AI Report Neural Network Coursework

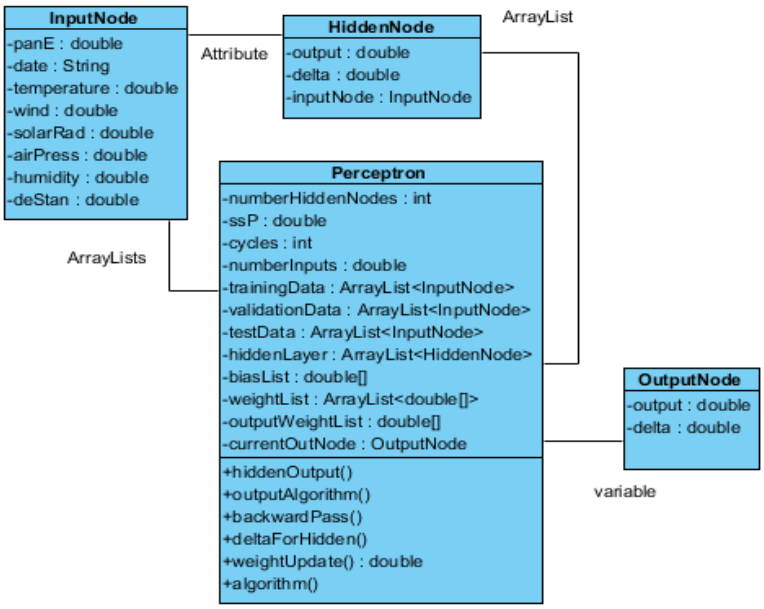
Implementation

Language used

I chose to use the language Java because I thought of the network in an object orientated way and believe Java implements this better than languages like Python. I have the most experience in this language so can implement the algorithm better than I could C# or C++.

Implementation – how

I used an **object orientated** approach where each input node, hidden node and output node is a class that can be used as a data type in the perceptron class (ANN class).



An **input node** has attributes which are columns of **data** given (standardised and cleaned) so temperature, wind etc. I also included attribute '**deStan'** which is the de-standardised evaporation value so that I can use it for comparison later in the algorithm.

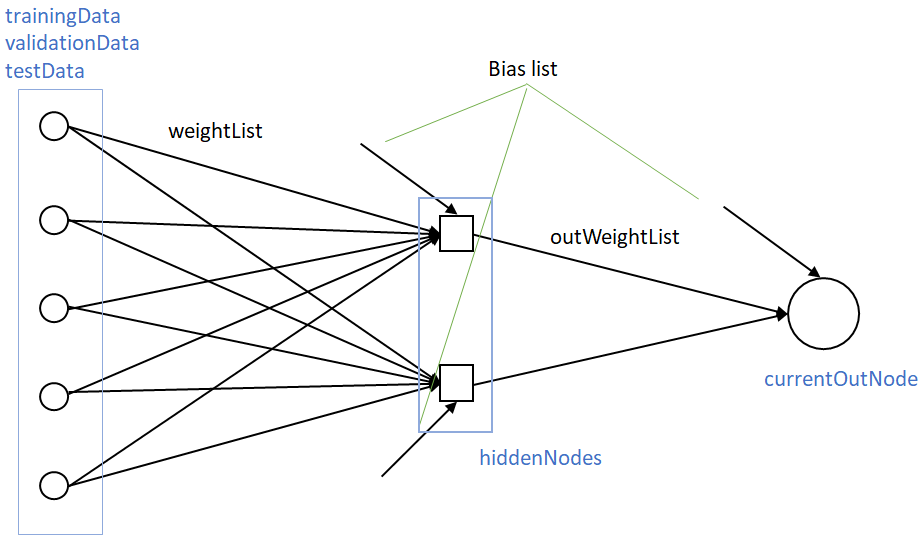
A **hidden node** uses the **input node** class as an attribute because then when running the algorithm an input node can be passed for each cycle of a row. A hidden node also has a **delta** value for updating weights between nodes and an **output** value for the output node.

An **output node** has attributes **output** (final output of weighted sums and bias for the node) and **delta** (for backward pass weight calculation). Only 1 output node is essential because only 1 output value is wanted.

<u>The **Perceptron** class</u>

Variable attributes:

- **numberHiddenNodes** is the number of hidden nodes I would like when running, this can vary easily for testing
- **ssP** is single step parameter for the learning rate
- **cycles** is the number of epochs the program goes through
- **numberInputs** – for this ANN this will always be 5



AI Report Neural Network Coursework

Storing data:

- **trainingData** – 60% of the data set for training with the algorithm
- **validationData** – 20% of data set for validating the algorithm (run periodically)
- **testData** – 20% of data for testing outputs at the end of the algorithm

} ArrayList<InputNode>

I used an ArrayList to store these values because they are not of fixed length, so I can add values as I import the data. This data type enabled me to make lists of the specific data types that I had created and so I could read and write values of the array.

- **hiddenLayer** – ArrayList of HiddenNodes
- **weightList** – Array list of double lists – each hidden node will have a list of weights going into it from the input layer (a list the size of the number of hidden nodes with each of them having a list of doubles which is the number of inputs in length)
- **biasList** – List of biases for the hidden layer and end value for the output layer
- **outputWeightList** – a list of weights from the hidden layer to the output node
- **currentOutNode** – is the output node (of type OutputNode) which changes values for each cycle (row of data)

Code flexibility

I created my code so it is flexible to be able to change the number of hidden nodes, cycles and ssP number for each run. This was effective since I could perform lots of different tests and see how is best to run it.

