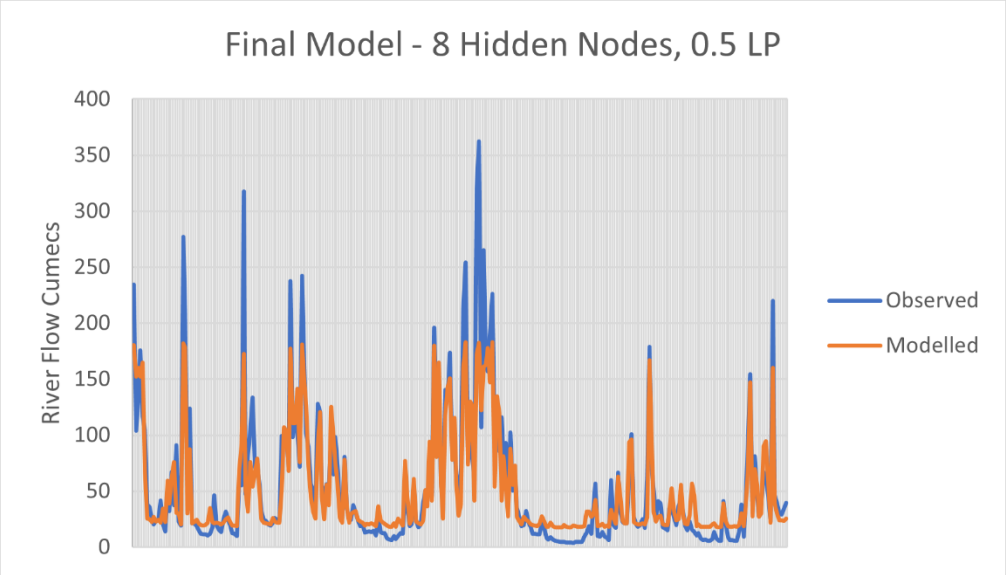


With all these comparisons in mind, I concluded that the model with 8 hidden nodes seemed to be the best, however, there was still clearly room for improvement. Looking at the graph for 8 hidden nodes, it could be much better at predicting both the highest and lowest values. As such, I decided to test out the 8 hidden node model with a few different starting learning parameters and see what results they yielded.

Learning Parameter	Hidden Nodes	RMSE	MSRE	CE	R-Squared
0.01	8	53.247	17.425	0.227	0.784
0.1	8	35.815	5.992	0.650	0.886
0.5	8	**26.441**	**0.894**	**0.809**	**0.913**

After testing these models, it was obvious that the starting learning parameter of 0.5 seemed to significantly outperform the others. This was likely because the higher learning parameter allowed it to more aggressively search the weight space and find more optimum weights. However, this would not necessarily always be the case as it could also result in skipping over the best weights my trying to search to much of the weight space at once. Looking at the learning parameter of 0.01, it seems like the value was too small for the algorithm to quickly find a more optimal set of weights. As such, I decided for my final model to use 8 hidden nodes with a learning parameter of 0.5.

Evaluation of final model



RMSE	MSRE	CE	R-Squared
26.441	0.894	0.809	0.913

Looking at the final model, it seems to be doing a decent job at accurately predicting what the river flow at Skelton will be. Looking at the MSRE, we can see that it seems to be less than 1 cumec between the observed and modelled values, which means on an average day, this model will give you a fairly accurate approximation. However, it is also clear from the graph that the MLP seems to struggle to predict the very peak values and the lowest values. Considering the lower values, if this model was to be used to forecast floods, that isn't too much of an issue. Conversely, considering the peak values, this may be a bit of an issue if it was to be used for predicting floods. Floods are more likely to happen when the river flow is at its highest, and as such this model seeming to cap off at predicting around 200 cumecs, may not be desirable. However, considering how accurately the model seems to be able to predict when the river flow will be high in general, it could still serve as a useful model for flood prediction. For example, you would know when it may be about to flood, due to the model predicting a high value, you just couldn't be certain. This would allow you to at the very least be prepared. Similarly, if the value predicted isn't high, then you could have a pretty good guess that the river won't be about to flood.

