Loughborough University

AI Methods

(21COB107)

AI Methods ANN Implementation report

Alan Wu F027425

Contents

Data pre-processing

In this section I will be detailing how I intend to remove any outliers in the data. As the data set includes data from all days of the year from 1993 up to 1996, my first step is to split the data accordingly so that outliers can be found based off seasons. I decided to go with the following season dates:

- Spring: 01/03/199X -> 31/05/199X
- Summer: 01/06/199X -> 31/08/199X
- Autumn: 01/09/199X -> 30/11/199X
- Winter: 01/12/199X -> end of February 199X

33	01/10/1993	10.8	6.475	12.783	26.3	12.8	8.8	37.6	17.6
34	02/10/1993	35.6	18.363	55.999	82.21	69.6	38.4	30.4	25.6
35	03/10/1993	44.4	16.297	31.95	104.4	0	0	0	0
36	04/10/1993	23	14.144	25.032	68.06	6.4	0.8	16	4
37	05/10/1993	25.7	16.141	34.012	69.72	46.4	52.8	28.8	20.8
38	06/10/1993	71.4	25.919	56.15	133.4	12.8	18.8	12.8	12
39	07/10/1993	95.4	35.334	82.066	185	23.6	28.4	20.8	31.2
40	08/10/1993	68.1	19.787	43.905	150.4	1.6	0	0.8	0.8
41	09/10/1993	37.7	13.397	30.387	90.31	0	13.6	0	1.6
42	10/10/1993	30.2	11.544	24.317	71.14	0.8	0	0.8	2.4

Figure 2 - highlighted data which I deemed to be legitimate

15	13/03/1993	8.95	2.746	7.741	20.56	0.8	0	23.2	6.4
16	14/03/1993	9.01	2.643	8.101	17.97	0	0	61.6	0
17	15/03/1993	-999	2.546	7.179	18.06	11.2	0	86	19.2
18	16/03/1993	-999	2.494	7.232	17.16	0.8	0	12.8	8
19	17/03/1993	9.18	2.559	9.854	20.06	0	0	0.8	4.8

Figure 1 - erroneous data within the data set

After splitting the data set into seasons, I used interquartile ranges to spot any outliers. I used lower and upper bounds via interquartile ranges to highlight any outliers which were in the data set, after doing this I went back through the highlighted cells and made a personal judgement to see if the cell data fitted patterns in context to the cells adjacent to it. For example, Figure 2 is an example of highlighted data which, in my opinion, fits the patterns within the data set – and will therefore be kept to train the perceptron, and Figure 1 is an example where I deemed the data to be an outlier, and will therefore be removed from the dataset.

After removing the outliers in the data set, I went in and split the data set into three sets; the training set (consisting of roughly 60% of the data), the validation set (roughly 25% of the data set) and the Test set (roughly 15% of the data set). To go about distributing the data set into three different sets, I decided the best way to do it would have been to go through the data set 5 rows at a time and assign 3 rows of data to the training set, 1 row to the training set and the final row to the test set. Figure 3 shows how I intend to do this; in order to achieve this, I created two additional columns, validator and test, and used modular arithmetic to flag each 4th and 5th row. I am aware however some rows like 20 exist, but these common multiples will end up in the validator set instead of the test set. My aim in doing this was to ensure that data for different months could be split evenly, as different months in different seasons could have varied weather; for example, the middle of summer would be hotter than the end of summer.

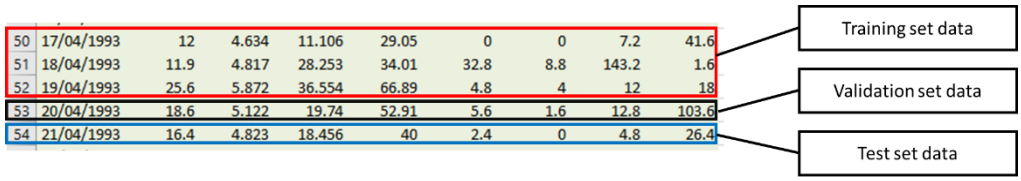


Figure 3 - A visualisation of my approach to distributing the data set into three different sets

Summary of removed data

I decided that I will be predicting Skelton's river flow using the data from the other rivers, and therefore will not be using the rainfall data. As previously mentioned, I had split the data set up into seasons, and I had used interquartile ranges to identify outliers. In most cases, I removed rows of data where there were more than 3 outliers in the row – if they did not fit patterns such as those seen in Figure 2, or if the row contained obviously erroneous data, such as those observed in Figure 1. I also removed the rows which held information about rainfall as I was not planning to use them as inputs for my implementation of an artificial neural network. A sample of this can be seen in Figure 4.

	A	B	C	D	E
1	date	Crakehill	Skip Bridg	Westwick	Skelton
2	01/01/1993	10.4	4.393	9.291	26.1
3	02/01/1993	9.95	4.239	8.622	24.86
4	03/01/1993	9.46	4.124	8.057	23.6
5	04/01/1993	9.41	4.363	7.925	23.47
6	05/01/1993	26.3	11.962	58.704	60.7
7	06/01/1993	32.1	10.237	34.416	98.01
8	07/01/1993	19.3	7.254	22.263	56.99
9	08/01/1993	22	7.266	29.587	56.66
10	09/01/1993	35.5	8.153	60.253	78.1
11	10/01/1993	51	13.276	93.951	125.7
12	11/01/1993	65.5	25.561	69.503	195.9

Figure 4 - A small snippet of the excel spreadsheet after it had the rainfall data and erroneous river flow data removed

Results

Once the outliers were removed, I was still left with 4 sheets, each containing data from individual seasons. I combined these sheets together and ordered them by date so they would be displayed in chronological order, after this, I then added 3 additional columns: *"ref"; "validation"; "test"*. These additional columns were used to help divide up the data set into a training set, a validation set and a test set – this was done using modular division on the "ref" column's values. I then applied a filter to these new columns to divide out the data, an example of this can be seen in Figure 5.