The code will only work for standardised data with 5 inputs and 1 output.

MLP algorithm

Main Loop – algorithm method

```
for (int h = 0; h<cycles; h++) {
	algorithm(trainingData, false);
	if (h!=0 && h%1000 == 0) {
		prevRMSE = validationRMSE;
		algorithm(validationData, true);
	}
}
```

The algorithm is run as follows:
For all cycles, the algorithm (essentially forwards pass, backwards pass and weight update) runs on training data, then periodically it runs on validation data to ensure I don't over-train the data to particular results. Note, when testing for every 500, 1000 and 2500 cycles for a total of 100000 cycles, the RMSE changed an insignificant amount.

After this I tested the network with test data, calculating a separate root mean squared error value for test data and outputting the results.

Step 1. Forward pass:

Firstly, I computed the output value of each hidden node. Since I have stored the hidden nodes in an array with input node as an attribute, I used the current input (temperature, wind etc.) multiplied by the corresponding weight in the hidden node weights. I then added on the **bias** for that node to get Si (**weighted sum**) value which I used in the activation function to change the output value of the hidden node.

```
public static void hiddenOutput(HiddenNode hiddenNode) {
	int indexOfHiddenNode = hiddenLayer.indexOf(hiddenNode);
	double si = (hiddenNode.inputNode.temperature * weightList.get(indexOfHiddenNode)[0]) +
(hiddenNode.inputNode.wind * weightList.get(indexOfHiddenNode)[1])
			+ (hiddenNode.inputNode.solarRad * weightList.get(indexOfHiddenNode)[2]) +
(hiddenNode.inputNode.airPress * weightList.get(indexOfHiddenNode)[3])
			+ (hiddenNode.inputNode.humidity * weightList.get(indexOfHiddenNode)[4]) +
biasList[indexOfHiddenNode];
	hiddenNode.output = 1/(1 +Math.exp(-si));
}
```

Step 2. End output:

Then I iterate through each training (or validation) example with correct output `panE` to work out the output from the forward pass. This is the function `outputAlgorithm()` which gets the sum of the hidden nodes output multiplied by the corresponding weight plus the bias for the output node. Finally, for **the output of the forward pass** at the output node I do `(1/(1 +Math.exp(-si)))` where si is the weighted sum calculated above which I take the negative exponential of.

```
public static void outputAlgorithm() {
	double si = 0;
	for (int n = 0; n<hiddenLayer.size(); n ++) {
		si += hiddenLayer.get(n).output*outputWeightList2[n];
	}
	si +=biasList[biasList.length-1];
	currentOutNode.output = (1/(1 +Math.exp(-si)));
}
```

Step 3. Backward pass computing for each node:

- First I calculated **delta δ** for the output cell with `backwardPass()`. fS is the activation function which is equal to CO * (1 – CO) where CO is the current output calculated in step 2. The delta for the output node is then calculated using this **activation function** value fS, `correctPanE` which is the actual/correct (standardised) value of evaporation and `currentOutNode.output` which is the current output of the network. Overall the delta of the output node equals the correct evaporation – modelled evaporation all multiplied by the activation function.

```
public static void backwardPass( double correctPanE) {
	double fS = currentOutNode.output * (1 - currentOutNode.output);
	//delta = (correct output - current output) * (current output (1 - current output))
	currentOutNode.delta = (correctPanE - currentOutNode.output) * fS;
}
```

- I then iterate over the hidden layer so that for each node the function `deltaForHidden` is executed; this calculates the delta value for the selected hidden node in the hidden layer. First I calculated the activation function value (fS) which same as before takes the output of that node (say ON) and equals ON * (1-ON). To calculate the delta I use the output weight of that node, times the delta of the output node and fS.

```
public static void deltaForHidden(HiddenNode hiddenNode) {
	double fS = hiddenNode.output * (1 - hiddenNode.output );
	hiddenNode.delta = (outputWeightList2[hiddenLayer.indexOf(hiddenNode)] *
	currentOutNode.delta ) * fS;
}
```

- Within the iteration over the hidden layer all weights are updated with *weightUpdate* function which has parameters of the current weight on that edge, the current delta value of its end point and the input value into the weight.
- **Step size parameter** is also used here to train the algorithm and find optimum (minimum error) within the search space.

```
public static double weightUpdate(double currentWeight,double currentDelta, double inputValue) {
            return currentWeight + (ssP *  currentDelta *inputValue);
      }
```

Finally, I tested the data using test data at the end using RMSE with the de-standardised data.

Training and network selection

Number of hidden nodes

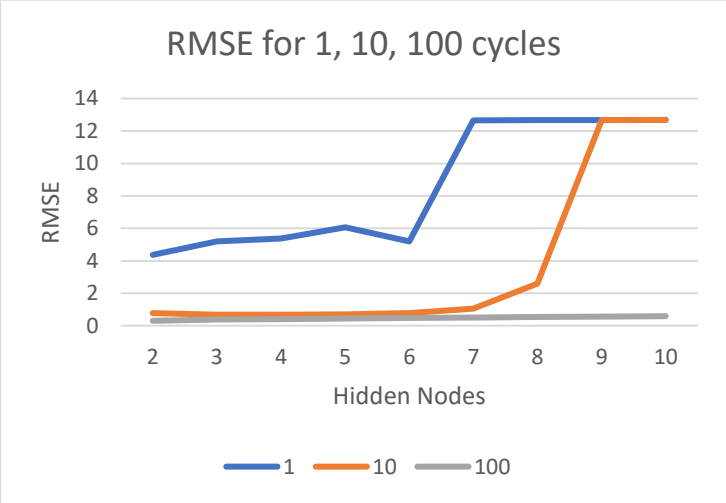
I selected the number of hidden nodes to test on for [n/2, 2n] where n is the number of inputs. I first tested how the RMSE changed on very low numbers of epochs because it is clear that a higher number of epochs produces less accurate results.
After this discovery I tested for node numbers between 2 and 6.