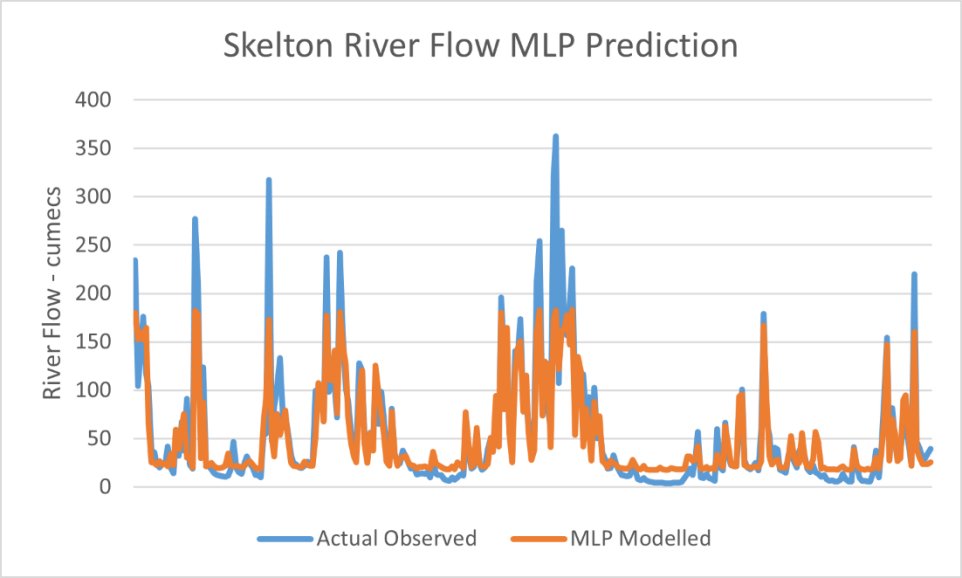
Thinking about why the model might struggle to predict these highest values, I have a couple of ideas. One may be that the very highest river flow values are only seen a handful of times a year and considering the data set only covers a period of around 4 years, it may just not be trained on enough peak values to be accurate at guessing them. Perhaps given an even greater set of data to train on the model would be able to improve on this.

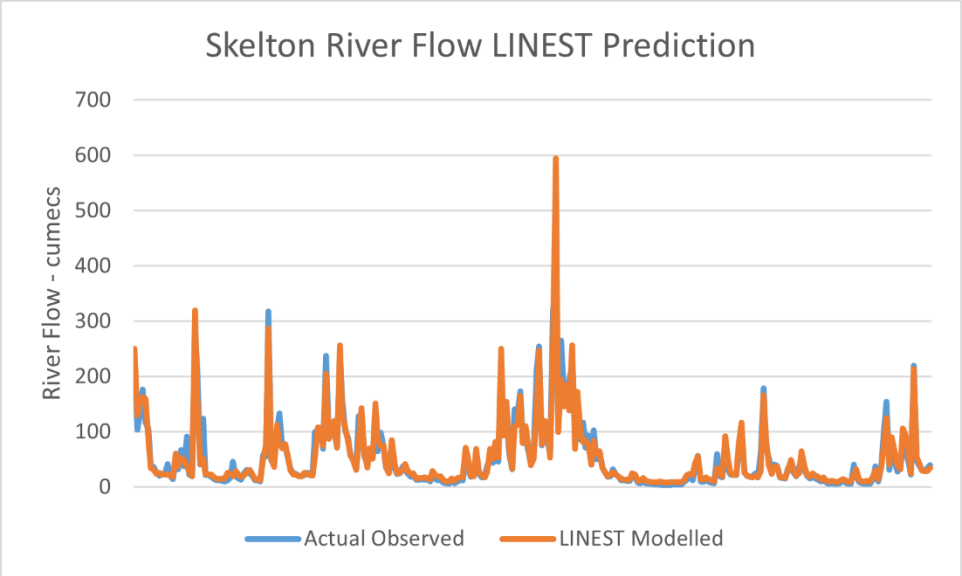
Overall, it seems as if an MLP is a decent model to use to predict for floods, but there may be alternatives as I'll discuss below.

Comparison with another data driven model

I decided to compare my model against the LINEST model in excel. I created the function for the LINEST model using the training and validation data and then ran the test data on this function. The results are shown below.

Model	RMSE	MSRE	CE	R-Squared
MLP	26.441	0.894	0.809	0.913
LINEST	21.674	0.872	0.872	0.940





When comparing my MLP compared to the LINEST function, it seems that the MLP model is outperformed by LINEST in every error metric. This is something I wasn't expecting, even though I assumed the LINEST function would result in a decent model. I believe this may have something to do with an error in my MLP implementation, but I have been unable to notice any errors in my code and it seemed to perform as expected on other examples. Perhaps if I were to locate this hypothetical error then my MLP would have been able to produce a better model, but I am not certain.

Overall, I believe this highlights how an MLP will not always be the best model to use for predicting something. As we can see above, it is outperformed by the LINEST function. However, it definitely should be considered for a lot of things. For example, given a better implementation, I'm sure that the MLP would be able to outperform the LINEST function, even if not by much. On the other hand, perhaps an alternative data driven model would be even better than that.

Finally, here is my prediction for the river flow at Skelton on 01/01/1997.

River Flow at Skelton Prediction 01/01/1997

My MLP	37.786 Cumecs
LINEST	33.162 Cumecs

Note: throughout this document some graphs are shown without x axis labels throughout this document, that is likely because it is from the test data. This data has dates and is organised chronologically but since the data won't necessarily be evenly spready from one date to the next i.e., you could have two consecutive days but then no data for the next 7 days, the x axis label has been omitted.

Appendix

Here is all my code. The setup class is simply where you can read the data and create and use the ANN. The source code can additionally be found in a separate folder attached with this document.