	A	B	C	D	E	F	G	H
1	date	Crakehill	Skip Bridg	Westwick	Skelton	ref	valid	test
6	05/01/1993	26.3	11.962	58.704	60.7	5	1	0
11	10/01/1993	51	13.276	93.951	125.7	10	2	0
16	16/01/1993	99	38.484	77.439	234.4	15	3	0
26	27/01/1993	33.2	16.112	39.664	92.52	25	1	0
31	01/02/1993	20.2	10.624	21.884	56.13	30	2	0
36	06/02/1993	15.5	6.641	14.918	37.74	35	3	0
46	17/02/1993	10.4	4.676	8.969	25.14	45	1	0
51	22/02/1993	8.52	3.777	7.723	20.43	50	2	0
56	27/02/1993	9.45	3.131	8.87	21.74	55	3	0
66	09/03/1993	10.5	2.906	9.187	24.2	65	1	0
71	14/03/1993	9.01	2.643	8.101	17.97	70	2	0
76	21/03/1993	7.75	2.48	8.516	18.08	75	3	0
86	31/03/1993	7.31	2.462	8.001	15.32	85	1	0
91	05/04/1993	17.9	6.957	45.864	44.18	90	2	0
96	12/04/1993	22.6	13.597	25.381	62.7	95	3	0
106	22/04/1993	14.3	4.348	14.608	37.47	105	1	0
111	27/04/1993	24.4	11.684	22.181	73.96	110	2	0

Figure 5 - Using modular division to filter the data set to show only the test data. This was achieved by filtering the test column to show only "0" and the validation row to hide value "0".

After doing this, I copied and pasted the filtered data into separated sheets on the same Excel workbook, my plan is to read all the data from the same workbook via different sheet numbers using the Apache POI API. I will elaborate on this further on in the report.

After all the data pre-processing, I am left with 9 sheets in total, where the leftmost 3 will be read by my neural network, this can be seen in Figure 6.

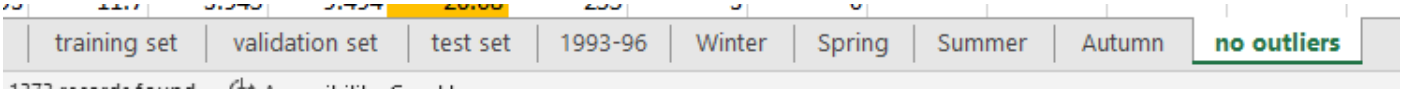


Figure 6 - All the sheets in my excel workbook

Normalisation of the data set

In my algorithm, I intended to use both the date and river flow of Skelton as inputs, but some further processing had to be done to allow for this as my algorithm would not understand Excel's short date format.

In order to use the date as an input, I had to convert all the dates "Short Date" number format to number formats. By doing so, all dates were assigned a number, which represents how many days that given date is from 01/01/1900; so, for example, 01/01/1993 becomes 33970 (as 01/01/1993 is 33970 days after 01/01/1900).

After this had been done, I read the excel sheets into my Java program using Apache POI, where I converted the results using my normalisation algorithms, and printed the results so that it could be stored in another part of the system. I had two separate algorithms; one for normalising the river flow so that the result would sit between 0 and 1, and the other for normalising the date so that the result would also sit within the boundaries of 0 and 1.

Normalising river flow

Figure 7 shows the code I wrote to normalise and de-normalise the various river flow data I will be using. As each river flow has different maximum flows, I gave them all different divisors, of which are defined as constants, see Figure 8, at the top of the excel_handler class. The maximums are 20% more than the given river's maximum flow in the excel spreadsheet, after outliers have been removed.

```
static int MAX_SKELTON = 320;    // n
static int MAX_CH = 132;     // maxim
static int MAX_SB = 54;      // maxim
static int MAX_WW = 162;     // maxim
```

Figure 8 - defined constants which are used for river flow standardisation

```
private static double normaliseFlow(double flow, int river) {
    double returnValue = 0;
    if(river==0) {      // crakehill
        returnValue = ((flow/MAX_CH)*0.8)+0.1;
    } if (river==1) {   // skip bridge
        returnValue = ((flow/MAX_SB)*0.8)+0.1;
    } if (river==2) {   // westwick
        returnValue = ((flow/MAX_WW)*0.8)+0.1;
    } if (river==3) {
        returnValue = ((flow/MAX_SKELTON)*0.8) + 0.1;
    }
    return returnValue;
}

private static double denormaliseFlow(double flow, int river) {
    double returnValue = 0;
    if(river==0) {      // crakehill
        returnValue = ((flow - 0.1) / 0.8) * MAX_CH;
    } if (river==1) {   // skip bridge
        returnValue = ((flow - 0.1) / 0.8) * MAX_SB;
    } if (river==2) {   // westwick
        returnValue = ((flow - 0.1) / 0.8) * MAX_WW;
    } if (river==3) {
        returnValue = ((flow - 0.1) / 0.8) * MAX_SKELTON;
    }
    return returnValue;
}
```

Figure 7 - code snipper for normalisation and denormalisation of river flow data, it uses an index "river" to also decide how to handle the incoming flow parameter

Normalising the date

Figure 9 shows the code I wrote to normalise and de-normalise the date values for data set. As previously mentioned, my implementation takes the date as an index, where the index is the number of days a given date is from 01/01/1900. I use a constant MIN which is defined as the lowest index of the data set, being 01/01/1993 or 33970 as an index. As there are ~1400 rows of data, I decided to keep the date space as 2000, and have the normalised values sit between 0 and 1 through the division of ((key-MIN) / 2000). The parameter key is the date as an index.

```
private static double normaliseKey(double key) {
    double returnValue = ((key - MIN) / 2000) + 0.1;
    return returnValue;
}

private static double deNormaliseKey(double key) {
    double returnValue = ((key - 0.1) * 2000) + MIN;
    return returnValue;
}
```