Epochs

To test the optimum number of epochs I ran this while loop so that on the validation data the RMSE is calculated and if it hasn't improved then don't cycle any further. I reached 217000 cycles. I tested this number a few more times in a for loop (stopping when reaching 217000) and the RMSE was consistently good.

```
while (prevRMSE >= validationRMSE && cycles<1000000) {
      cycles++;
      algorithm(trainingData, false);
      if (cycles!=0 && cycles%1000 == 0) {
            prevRMSE = validationRMSE;
            algorithm(validationData, true);
      }
}
```

```
Cycles,217000
Hidden Nodes,2
RMSE,0.28173315452944314
ssp,0.1
```



Modifications and Improvements

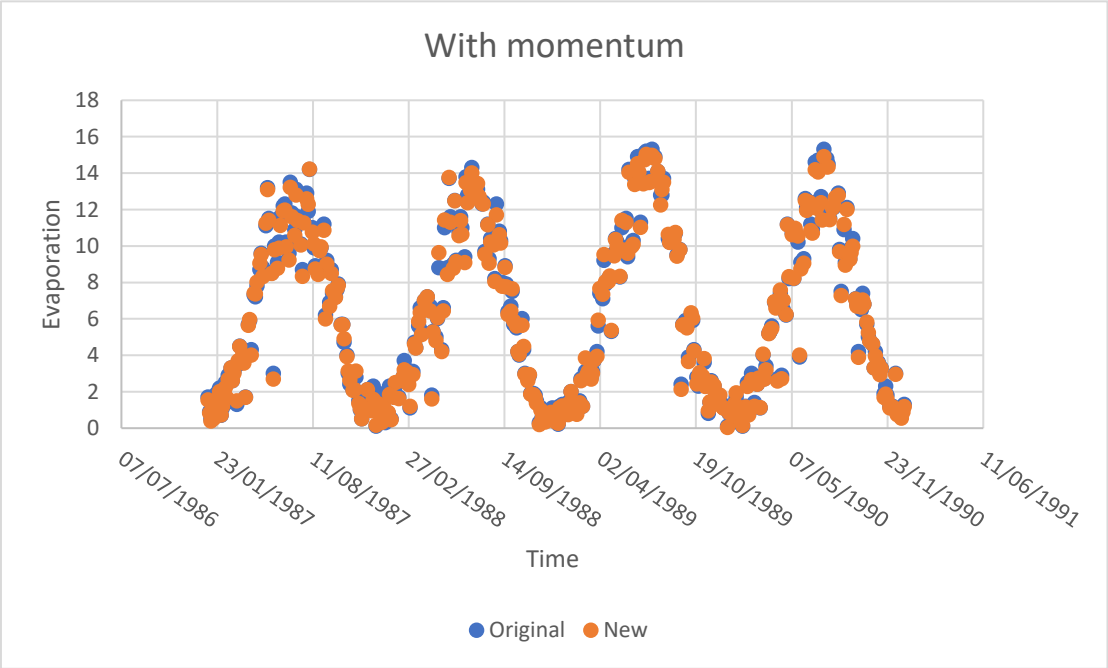
Momentum

With momentum added in weight change, the results were mixed in terms of better RMSE with/without momentum however showed the reasonable trend that it worked better with a smaller step size parameter.

```
public static double weightUpdate(double currentWeight,double currentDelta, double inputValue) {
      double newWeight = currentWeight + (ssP *  currentDelta *inputValue);
      double change = newWeight - currentWeight;
      return newWeight + omega * change;
}
```

Change	With momentum	Without momentum
-	Cycles,217000 Hidden Nodes,2 RMSE,0.276506132240619 ssp,0.6	Cycles,217000 Hidden Nodes,2 RMSE,0.2383374884835927 ssp,0.6
ssP	Cycles,217000 Hidden Nodes,2 RMSE,0.2223140277725283 ssp,0.1	Cycles,217000 Hidden Nodes,2 RMSE,0.22232111723061007 ssp,0.1
Cycles	Cycles,100000 Hidden Nodes,2 RMSE,0.22454017146158942 ssp,0.1	Cycles,100000 Hidden Nodes,2 RMSE,0.28258880526501956 ssp,0.1

This graph shows a strong clear correlation between predicted and actual evaporation levels.



Bold driver

Only used when using validation data therefore is only every 1000 epochs.

Before implementing

```
Cycles,100000
Hidden Nodes,2
RMSE,0.22335711832300212
ssp,0.1
```

Before implementing bold driver I had a relatively successful RMSE of 0.223.

First implementation

```
Cycles,100000
Hidden Nodes,2
RMSE,0.30081171912196364
ssp,6.024006916124239
```

```
if (prevRMSE >validationRMSE) {
        ssP*=1.1;
}
```

The first implementation had an RMSE of 0.3008 which relatively wasn't too bad but definitely didn't improve the results (compared to 0.223). This if statement shows if the previous RMSE is better than the validation then ssP should be 110% of what it is currently.