Setup.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Setup {
    public static void main(String[] args) throws IOException {
        //read data from relevant text files and put into appropriate sets,
change as appropriate to fit your data set
        double[][] trainingSet = new double[874][8];
        double[][] validationSet = new double[292][8];
        double[][] testSet = new double[292][8];
        String[] testDates = new String[292];
        BufferedReader br = new BufferedReader(new
FileReader("Train&ValidationData.txt"));
        String line = null;
        int count = 0;
        while((line = br.readLine()) != null){
            String[] data = line.split(",");
            double row[] =
{Double.parseDouble(data[0]),Double.parseDouble(data[1]),Double.parseDouble
(data[2]),Double.parseDouble(data[3]),

Double.parseDouble(data[4]),Double.parseDouble(data[5]),Double.parseDouble(
data[6]),Double.parseDouble(data[7])};
            if(count < 874){
                trainingSet[count] = row;
            }else{
                validationSet[count-874] = row;
            }
            count++;
        }
        br.close();
        br = new BufferedReader(new FileReader("TestData.txt"));
        count = 0;
        while((line = br.readLine()) != null){
            String[] data = line.split(",");
            double row[] =
{Double.parseDouble(data[0]),Double.parseDouble(data[1]),Double.parseDouble
(data[2]),Double.parseDouble(data[3]),

Double.parseDouble(data[4]),Double.parseDouble(data[5]),Double.parseDouble(
data[6]),Double.parseDouble(data[7])};
            String date = data[8];
            testSet[count] = row;
            testDates[count] = date;
            count++;
        }
        br.close();

        /*an array with element 0 containing the amount of inputs, elements
1 to n-1 containing the amount
        of neurons in that respective hidden layer and element n containing
the amount of output neurons*/
        int[] layerSizes = {7,8,1};
```

```
        ANN ann = new ANN(layerSizes); //create an ann
        ann.train(20000, 0.5, trainingSet, validationSet, testSet,
testDates); //train the ann
        double[] x =
{0.154107,0.148732,0.124928,0.164312,0.145753,0.129744,0.11042}; //some
predictors
        System.out.println(ann.predict(x)[0]); //the prediction
    }

}
```

Layer.java

```
public class Layer {
    private double[] weights;
    private double[] biases;
    private int numNodes;
    private int numInputs;

    public Layer(int numWeight, int numBiases, int numNodes, int numInputs)
{
        double[] weights = new double[numWeight];
        double[] biases = new double[numBiases];
        double min = -2.0/numInputs;
        double max = 2.0/numInputs;
        for(int i = 0; i < numWeight; i++){
            weights[i] = min + (Math.random() * (max - min));
        }
        for(int i = 0; i < numBiases; i++){
            biases[i] = min + (Math.random() * (max - min));
        }
        this.weights = weights;
        this.biases = biases;
        this.numNodes = numNodes;
        this.numInputs = numInputs;
    }

    public double[] getWeights(){ return this.weights; }
    public double[] getBiases(){ return this.biases; }
    public int getNumNodes(){ return this.numNodes; }
    public int getNumInputs() { return this.numInputs; }

}
```

ANN.java

Basic (no improvements)

```
import java.io.*;

public class ANN {
    private Layer[] layers;

    public ANN(int[] layerSizes){
        Layer[] layers = new Layer[layerSizes.length-1];
        for(int i = 0; i < layerSizes.length-1; i++){
            int numWeights = layerSizes[i]*layerSizes[i+1];
            int numBiases = layerSizes[i+1];
            Layer currentLayer = new Layer(numWeights, numBiases,
layerSizes[i+1], layerSizes[i]);
            layers[i] = currentLayer;
        }
        this.layers = layers;
    }
```

```
    private double[][] forwardPass(double[] predictors){
        double[][] allActivations = new double[this.layers.length][];
//save all the activation values
        for (int i = 0; i < this.layers.length; i++) {
            Layer currentLayer = this.layers[i];
            double[] activations = new double[currentLayer.getNumNodes()];
            for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                double sum = currentLayer.getBiases()[j];
                for (int k = j * currentLayer.getNumInputs(); k < (j + 1) *
currentLayer.getNumInputs(); k++) {
                    sum += currentLayer.getWeights()[k] * predictors[k %
currentLayer.getNumInputs()];
                }
                double activation = (1 / (1 + Math.exp(sum * -1)));
                activations[j] = activation;
            }
            predictors = activations;
            allActivations[i] = activations;
        }
        return allActivations;
    }