Figure 9 - code snipper of my date normalisaton code

Algorithm implementation

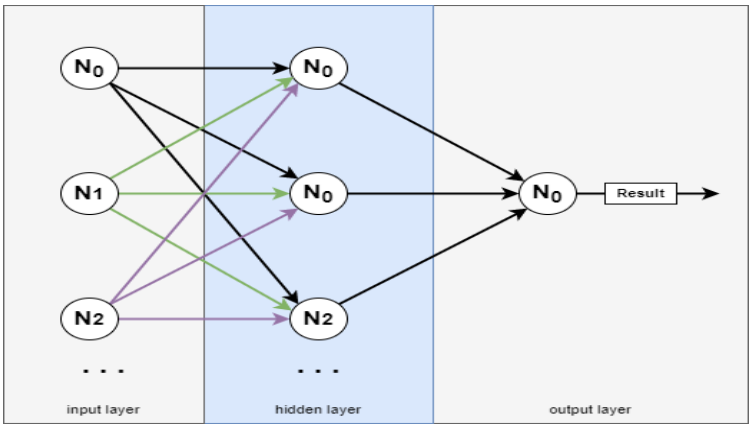


Figure 10 - Diagram representation of how my ANN implementation is structured

In my implementation of my artificial network, I intend to use 4 inputs, 6 hidden neurons and 1 output neuron for each row of data in the data sets. Figure 10 shows how I have structured my implementation. I will be using the date, river flow of CrakeHill, WestWick and Skip Bridge as my inputs, along with a target result, being Skelton's river flow on a given day.

My solution has been made using Java, where it comprises of 9 total classes – 5 of which are used for the neural network itself, and the other 4 being used to handle data to and from the excel sheets. My implementation reads data from the excel sheet, but requires that the user stores the output into a separate class, named **`data_sets`** and **`Training_data`**, before the program can run the neural network on the data set. After training the network on a given data set, the program will store all outputs in an excel file under the root directory of the java project, the name of this excel file is "results". In results.xlsx, there will be a sheet called "results", of which stores the outputs of my program. It should be noted however that the Excel file has dates stored as indexes from 01/01/1900, so in order to view the output data, some processing has to be done: highlight all date cells and change the date representation from text to short date. In order for my implementation to work directly with excel, I used the Apache POI API.

In the following sections, I will be detailing how my program runs, as well as the individual functionality of each class within my implementation. I have attached the source code for my solution at the end of this report.

Driver class

The driver class in my implementation is used as an application driver, the main method is used to run and train the neural network. In this class, the number of epochs for training is defined as a constant. When the application is run, it will construct a network with 6 hidden neurons, after this the user is prompted to choose an option to further proceed: run, run_validation, run_test, train or exit.

Running the program will go through all training data and forward propagate them to get outputs. These outputs are stored in an array of doubles called "results", which is defined at the start of the main method. This results array is then parameterised and passed to a procedure called "writeToExcel" belonging to the excel_handler class, which writes the results into an excel file called "results" under the root directory of the java project, it will take in a number of parameters to know where to store the file and what to put in it. After this is done, the system will prompt the user with a message to let them know that their results are ready to view.

Running validation and running tests will also go through all training data and forward propagate them to get outputs for the results array. A call is then made to the "validate" function in the runner class, of which will validate the outcomes of the training data, or, a call is made to the "test" function in the runner class to use the training outcomes on the test data set These functions also make a call to the "writeToExcel" procedure to store their results. Unfortunately, I did not have enough time to modify the validation set for it to include learning re-enforcements and so as a result, the validation set does not really validate the learning outcomes. I will speak more about this further on.

If the user chooses the train option, the program will go through all pre-defined number of epochs, where in each epoch the program will iterate through all training data, followed by forward propagating and backpropagation of the error. After all epochs have been iterated through, the program will prompt the user to let them know that

training on the training data set has been completed. Additionally, while training the data, the program will keep track of the root mean squared error by storing it in an array of doubles; after training is done, this array is written to an excel file where a graph can be made to observed the performance of the neural networks ability to learn. I will speak more about this in the evaluation section.

Lastly, if the user chooses the “exit” option, the program will terminate.

Neural Network Class

The neural network class hosts the logic for back propagating the error as well as the running of the network. We also define constants for the learning rate, the number of input neurons and the number of output neurons; in this case there are 4 input neurons and 1 output neuron for each row of data in the data set. The number of hidden neurons in the network is defined in the driver class, of which as previously mentioned is 6. The learning rate dictates how quickly and how much the neural network will learn in each epoch, a lower learning rate means less is learnt, but each epoch is faster and a higher learning rate means more is learnt per epoch at the cost of speed. There are advantages and disadvantages to using learning rates too high or too low; if a low learning rate is used, it may not converge fully; if a high learning rate is used, it may converge too quickly – both of which result in suboptimal solutions. Additionally, defined at the start of the neural network class is an array of the different layers which belong to the network itself.

The forward propagation method, called “fProp” runs the network by taking in an input array, of which contains the inputs and the target result for each neuron in the network. This method will iterate through all layers of the network via a for loop to look for the layer in which a given neuron resides in, there are then three different outcomes for each neuron.

- If the input belongs to the forward propagation method is in the input layer, the neuron’s output is set to be the parameterised input value.
- If the input belongs to the hidden layer, the weighted sum is calculated by adding together the weights of each neuron. The activation function (sigmoid function is used here) is then applied to the weighted sum and this is set to be the output for each neuron in the hidden layer.
- If the input belongs to the output layer, the sigmoid function is applied to the output neuron’s weighted sum and the outcome of this is set to the output for the output neuron

In the Neural Network class, we also have the backpropError method, of which takes in a target result as a parameter and aims to backpropagate the error. To begin with, the backpropError method separates out each neuron by layer type by putting them into different arrays (“inputNeuron”, “hiddenNeuron” and a single “outputNeuron”). This method calculates the error on the output neuron, this error is then used to calculate the weights for our output neuron. We then backpropagate this error to the hidden neurons and use it to calculate the error on the hidden neurons. We can then use the error of the hidden neurons to calculate the weights of the hidden neurons.