AI Report Neural Network Coursework

With restrictions

```
Cycles,100000
Hidden Nodes,2
RMSE,0.27296728217661703
ssp,0.45949729863572203
```

```
if (prevRMSE >validationRMSE &&
ssP*1.1<0.5)
```

I added a restriction to the size of ssP (when increasing) so that there wasn't a snowballing effect; too high a number may cause training to oscillate whereas to low may cause it to become trapped in a local minima. The results improved after implementing the limit.

With increasing RMSE

```
Cycles,100000
Hidden Nodes,2
RMSE,0.28559170355402297
ssp,1.1129715058067E-24
```

```
if (h!=0 && h%1000 == 0) {
	prevRMSE = validationRMSE;
	algorithm(validationData, true);
	if (prevRMSE > validationRMSE && ssP*1.1<0.5) {
		ssP *=1.1;
	} else {
		ssP = prevSSP*0.5;
	}
}
```

The result of RMSE have risen slightly since the previous test without this, however I noticed how small the step parameter value is here so then tested with restrictions to ensure the ssP didn't reach too small.

Increasing RMSE - with limits

```
Cycles,100000
Hidden Nodes,2
RMSE,0.2771559482794238
ssp,0.10001954172780657
```

```
else if (prevSSP*0.5 > 0.01){
```

The result has reduced slightly again however all of these results have not been as good as without bold driver implemented.

Annealing

Without annealing

```
Cycles,100000
Hidden Nodes,2
RMSE,0.2770564461641548
ssp,0.1
```

$$f(x) = p + (q - p)\left(1 - \frac{1}{1 + e^{10 - \frac{20x}{r}}}\right)$$

I modified the algorithm, trying annealing for the step parameter with the formula above. Which I implemented as follows where h is the count for the current amount of epochs so far, 0.01 is the minimum ssP and 0.1 is the maximum ssP.

```
ssP = 0.01 + (0.1 - 0.01)*(1-(1/ 1+ Math.exp(10 -((20*h) / cycles))));
```

```
Cycles,100000
Hidden Nodes,2
RMSE,10.231518991224025
ssp,0.009988893117632193
```

Evidently this was a very unsuccessful with a very high RMSE; after observing changing ssP, I saw it changed extremely dramatically.

I then implemented an if statement so that there was a limit to how low or high the step parameter could go using this if statement:

```
Cycles,10000
Hidden Nodes,2
RMSE,0.27685445614779
ssp,0.09987041970570
```

```
if (0.01 + (0.1 - 0.01)*(1-(1/ 1+ Math.exp(10 -((20*h) / cycles)))) <0.1 && 0.01 + (0.1 - 0.01)*(1-(1/ 1+ Math.exp(10 -((20*h) / cycles)))) >0.01) {
      ssP = 0.01 + (0.1 - 0.01)*(1-(1/ 1+ Math.exp(10 - ((20*h) / cycles))));
}
```

This showed clear better results with an RMSE of 0.2768.
The first test with annealing showed pointless because (after observing changing ssP values) it was clear that a bad RMSE result was from the ssP value reaching incredibly low and high points. Therefore, after implementing the if statement above, the results improved massively, being one of the best yet.

```
if (0.1 + (1.15 - 0.1)*(1-(1/ 1+ Math.exp(10 -((20*h) / cycles)))) <1.15 && 0.1 + (1.15 - 0.1)*(1-(1/ 1+ Math.exp(10 -((20*h) / cycles)))) >0.01) {
      ssP = 0.1 + (1.15 - 0.1)*(1-(1/ 1+ Math.exp(10 - ((20*h) / cycles))));
}
```

With 0.1 to 1.15 and limit

```
Cycles,100000
Hidden Nodes,2
RMSE,0.2770878154524702
ssp,0.09987041970570891
```

With more cycles

```
Cycles,200000
Hidden Nodes,2
RMSE,0.27685445614779053
ssp,0.09987041970570891
```

Increasing cycles slightly improved the RMSE on this test.

Increased cycles further

```
Cycles,400000
Hidden Nodes,2
RMSE,0.22219140934988663
ssp,0.09987041970570891
```

Increasing cycles improved the RMSE on this test.

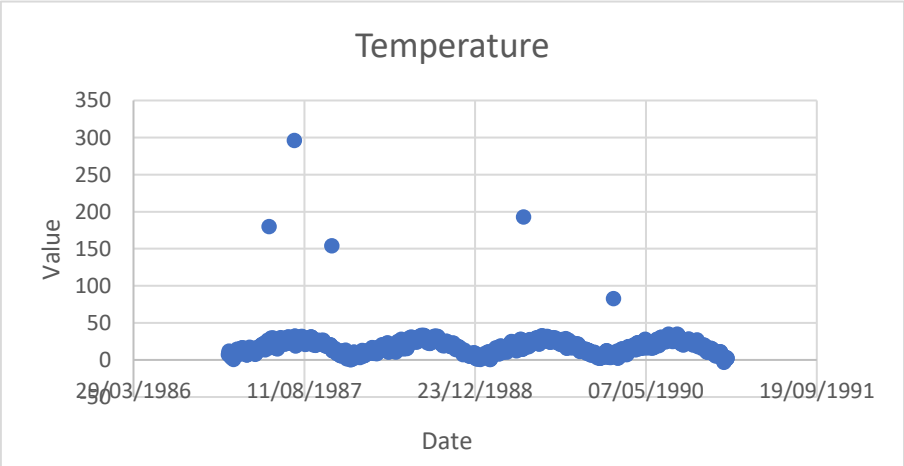
increased cycles to 60000

```
Cycles,600000
Hidden Nodes,2
RMSE,0.277751738918449
ssp,0.09987041970570891
```

The data was over-trained in this test.