    private double[][] backwardPass(double observedValue, double[][]
activations){
        double[][] allDeltas = new double[this.layers.length][]; //save all
the delta values
        for (int i = this.layers.length - 1; i >= 0; i--) {
            Layer currentLayer = this.layers[i];
            double[] deltas = new double[currentLayer.getNumNodes()];
            if (i == this.layers.length - 1) { //calculate delta if output
node
                for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                    double activation = activations[i][j];
                    double delta = (observedValue - activation) *
(activation * (1 - activation));
                    deltas[j] = delta;
                }
            } else { //calculate delta if hidden node
                for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                    Layer nextLayer = this.layers[i + 1];
                    double sumDeltasMultWeights = 0;
                    double activation = activations[i][j];
                    for (int k = 0; k < nextLayer.getNumNodes(); k++) {
                        sumDeltasMultWeights += (allDeltas[i + 1][k] *
nextLayer.getWeights()[(k *currentLayer.getNumNodes()) + j]);
                    }
                    double delta = sumDeltasMultWeights * (activation * (1
- activation));
                    deltas[j] = delta;
                }
            }
            allDeltas[i] = deltas;
        }
        return allDeltas;
    }

    private void updateVariables(double lp, double[][] activations,
double[][] deltas){
        for (int i = 0; i < this.layers.length; i++) {
            Layer currentLayer = this.layers[i];
```

```
            for (int j = 0; j < currentLayer.getWeights().length; j++) {
                double activation = activations[i][j %
currentLayer.getNumNodes()];
                double delta = deltas[i][j % currentLayer.getNumNodes()];
                double currentWeightChange = (lp * delta * activation);
                currentLayer.getWeights()[j] += currentWeightChange;
            }
            for (int j = 0; j < currentLayer.getBiases().length; j++) {
                double activation = activations[i][j %
currentLayer.getNumNodes()];
                double delta = deltas[i][j % currentLayer.getNumNodes()];
                double currentWeightChange = (lp * delta * activation);
                currentLayer.getBiases()[j] += currentWeightChange;
            }
        }
    }

    public void train(int epochs, double lp, double[][] trainingSet,
double[][] validationSet, double[][] testSet, String[] testDates){
        double previousTrainingRMSE = Double.MAX_VALUE;
        double previousValidationRMSE = Double.MAX_VALUE;
        for(int i = 0; i < epochs; i++) {

            double trainingTotalError = 0;
            double trainingRMSE;
            for (int j = 0; j < trainingSet.length; j++) { //go through the
training set
                double[] predictors = new double[trainingSet[j].length-1];
                for (int k = 0; k < trainingSet[j].length-1; k++){
                    predictors[k] = trainingSet[j][k];
                }
                double[][] activations = forwardPass(predictors); //forward
pass
                double observedValue =
trainingSet[j][trainingSet[j].length-1];
                double modelledValue = activations[activations.length-
1][0];
                trainingTotalError += (modelledValue-
observedValue)*(modelledValue-observedValue);
                double[][] deltas = backwardPass(observedValue,
activations); //backward pass
                updateVariables(lp, activations, deltas); //update
variables
            }
            trainingRMSE =
Math.sqrt(trainingTotalError/trainingSet.length); //calculate training RMSE
            System.out.println("Epoch no: " + i + ", Training RMSE: " +
trainingRMSE);

            try { //save the error and epoch to txt so can draw a graph
showing how the error changed over time
                FileWriter writer = new FileWriter("ErrorVEpochLp" + lp
+".txt", true);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(i + ", " + trainingRMSE + "\n");
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            if(i % 1000 == 0) { //check to see if training should stop
```

```
every 1000 epochs
                double validationTotalError = 0;
                double validationRMSE;
                for (int j = 0; j < validationSet.length; j++) {
                    double[] predictors = new
double[validationSet[j].length-1];
                    for (int k = 0; k < validationSet[j].length-1; k++){
                        predictors[k] = validationSet[j][k];
                    }
                    double[][] activations = forwardPass(predictors);
                    double observedValue =
validationSet[j][validationSet[j].length-1];
                    double modelledValue = activations[activations.length-
1][0];
                    validationTotalError += (modelledValue-
observedValue)*(modelledValue-observedValue);
                }
                validationRMSE =
Math.sqrt(validationTotalError/validationSet.length); //calculate
validation RMSE
                if (validationRMSE > previousValidationRMSE && trainingRMSE
< previousTrainingRMSE) { //check if validation error has increased but
training error hasn't (over-training)
                    System.out.println("Terminated training after " + i + "
epochs as validation error began to increase");
                    this.test(testSet, testDates);
                    return;
                }
                previousValidationRMSE = validationRMSE;
                previousTrainingRMSE = trainingRMSE;
            }

        } //completed all the epochs
        System.out.println("Completed Training");
        this.test(testSet, testDates);
    }

    private void test(double[][] testSet, String[] testDates){
        double totalTestError = 0;
        for (int j = 0; j < testSet.length; j++) {
            double[] predictors = new double[testSet[j].length-1];
            for (int k = 0; k < testSet[j].length-1; k++){
                predictors[k] = testSet[j][k];
            }
            String date = testDates[j];
            double observedValue = testSet[j][testSet[j].length-1];
            double modelledValue = predict(predictors);
            totalTestError += (modelledValue-observedValue)*(modelledValue-
observedValue);
            try {
                FileWriter writer = new
FileWriter("ObservedVsModelled.txt", true);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(observedValue + ", " + modelledValue +
"," + date + "\n");
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        double testRMSE = Math.sqrt(totalTestError/testSet.length);
```