Neuron class

My neural network implementation comprises of a network comprising of neuron objects belonging to the neuron class; the Neuron class is used to represent a neuron in the network. Each neuron has a set of weights which is represented by a list of the primitive data type, doubles. Additionally, each neuron has an output, error and layer type. The error and output are used during the forward and back propagation processes, and the layer type is used to

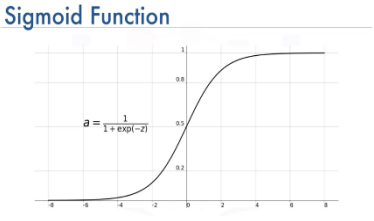


Figure 11 - Sigmoid function

```
public Neuron(e_layerTypes layerType, int numbOfWeights) {
    this.layerType = layerType;
    if (layerType != e_layerTypes.INPUT) {
        weights = new double[numbOfWeights];
        IntStream.range(0, numbOfWeights).forEach(x -> weights[x] = 0.5 - Math.random());
    }
}
```

Figure 12 – constructor for the neuron class

structure the network. The constructor of the Neuron class will initialise the layer type of a neuron (a value from the Enum class “e_layerTypes”), and if the instantiated neuron belongs to the hidden or output layer, weights are randomly initialised for that given neuron, this can be seen in Figure 12. Contained in the Neuron class is the function used to apply the sigmoid activation function, called “applySigmoid”, of which will calculate an output for the neuron. Additionally, there is a function used to calculate the derivative of the neuron’s output, called “calcDerivative”.

Layer Class

The layer class represents a layer in the network (either input, hidden or output). Each layer becomes a layer object of which will comprise of an array of neurons – these neurons belong to a given layer; for example, a hidden neuron will belong to the hidden layer. The layer type is chosen from the Enum class “e_layerTypes”. The layer class contains specified getter and setter methods, as well as a toString method to return information about a layer, of which was useful to use during the debugging process. The constructor takes in the neural network that a given later belongs to and it will initialise the layer type, depending on what layer we are in, we will either populate the neurons array with input, hidden or output neurons.

Enum e_layerTypes

This Enum is used to define the different constants that a layer can have in my implementation of the neural network, comprises of three types: input, hidden and output.

data_sets Class

This class contains the validation and test sets after it has been read in from excel and normalised via the normalisation functions in Figure 7 and Figure 9. This class stores each data set as a 3 dimensional array. More on this will be covered in the excel_handler class.

My reasoning for having a separate class to store the data sets is because reading in an excel file and passing it around my program as a parameter was too processor intensive for my laptop, which resulted in slower processing. Although not an ideal programming style, I decided to make this compromise so that I could benefit from processing speeds. The downside of doing this is that the data_set class would have to be overhauled if there were changes made to the data set, and that the overall implementation would not work flexibly with other data sets – unless they were processed and stored into this class. Another thing which I would like to mention is that there is a byte limit to each java class, so as a result, my implementation would not be able to work on really large data sets.

TrainingData Class

This class contains the Training data set after it has been read in from excel and normalised via the normalisation functions in Figure 7 and Figure 9. This class stores each data set as a 3 dimensional array. More on this will be covered in the excel_handler class.

This had to be stored in a separate class as there was not enough space to put it in the same class as the other data sets.

excel_handler Class

This class handles the reading and writing of the data sets and various result excel files in my implementation.

When reading the data set, my algorithm takes in the individual dates of the data set as a “date” and the various flow data as a under different variable names (“ckFlow”, “sbFlow”, “wwFlow”, “target”). I initially wanted to read the data sets as HashMaps and pass them around as parameters as iterating through HashMaps is usually fast, however I faced two main problems from doing this: firstly, storing the data as a HashMap resulted in data not being processed in chronological order which really messed with the outputs of my neural network; secondly, as previously mentioned, this approach was simply too processor intensive for my laptop, causing performance issues.

After abandoning this idea, I decided instead to use the previously covered “data_sets” (and “TraininigData”) class; in order to populate the different 3 dimensional arrays in this class, the user has to use the commented out getData() functions in the main method of the excel_handler class to print each data set into an acceptable format to the console. After the data set is completely read, the user has to copy and paste the console output to the relevant array in the data_set class. As previously covered, this of course is not ideal and has its own problems. However, each row of data is formatted as:

```
{{DATE, CRAKEHILL FLOW, SKIPBRIDGE FLOW, WEST WICK FLOW} , {TARGET (SKELTON’S) RIVER FLOW}}
```

Contained in the excel_handler class is also the procedures which write to the various excel files and the data normalisation algorithms shown in Figure 7 and Figure 9.

Write errors: This procedure will take in the array of root mean squared errors from the driver class and store the values in an excel file called “errors.xlsx”. In this file, the error is plotted against epochs and a graph can be made to visualise the performance of the neural network.

writeToExcel procedure takes in a few parameters and writes a parameterised array of doubles to the correct excel file. The excel file can then be opened to plot graphs in order to visualise the learnings of the neural network.

ANN evaluation

In this section I aim to evaluate my ANN implementation by discussing my personal experiences of this project during development, as well as discussing the performance of my implementation through the use of root mean squared errors.

In my implementation, I did not manage to implement any improvement strategies, this is because I found that I spent too much time prioritising implementing the Apache POI API so that my program could work directly with the Excel sheets. Despite my efforts, my implementation does not fully work directly with the Excel sheets as the user has to store console outputs into a separate class as previously mentioned in the implantation section. Additionally, my implementation does not validate the outcomes of the training data set as there are no learning re-enforcements in my implementation, this resulted in some hard coded outcomes, and the neural network predicting roughly 50% of the river flow within an acceptable range from the actual value. Despite this, my artificial neural network is still able to match patterns, but is unable to correctly predict the correct flow value, see Figure 21 and Figure 22 for a visualisation.