Data pre-processing (cleansing and data splitting)

Firstly to get an idea of how to process the data, I created graphs so that I could visually see what the results were like (if spread of only a handful of outliers – if any).



Temperature, wind, solar radiation and DSP (air pressure) had clear outliers that should be removed whereas DRH had more of a spread rather than anomalies. To process the data I wrote a program in Java (StandardisingData) which overall takes the whole data set (from a csv

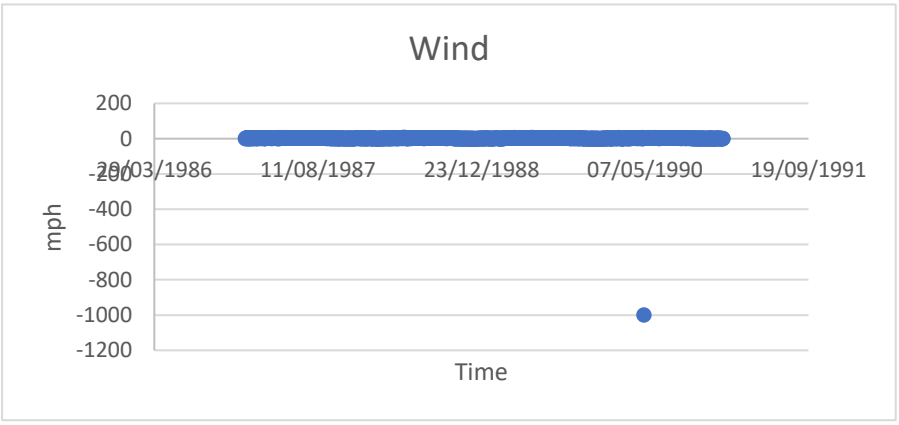
file) and creates 3 sets of cleansed and standardised data.

For temperature, the clear outliers are above 50 celcius, so my program will remove these rows because I think this is a clear way to keep data accurate. Shown by this if statement in my code.

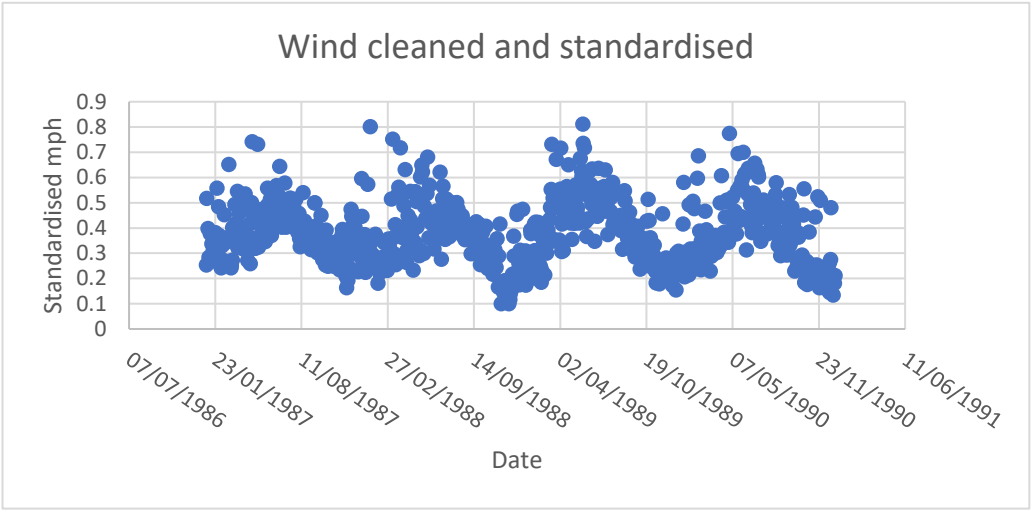
```
inputDataRaw.get(count).temperature <50
```

For wind, there is only 1 clear outlier so I could pick this from the data outright. The results picked out exactly 1 row of data which is what I desired.

The cleaned and standardised wind graph follows much clearer trends like the other results.

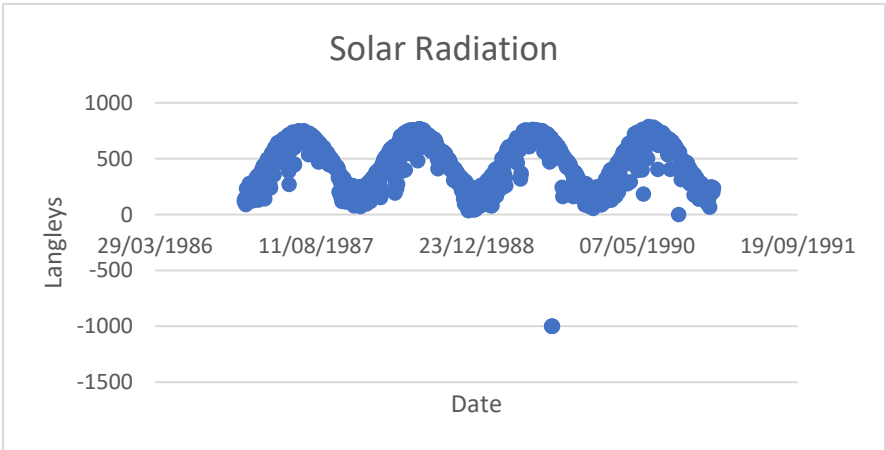



```
&& inputDataRaw.get(count).wind > -100
```



Solar radiation has a fewer clear outliers because the values are valid but don't follow the trend.