```
        System.out.println("Test RMSE: " + testRMSE);
        System.out.println("Finished Testing");
    }

    public double predict(double[] predictors){
        double[][] activations = forwardPass(predictors);
        return activations[activations.length-1][0];
    }

}
```

All improvements (momentum, annealing and weight decay)

```
import java.io.*;

public class ANN {
    private Layer[] layers;
    private double[][] previousWeightChanges;
    private double[][] previousBiasChanges;

    public ANN(int[] layerSizes){
        Layer[] layers = new Layer[layerSizes.length-1];
        for(int i = 0; i < layerSizes.length-1; i++){
            int numWeights = layerSizes[i]*layerSizes[i+1];
            int numBiases = layerSizes[i+1];
            Layer currentLayer = new Layer(numWeights, numBiases,
layerSizes[i+1], layerSizes[i]);
            layers[i] = currentLayer;
        }
        this.layers = layers;

        //momentum stuff - creates an array which holds the previous weight
changes
        double[][] previousWeightChanges = new
double[this.layers.length][];
        double[][] previousBiasChanges = new double[this.layers.length][];
        for(int i = 0; i < this.layers.length; i++){
            double[] prevWeightChangesL = new
double[this.layers[i].getWeights().length];
            double[] prevBiasChangesL = new
double[this.layers[i].getBiases().length];
            for(int j = 0; j < this.layers[i].getWeights().length; j++){
                prevWeightChangesL[j] = 0;
            }
            for(int j = 0; j < this.layers[i].getBiases().length; j++){
                prevBiasChangesL[j] = 0;
            }
            previousWeightChanges[i] = prevWeightChangesL;
            previousBiasChanges[i] = prevBiasChangesL;
        }
        this.previousWeightChanges = previousWeightChanges;
        this.previousBiasChanges = previousBiasChanges;
    }

    public double getOmega(){ //for use in weight decay
        double sum = 0;
        double count = 0;
        for(int i = 0; i < this.layers.length; i++){
            Layer currentLayer = this.layers[i];
            for(int j = 0; j < currentLayer.getWeights().length; j++){
                sum +=
```

```
(currentLayer.getWeights()[j]*currentLayer.getWeights()[j]);
                count++;
            }
            for(int j = 0; j < currentLayer.getBiases().length; j++){
                sum +=
(currentLayer.getBiases()[j]*currentLayer.getBiases()[j]);
                count++;
            }
        }
        return (1/(2*count))*sum;
    }

    private double[][] forwardPass(double[] predictors){
        double[][] allActivations = new double[this.layers.length][];
//save all the activation values
        for (int i = 0; i < this.layers.length; i++) {
            Layer currentLayer = this.layers[i];
            double[] activations = new double[currentLayer.getNumNodes()];
            for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                double sum = currentLayer.getBiases()[j];
                for (int k = j * currentLayer.getNumInputs(); k < (j + 1) *
currentLayer.getNumInputs(); k++) {
                    sum += currentLayer.getWeights()[k] * predictors[k %
currentLayer.getNumInputs()];
                }
                double activation = (1 / (1 + Math.exp(sum * -1)));
                activations[j] = activation;
            }
            predictors = activations;
            allActivations[i] = activations;
        }
        return allActivations;
    }

    private double[][] backwardPass(double observedValue, double[][]
activations, double upsilon){
        double[][] allDeltas = new double[this.layers.length][]; //save all
the delta values
        for (int i = this.layers.length - 1; i >= 0; i--) {
            Layer currentLayer = this.layers[i];
            double[] deltas = new double[currentLayer.getNumNodes()];
            if (i == this.layers.length - 1) { //calculate delta if output
node
                for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                    double activation = activations[i][j];
                    double delta = ((observedValue - activation) +
(getOmega() * upsilon)) * (activation * (1 - activation)); //with weight
decay
                    deltas[j] = delta;
                }
            } else { //calculate delta if hidden node
                for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                    Layer nextLayer = this.layers[i + 1];
                    double sumDeltasMultWeights = 0;
                    double activation = activations[i][j];
                    for (int k = 0; k < nextLayer.getNumNodes(); k++) {
                        sumDeltasMultWeights += (allDeltas[i + 1][k] *
nextLayer.getWeights()[(k *currentLayer.getNumNodes()) + j]);
                    }
                    double delta = sumDeltasMultWeights * (activation * (1
- activation));
```