I want to add that my overall programming in this implementation is far from ideal as there are a number of hard coded elements, If I had more time to develop this, I would work on making my implementation work directly with excel so that it would be easier to alter the data sets, additionally, I worked to get a working solution before making any improvements to the coding style.

Another thing which I would like to add is that my implantation requires the user to further process the output data before it can be viewed. This is due to how my implantation reads in the flow dates as indexes from 01/01/1900. As mentioned earlier, my priorities were to get the code to work first and worry about smaller tasks, such as formatting the date, afterwards. In retrospect however, I feel that implementing this feature earlier would have saved me a lot of time as I started realising how long it takes me to highlight all date cells and change their format (even with CTRL space). Additionally, a function to automatically generate graphs would have been ideal too as most of my debugging time was spent generating and analysing the various output graphs which my implantation makes.

However, not all is bad. My implementation of the training algorithm, in my opinion, works fairly well; it is able to follow the patterns observed in the actual training data set. Below I have attached some side-by-side comparisons of my trained data outcome and the actual training data, the graphs were generated using Excel.

(cont'd…)

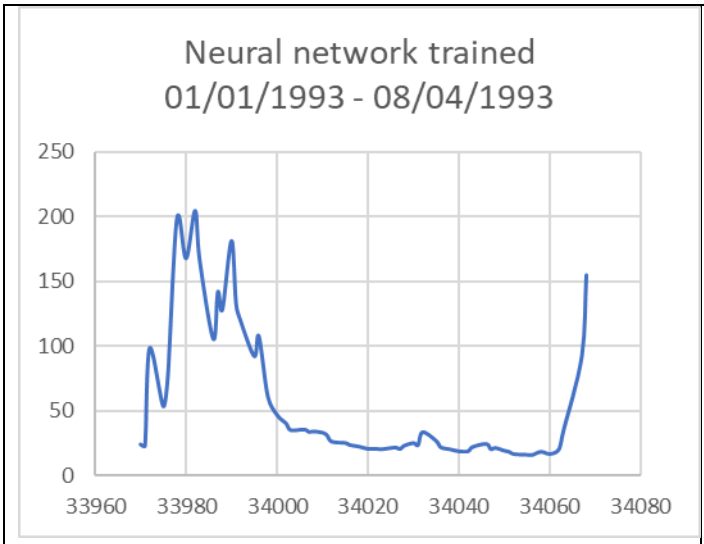


Figure 13- Excel generated graph of the first 59 days of training data outcomes, where river flow is plotted on the y axis and the date, as an index, is plotted along the x axis.

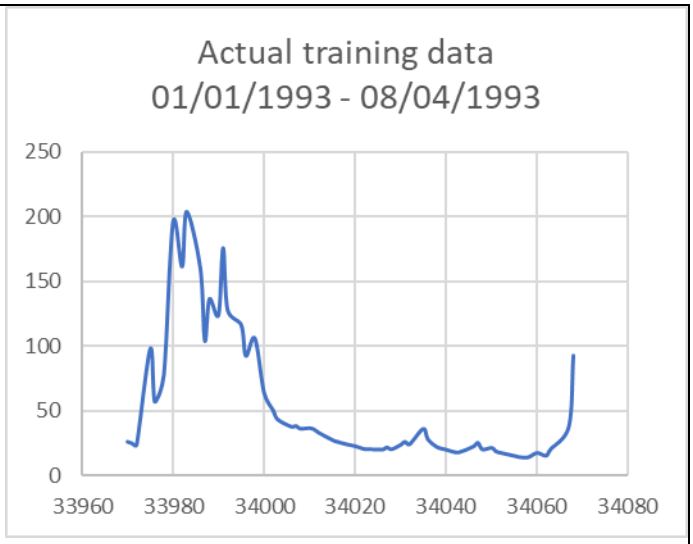


Figure 14 - Excel generated graph of the first 59 days from the training data set, where river flow is plotted on the y axis and the date, as an index, is plotted along the x axis.

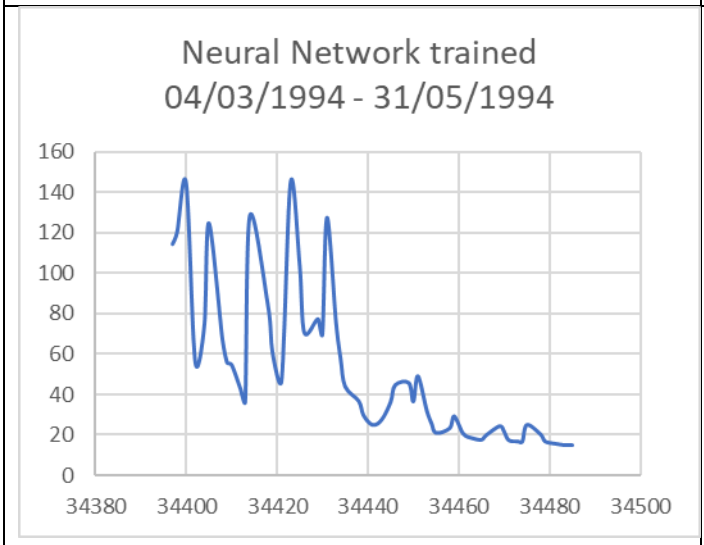


Figure 15 - Excel generated graph of the spring months of 1994 from the training data outcomes, where river flow is plotted on the y axis and the date, as an index, is plotted along the x axis.