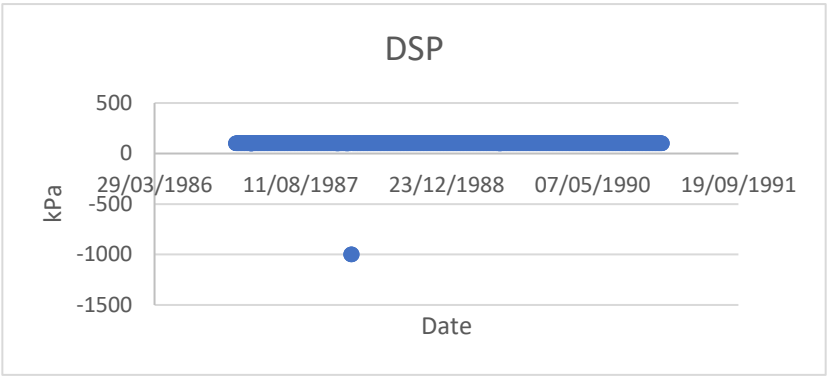
The first outlier is at -999 Langley's which is clearly an anomaly and would affect training. Secondly, the value 0 Langley's is an anomaly so I will also remove this value.




```
(inputDataRaw.get(count).solarRad >0)
```

DSP similarly had 1 clear outlier at -1000 when others were at 0 so I removed this.

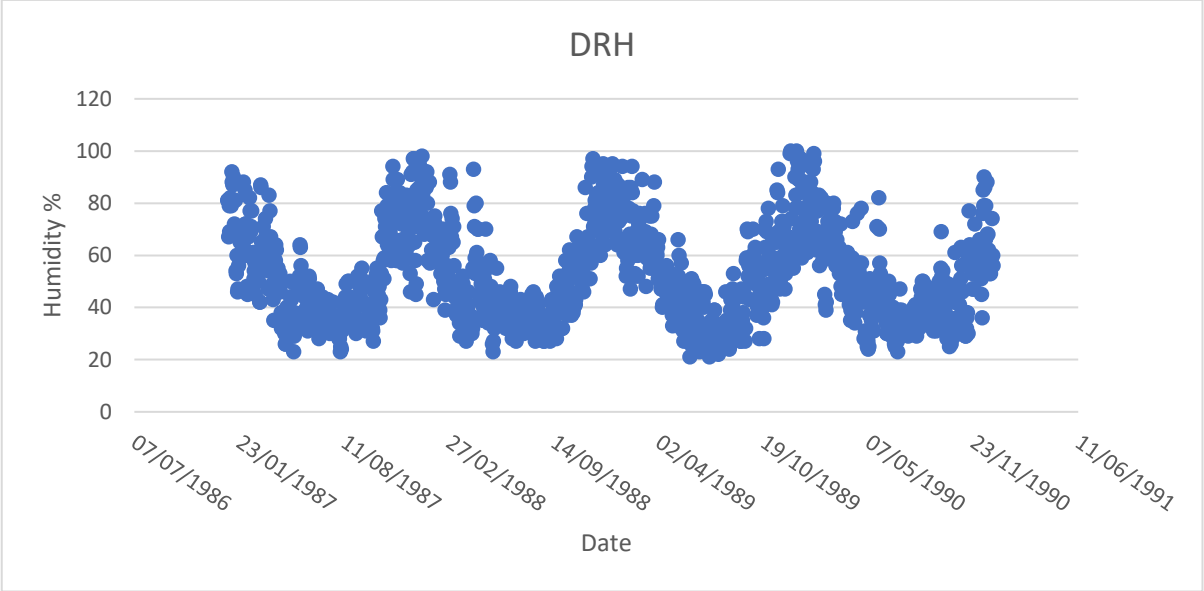
```
inputDataRaw.get(count).airPress >0
```



DRH was a little more complicated since the values of humidity were very spread despite following a trend. Firstly, I calculate the mean, maximum and minimum of DRH from the raw data. I then created my own formula to take out any anomalous data that strays too far from the mean [code reference 1]. I used the key Boolean choice:

If (inputdata < average + (average*0.9)) && (inputData > (average – (average*0.9)))

Where input data is the current piece of data (cell from database); I decided on this method because it after looking at the spread of the data on graphs, there is often a clear trend in the data sets with just a few anomalies. Therefore, my method will pick out these anomalies so it doesn't skew the results or training of the algorithm.



Interpolation

There were a few **empty** values in the data, so I interpolated these by calculating the **mean** of the 4 values adjacent to the data (in that field). For example, D429 was blank so I did (D427-D431) / 4 which resulted in 561.3 and seemed accurate for the data set and trends shown. I also did this for the random cells with a **letter 'a'** and other blank cells, each time checking the result seemed reasonable.

Standardisation

It is important to standardise this dataset because the values are very different to each other, for example the average of wind was 2.23mph whereas solar radiation was 454.31 Langley's. With this data, the ANN would require very small weighting factors to be applied which could cause inaccuracies, insignificantly small changes to weights made by the backpropagation algorithm and some inputs dominating the network more than others.