```
                    deltas[j] = delta;
                }
            }
            allDeltas[i] = deltas;
        }
        return allDeltas;
    }

    private void updateVariables(double lp, double[][] activations,
double[][] deltas){
        for (int i = 0; i < this.layers.length; i++) {
            Layer currentLayer = this.layers[i];
            for (int j = 0; j < currentLayer.getWeights().length; j++) {
                double activation = activations[i][j %
currentLayer.getNumNodes()];
                double delta = deltas[i][j % currentLayer.getNumNodes()];
                double prevWeightChange = this.previousWeightChanges[i][j];
//get previous weight change to help calculate momentum
                double currentWeightChange = (lp * delta * activation) +
(0.9 * prevWeightChange);
                currentLayer.getWeights()[j] += currentWeightChange;
                this.previousWeightChanges[i][j] = currentWeightChange;
//save current weight change so can be used next time for momentum
            }
            for (int j = 0; j < currentLayer.getBiases().length; j++) {
                double activation = activations[i][j %
currentLayer.getNumNodes()];
                double delta = deltas[i][j % currentLayer.getNumNodes()];
                double prevWeightChange = this.previousBiasChanges[i][j];
//get previous weight change to help calculate momentum
                double currentWeightChange = (lp * delta * activation) +
(0.9 * prevWeightChange);
                currentLayer.getBiases()[j] += currentWeightChange;
                this.previousBiasChanges[i][j] = currentWeightChange;
//save current weight change so can be used next time for momentum
            }
        }
    }

    public void train(int epochs, double lp, double[][] trainingSet,
double[][] validationSet, double[][] testSet, String[] testDates){
        double previousTrainingRMSE = Double.MAX_VALUE;
        double previousValidationRMSE = Double.MAX_VALUE;
        double startLp = lp; //for annealing
        double endLp = 0.01; //for annealing
        for(int i = 0; i < epochs; i++) {

            lp = endLp + ((startLp - endLp) * (1 - (1 / (1 + Math.exp(10.0
- ((20.0 * i) / epochs)))))); //simulated annealing
            double trainingTotalError = 0;
            double trainingRMSE;
            for (int j = 0; j < trainingSet.length; j++) { //go through the
training set
                double[] predictors = new double[trainingSet[j].length-1];
                for (int k = 0; k < trainingSet[j].length-1; k++){
                    predictors[k] = trainingSet[j][k];
                }
                double[][] activations = forwardPass(predictors); //forward
pass
                double observedValue =
trainingSet[j][trainingSet[j].length-1];
```

```
                double modelledValue = activations[activations.length-
1][0];
                trainingTotalError += (modelledValue-
observedValue)*(modelledValue-observedValue);
                double upsilon = 1/((i+1)*lp); //calculate upsilon for
weight decay
                double[][] deltas = backwardPass(observedValue,
activations, upsilon); //backward pass
                updateVariables(lp, activations, deltas); //update
variables
            }
            trainingRMSE =
Math.sqrt(trainingTotalError/trainingSet.length); //calculate training RMSE
            System.out.println("Epoch no: " + i + ", Training RMSE: " +
trainingRMSE);

            try { //save the error and epoch to txt so can draw a graph
showing how the error changed over time
                FileWriter writer = new FileWriter("ErrorVEpochLp" +
startLp +".txt", true);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(i + ", " + trainingRMSE + "\n");
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            if(i % 1000 == 0) { //check to see if training should stop
every 1000 epochs
                double validationTotalError = 0;
                double validationRMSE;
                for (int j = 0; j < validationSet.length; j++) {
                    double[] predictors = new
double[validationSet[j].length-1];
                    for (int k = 0; k < validationSet[j].length-1; k++){
                        predictors[k] = validationSet[j][k];
                    }
                    double[][] activations = forwardPass(predictors);
                    double observedValue =
validationSet[j][validationSet[j].length-1];
                    double modelledValue = activations[activations.length-
1][0];
                    validationTotalError += (modelledValue-
observedValue)*(modelledValue-observedValue);
                }
                validationRMSE =
Math.sqrt(validationTotalError/validationSet.length); //calculate
validation RMSE
                if (validationRMSE > previousValidationRMSE && trainingRMSE
< previousTrainingRMSE) { //check if validation error has increased but
training error hasn't (over-training)
                    System.out.println("Terminated training after " + i + "
epochs as validation error began to increase");
                    this.test(testSet, testDates);
                    return;
                }
                previousValidationRMSE = validationRMSE;
                previousTrainingRMSE = trainingRMSE;
            }

        } //completed all the epochs
```

```
        System.out.println("Completed Training");
        this.test(testSet, testDates);
    }

    private void test(double[][] testSet, String[] testDates){
        double totalTestError = 0;
        for (int j = 0; j < testSet.length; j++) {
            double[] predictors = new double[testSet[j].length-1];
            for (int k = 0; k < testSet[j].length-1; k++){
                predictors[k] = testSet[j][k];
            }
            double[][] activations = forwardPass(predictors);
            String date = testDates[j];
            double observedValue = testSet[j][testSet[j].length-1];
            double modelledValue = activations[activations.length-1][0];
            totalTestError += (modelledValue-observedValue)*(modelledValue-
observedValue);
            try {
                FileWriter writer = new
FileWriter("ObservedVsModelled.txt", true);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(observedValue + ", " + modelledValue +
"," + date + "\n");
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        double testRMSE = Math.sqrt(totalTestError/testSet.length);
        System.out.println("Test RMSE: " + testRMSE);
        System.out.println("Finished Testing");
    }

    public double[] predict(double[] predictors){
        double[][] activations = forwardPass(predictors);
        return activations[activations.length-1];
    }