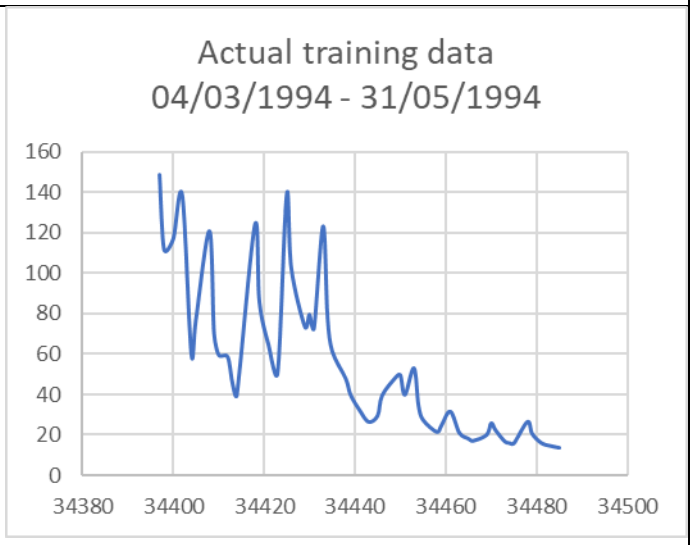


Figure 16 - Excel generated graph of the spring months of 1994 the training data set, where river flow is plotted on the y axis and the date, as an index, is plotted along the x axis.

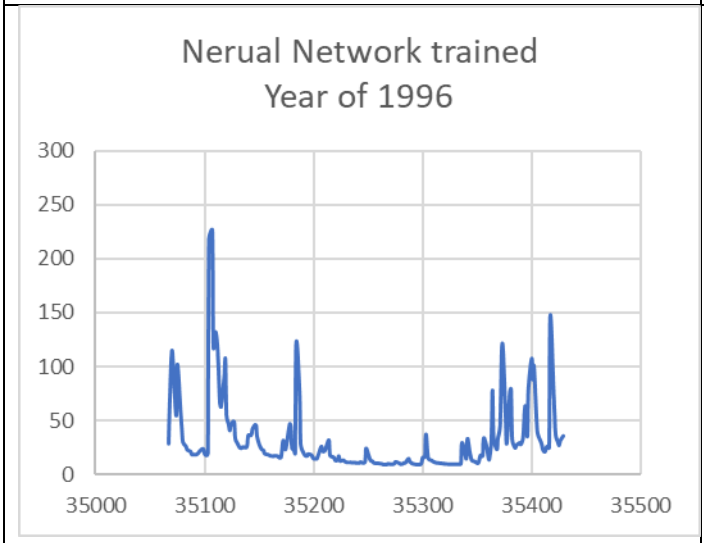


Figure 17 - Excel generated graph of all training outcomes of 1996 from the training data outcomes, where river flow is plotted on the y axis and the date, as an index, is plotted along the x axis.

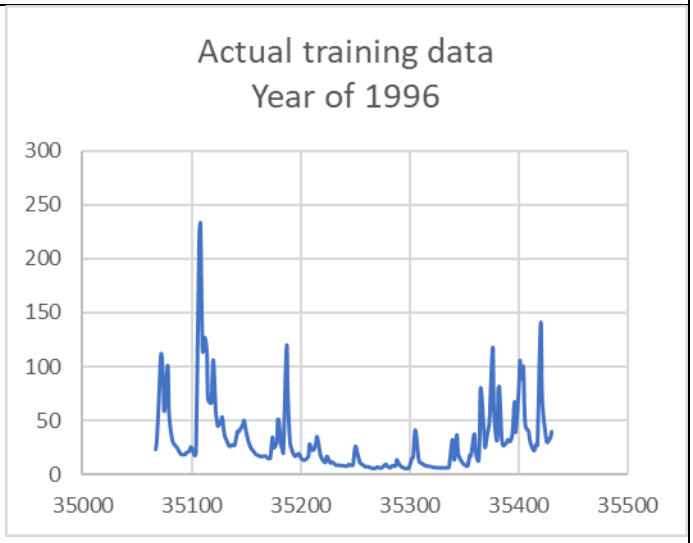


Figure 18 - Excel generated graph of all 1996 training data from the training data set, where river flow is plotted on the y axis and the date, as an index, is plotted along the x axis.

As can be seen from the results on the previous page, the training algorithm trains well enough to follow the patterns of the actual data set.

In order to choose an appropriate training configuration (epochs and learning rate), I used the root mean squared errors to find a suitable fit. I tried a number of configurations and felt that 60 000 epochs of 0.8 learning rate would be suitable. My reasoning for choosing such a high learning rate is because I have 4 input neurons, so for my neural network to learn more each epoch, I decided to go with a higher learning rate. I did also find that 60 000 epochs were also where the 0.8 LR would also roughly have the lowest error, which is why I went with 60 000 epochs, Figure 19 shows my results.