I standardised the data set with respect to the range of the values to a **[0.1, 0.9] range** – using formula:

$$S_i = \frac{R_i - Min}{Max - Min}$$

S_i : Standardised value
R_i : Raw value

As implemented in code below, the function standardise takes parameters attribute (from inputs), the **minimum** value and the **maximum** value calculated for the attribute.

I **tested** the standardisation with a [0,1] range at first but because I had cleansed the data I thought it would be a good idea to narrow this range and allow for anomalous values that may be fed to the algorithm.

```
public static double standardise(double attribute, double min, double max ) {
            return round3(0.8 * ((attribute - min ) / (max-min)) +0.1);
}
```

Splitting the Data

I split the data to 3 sets **training (60% of data), validation (20%)** and **test (20%)** because I will then use training to adjust the weights to get reliable outputs, validation to prove the validity of this data and test for data that the network hasn't seen yet.

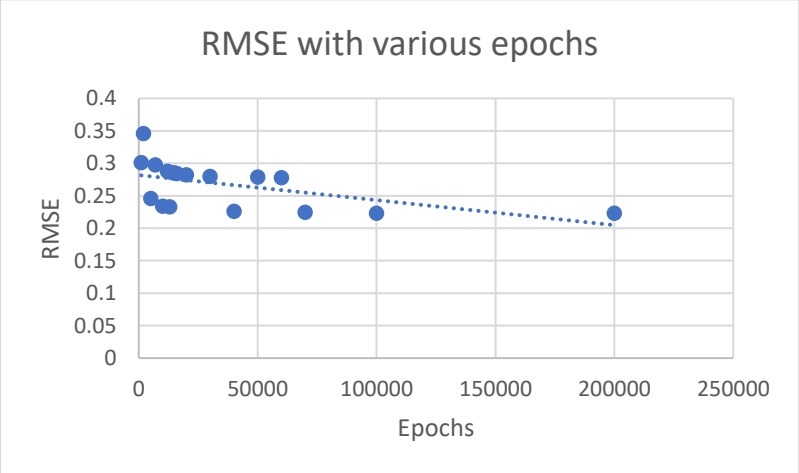
To make these sets **representative** I split the standardised data using if statements where for an index (iterating over the standardised data Array List) is equal to a multiple of 4 then it belongs to test data, if it is equal to a multiple of 3 then add it to the validation data array and anything else will be added to the training data. The table below shows the number of data sets output, it is split as planned.

Data rows	Set
361	Test
360	Validation
720	Training

Note: I also kept note of the regular evaporation value as an extra column so it can be used in the Perceptron class later.

Evaluation of final model

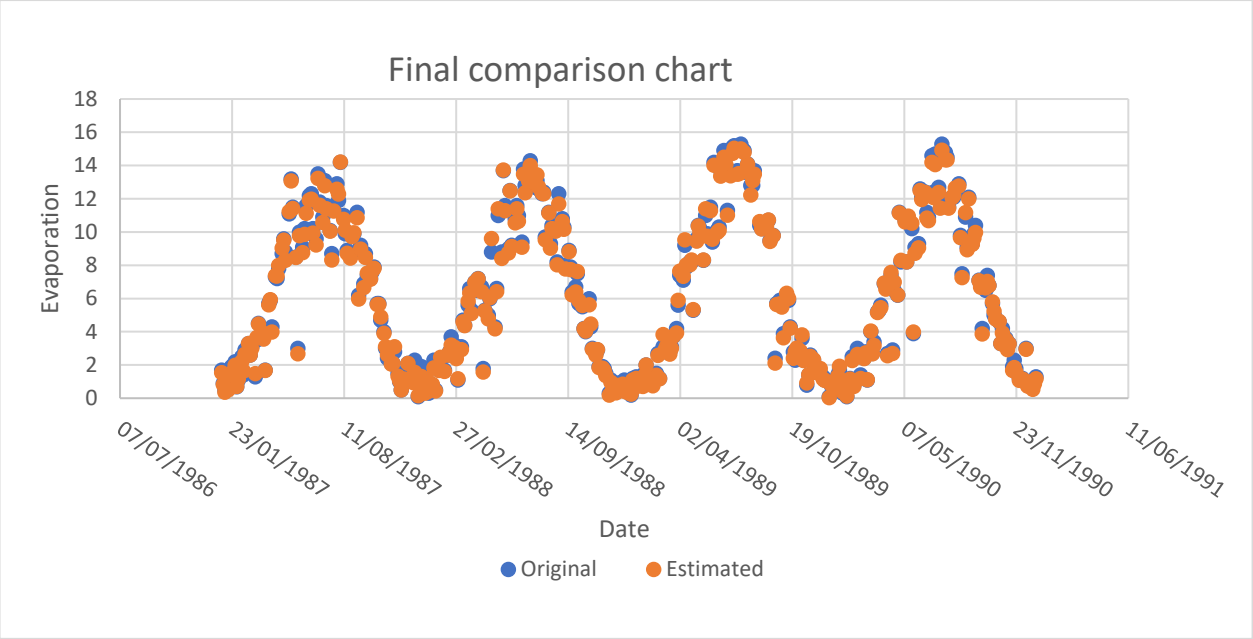
Overall, I performed a lot of testing with various cycles, ssP, hidden nodes and modifications such as momentum, bold driver and annealing. This table shows various RMSE values as the number of epochs vary. The line of best fit shows a small trend of decreasing RMSE as the number of epochs increase.



Past a very low number of cycles the RMSE value gradually got smaller with many local minima's; therefore, I tested a range of cycle numbers to know the best. In the end, 200000 cycles is still the optimum number of cycles, as the while loop roughly estimated above.