}
```

Final version (momentum and weight decay)

```
import java.io.*;

public class ANN {
    private Layer[] layers;
    private double[][] previousWeightChanges;
    private double[][] previousBiasChanges;

    public ANN(int[] layerSizes){
        Layer[] layers = new Layer[layerSizes.length-1];
        for(int i = 0; i < layerSizes.length-1; i++){
            int numWeights = layerSizes[i]*layerSizes[i+1];
            int numBiases = layerSizes[i+1];
            Layer currentLayer = new Layer(numWeights, numBiases,
layerSizes[i+1], layerSizes[i]);
            layers[i] = currentLayer;
        }
        this.layers = layers;
```

```
        //momentum stuff - creates an array which holds the previous weight
changes
        double[][] previousWeightChanges = new
double[this.layers.length][];
        double[][] previousBiasChanges = new double[this.layers.length][];
        for(int i = 0; i < this.layers.length; i++){
            double[] prevWeightChangesL = new
double[this.layers[i].getWeights().length];
            double[] prevBiasChangesL = new
double[this.layers[i].getBiases().length];
            for(int j = 0; j < this.layers[i].getWeights().length; j++){
                prevWeightChangesL[j] = 0;
            }
            for(int j = 0; j < this.layers[i].getBiases().length; j++){
                prevBiasChangesL[j] = 0;
            }
            previousWeightChanges[i] = prevWeightChangesL;
            previousBiasChanges[i] = prevBiasChangesL;
        }
        this.previousWeightChanges = previousWeightChanges;
        this.previousBiasChanges = previousBiasChanges;
    }

    public double getOmega(){ //for use in weight decay
        double sum = 0;
        double count = 0;
        for(int i = 0; i < this.layers.length; i++){
            Layer currentLayer = this.layers[i];
            for(int j = 0; j < currentLayer.getWeights().length; j++){
                sum +=
(currentLayer.getWeights()[j]*currentLayer.getWeights()[j]);
                count++;
            }
            for(int j = 0; j < currentLayer.getBiases().length; j++){
                sum +=
(currentLayer.getBiases()[j]*currentLayer.getBiases()[j]);
                count++;
            }
        }
        return (1/(2*count))*sum;
    }

    private double[][] forwardPass(double[] predictors){
        double[][] allActivations = new double[this.layers.length][];
//save all the activation values
        for (int i = 0; i < this.layers.length; i++) {
            Layer currentLayer = this.layers[i];
            double[] activations = new double[currentLayer.getNumNodes()];
            for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                double sum = currentLayer.getBiases()[j];
                for (int k = j * currentLayer.getNumInputs(); k < (j + 1) *
currentLayer.getNumInputs(); k++) {
                    sum += currentLayer.getWeights()[k] * predictors[k %
currentLayer.getNumInputs()];
                }
                double activation = (1 / (1 + Math.exp(sum * -1)));
                activations[j] = activation;
            }
            predictors = activations;
            allActivations[i] = activations;
```

```
        }
        return allActivations;
    }

    private double[][] backwardPass(double observedValue, double[][]
activations, double upsilon){
        double[][] allDeltas = new double[this.layers.length][]; //save all
the delta values
        for (int i = this.layers.length - 1; i >= 0; i--) {
            Layer currentLayer = this.layers[i];
            double[] deltas = new double[currentLayer.getNumNodes()];
            if (i == this.layers.length - 1) { //calculate delta if output
node
                for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                    double activation = activations[i][j];
                    double delta = ((observedValue - activation) +
(getOmega() * upsilon)) * (activation * (1 - activation)); //with weight
decay
                    deltas[j] = delta;
                }
            } else { //calculate delta if hidden node
                for (int j = 0; j < currentLayer.getNumNodes(); j++) {
                    Layer nextLayer = this.layers[i + 1];
                    double sumDeltasMultWeights = 0;
                    double activation = activations[i][j];
                    for (int k = 0; k < nextLayer.getNumNodes(); k++) {
                        sumDeltasMultWeights += (allDeltas[i + 1][k] *
nextLayer.getWeights()[(k *currentLayer.getNumNodes()) + j]);
                    }
                    double delta = sumDeltasMultWeights * (activation * (1
- activation));
                    deltas[j] = delta;
                }
            }
            allDeltas[i] = deltas;
        }
        return allDeltas;
    }

    private void updateVariables(double lp, double[][] activations,
double[][] deltas){
        for (int i = 0; i < this.layers.length; i++) {
            Layer currentLayer = this.layers[i];
            for (int j = 0; j < currentLayer.getWeights().length; j++) {
                double activation = activations[i][j %
currentLayer.getNumNodes()];
                double delta = deltas[i][j % currentLayer.getNumNodes()];
                double prevWeightChange = this.previousWeightChanges[i][j];
//get previous weight change to help calculate momentum
                double currentWeightChange = (lp * delta * activation) +
(0.9 * prevWeightChange);
                currentLayer.getWeights()[j] += currentWeightChange;
                this.previousWeightChanges[i][j] = currentWeightChange;
//save current weight change so can be used next time for momentum
            }
            for (int j = 0; j < currentLayer.getBiases().length; j++) {
                double activation = activations[i][j %
currentLayer.getNumNodes()];
                double delta = deltas[i][j % currentLayer.getNumNodes()];
                double prevWeightChange = this.previousBiasChanges[i][j];
//get previous weight change to help calculate momentum
```