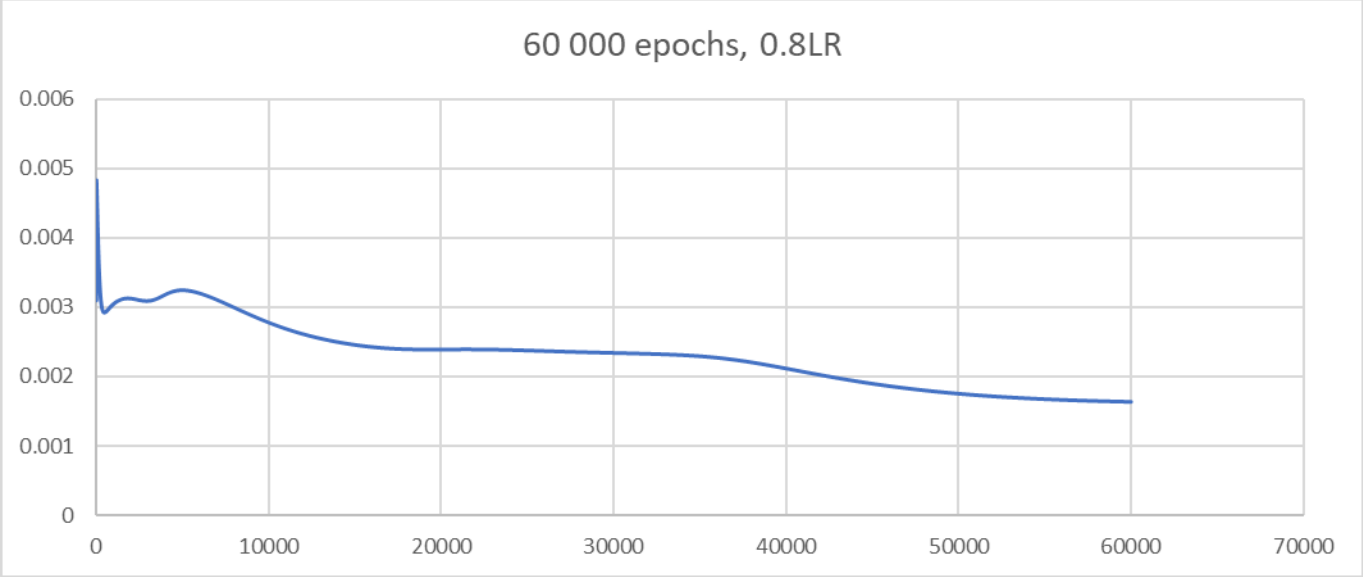
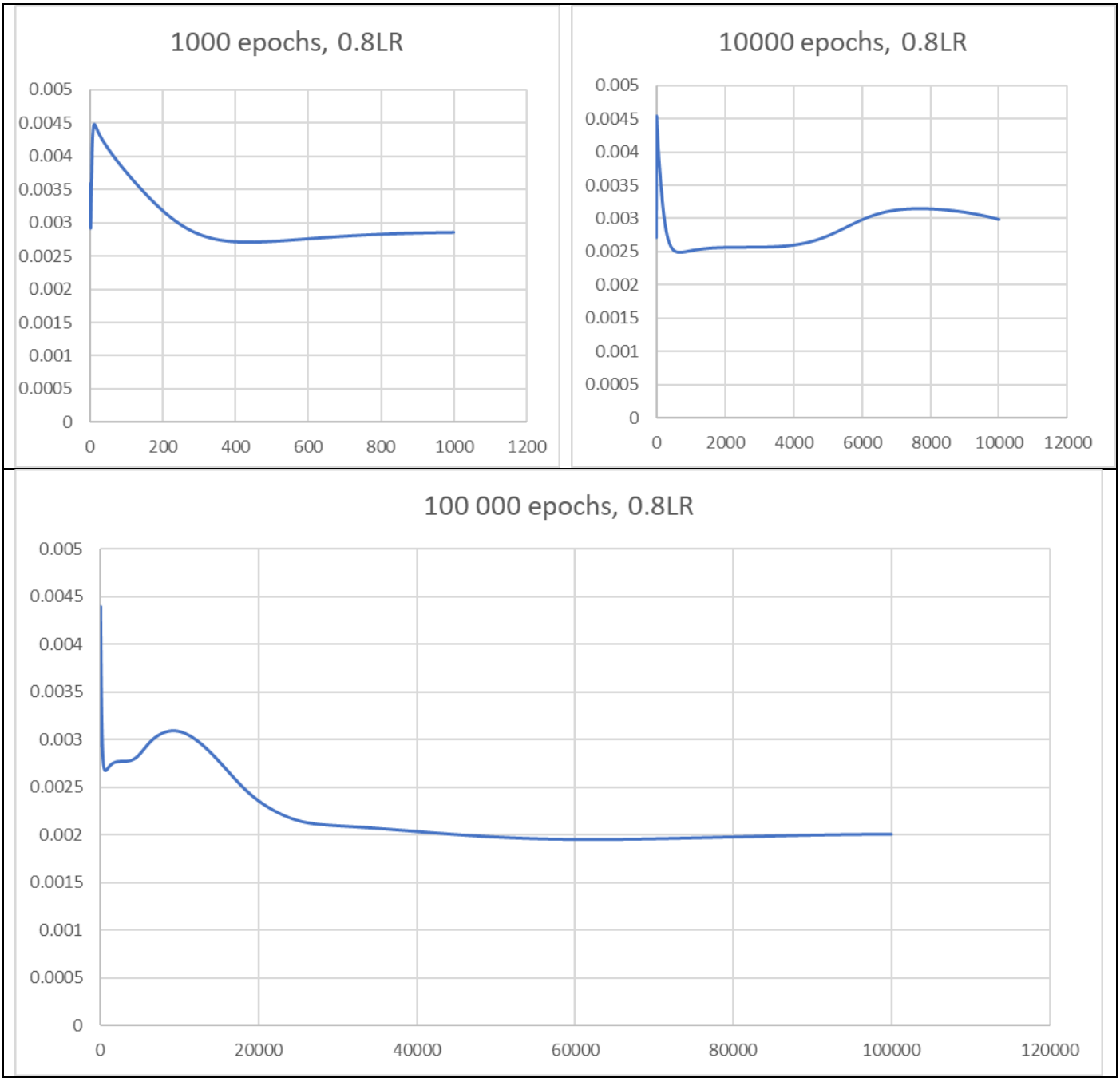
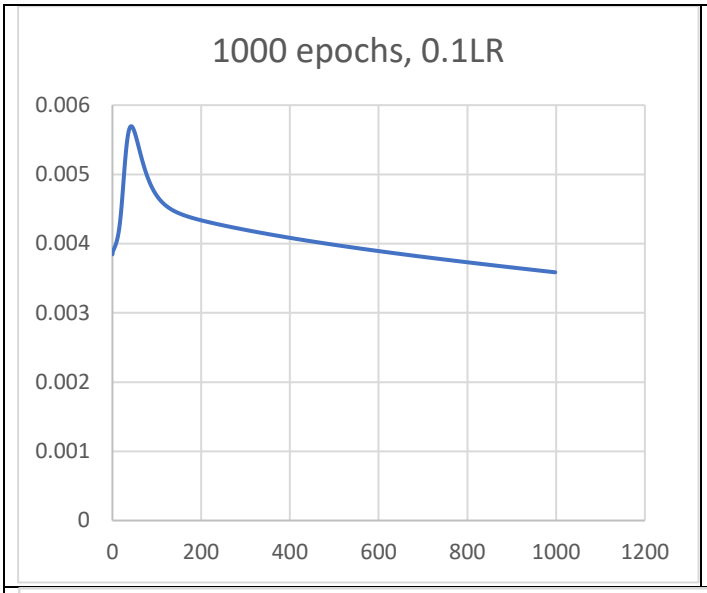
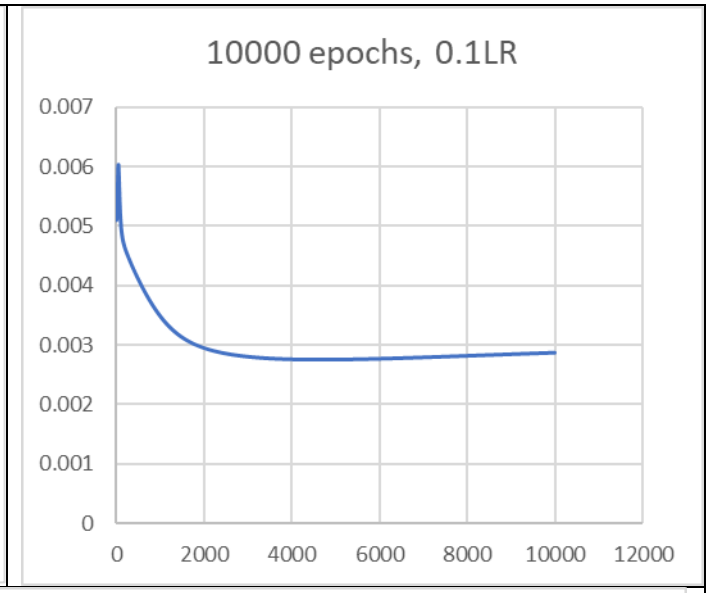