I used momentum on updating weights as this improved the RMSE, especially with a smaller step size used. I tested bold driver with different modifications and this method didn't improve my results compared to other code techniques so I didn't use this in my final model

Annealing improved my results from 0.27… to 0.22… at a certain number of cycles so I implemented this modification in my final program.
Throughout multiple tests, using just 2 hidden nodes worked best to get close to the desired output.



The final comparison chart below shows visually the accuracy of my final model as the trend is followed throughout each year. The **MRSE** (mean squared relative error) of this is 0.010381 which is close to 0 which would be the perfect model, therefore showing a successful ANN; this is relative to the observed value. The **CE** (coefficient value) is 0.997503 which is close to 1 which would represent a perfect model. It could also be possible that **date** is used as a parameter because month of the year could follow a trend which the ANN could learn from.

Comparison with another data driven model

Excels LINEST data driven model

I used the formula `=LINEST (G2:G720, B2:F720)` with data looking like this sample:

Wind	SR	Air P	Humid	Standard	Actual Ev	y = b1x	Destan	Colum							
0.25366	0.2084	0.63333	0.73158	0.135	0.8	0.20816	2.14974	1.8218							
0.5177	0.17793	0.51026	0.58421	0.19	1.9	0.08628	-0.27263	4.7203	0.1	min					
0.39765	0.15924	0.55128	0.74211	0.135	0.8	0.11416	0.28144	0.26891	16.1	max					
0.28422	0.31117	0.7359	0.71053	0.145	1	0.06344	-0.72664	2.98127		humid	air	SR	wind	temp	intercept
0.37801	0.22698	0.71538	0.79474	0.125	0.6	0.03768	-1.23858	3.38038	coefficien	-0.14682	0.02322	0.38946	0.28236	0.55403	-0.16187
0.38515	0.1808	0.69487	0.82632	0.12	0.5	0.078	-0.43725	0.87844	standard	0.01782	0.02377	0.01724	0.01766	0.0224	0.02524

On the right you can see the values I used that were the calculated gradients of each input parameter and the y intercept is the 5th value. I then used the formula:

```
=$Q$6*B3 +$P$6*C3 + $O$6*D3 + $N$6*E3 + $M$6*F3 +$R$6
```

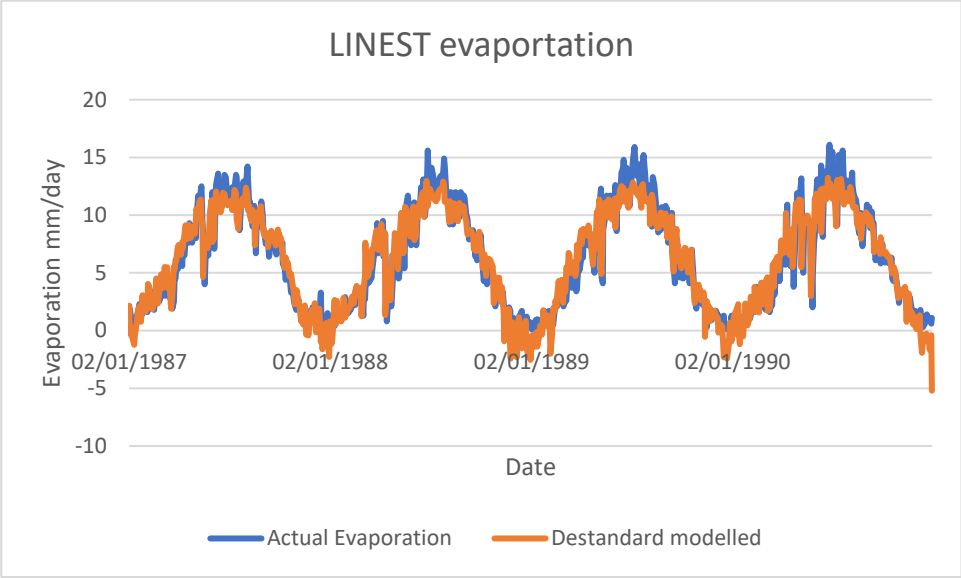
This multiplies each input value by the gradient, similar to my program and adds the y intercept on the end for the predicted output value. I then also standardised the data the same way that I did in my program using formula

```
=((I2 - 0.1) /0.8 * ($K$4 - $K$3) + $K$3)
```

Which takes the ((output – 0.1) / 0.8 * (Max – Min) + Min). This graph now demonstrates how the modelled data performs. The values are following trends opposite of what is meant to be predicted for evaporation which shows the model isn't accurate. Similarly, the model isn't effective because

the more extreme values are never reached when they should be. For example, the modelled data is never predicted to go above 10.6 mm/day whereas there are many values above this and up to 16.1 mm/day – again showing the model isn't accurate.

The RMSE value I calculated for this data set is 1.66393 which is considerably higher than 0.22 that I got, also showing that the model not being tailored to the data set is ineffective. The graph shows the modelled values are relatively similar to the actual values however at the higher extremes the values aren't reached. Overall this data driven model predicts the value to be slightly lower than what it is as a general trend so isn't as accurate as mine. This is because my model has been more tailored to the data set.



<u>Challenges</u>

A problem I encountered in this process was the RMSE value changing each time I ran the program because of the random values I got for weights. I worked around this because it tended to follow trends in particular values of data, so if I saw a clear anomaly for example this was one of the graphs I created when testing so I ran this code again when the anomaly was realised and a much more reasonable answer was produced.