```
                double currentWeightChange = (lp * delta * activation) +
(0.9 * prevWeightChange);
                currentLayer.getBiases()[j] += currentWeightChange;
                this.previousBiasChanges[i][j] = currentWeightChange;
//save current weight change so can be used next time for momentum
            }
        }
    }

    public void train(int epochs, double lp, double[][] trainingSet,
double[][] validationSet, double[][] testSet, String[] testDates){
        double previousTrainingRMSE = Double.MAX_VALUE;
        double previousValidationRMSE = Double.MAX_VALUE;
        for(int i = 0; i < epochs; i++) {

            double trainingTotalError = 0;
            double trainingRMSE;
            for (int j = 0; j < trainingSet.length; j++) { //go through the
training set
                double[] predictors = new double[trainingSet[j].length-1];
                for (int k = 0; k < trainingSet[j].length-1; k++){
                    predictors[k] = trainingSet[j][k];
                }
                double[][] activations = forwardPass(predictors); //forward
pass
                double observedValue =
trainingSet[j][trainingSet[j].length-1];
                double modelledValue = activations[activations.length-
1][0];
                trainingTotalError += (modelledValue-
observedValue)*(modelledValue-observedValue);
                double upsilon = 1/((i+1)*lp); //calculate upsilon for
weight decay
                double[][] deltas = backwardPass(observedValue,
activations, upsilon); //backward pass
                updateVariables(lp, activations, deltas); //update
variables
            }
            trainingRMSE =
Math.sqrt(trainingTotalError/trainingSet.length); //calculate training RMSE
            System.out.println("Epoch no: " + i + ", Training RMSE: " +
trainingRMSE);

            try { //save the error and epoch to txt so can draw a graph
showing how the error changed over time
                FileWriter writer = new FileWriter("ErrorVEpochLp" + lp
+".txt", true);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(i + ", " + trainingRMSE + "\n");
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            if(i % 1000 == 0) { //check to see if training should stop
every 1000 epochs
                double validationTotalError = 0;
                double validationRMSE;
                for (int j = 0; j < validationSet.length; j++) {
                    double[] predictors = new
double[validationSet[j].length-1];
```

```
                    for (int k = 0; k < validationSet[j].length-1; k++){
                        predictors[k] = validationSet[j][k];
                    }
                    double[][] activations = forwardPass(predictors);
                    double observedValue =
validationSet[j][validationSet[j].length-1];
                    double modelledValue = activations[activations.length-
1][0];
                    validationTotalError += (modelledValue-
observedValue)*(modelledValue-observedValue);
                }
                validationRMSE =
Math.sqrt(validationTotalError/validationSet.length); //calculate
validation RMSE
                if (validationRMSE > previousValidationRMSE && trainingRMSE
< previousTrainingRMSE) { //check if validation error has increased but
training error hasn't (over-training)
                    System.out.println("Terminated training after " + i + "
epochs as validation error began to increase");
                    this.test(testSet, testDates);
                    return;
                }
                previousValidationRMSE = validationRMSE;
                previousTrainingRMSE = trainingRMSE;
            }

        } //completed all the epochs
        System.out.println("Completed Training");
        this.test(testSet, testDates);
    }

    private void test(double[][] testSet, String[] testDates){
        double totalTestError = 0;
        for (int j = 0; j < testSet.length; j++) {
            double[] predictors = new double[testSet[j].length-1];
            for (int k = 0; k < testSet[j].length-1; k++){
                predictors[k] = testSet[j][k];
            }
            double[][] activations = forwardPass(predictors);
            String date = testDates[j];
            double observedValue = testSet[j][testSet[j].length-1];
            double modelledValue = activations[activations.length-1][0];
            totalTestError += (modelledValue-observedValue)*(modelledValue-
observedValue);
            try {
                FileWriter writer = new
FileWriter("ObservedVsModelled.txt", true);
                BufferedWriter bufferedWriter = new BufferedWriter(writer);
                bufferedWriter.write(observedValue + ", " + modelledValue +
"," + date + "\n");
                bufferedWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        double testRMSE = Math.sqrt(totalTestError/testSet.length);
        System.out.println("Test RMSE: " + testRMSE);
        System.out.println("Finished Testing");
    }

    public double[] predict(double[] predictors){
```

```
        double[][] activations = forwardPass(predictors);
        return activations[activations.length-1];
    }

}
```