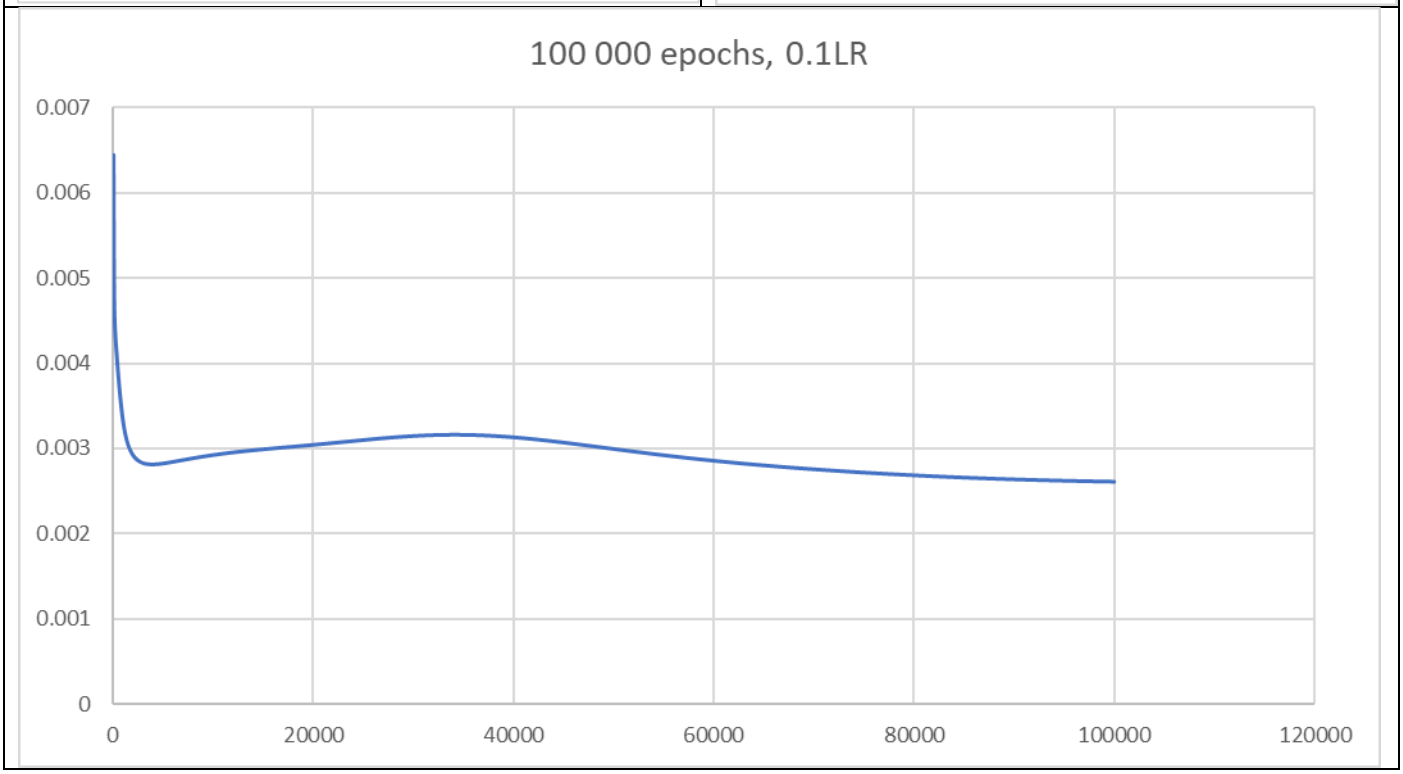
Figure 19 - Use of 60 000 epochs and a lerning rate of 0.8

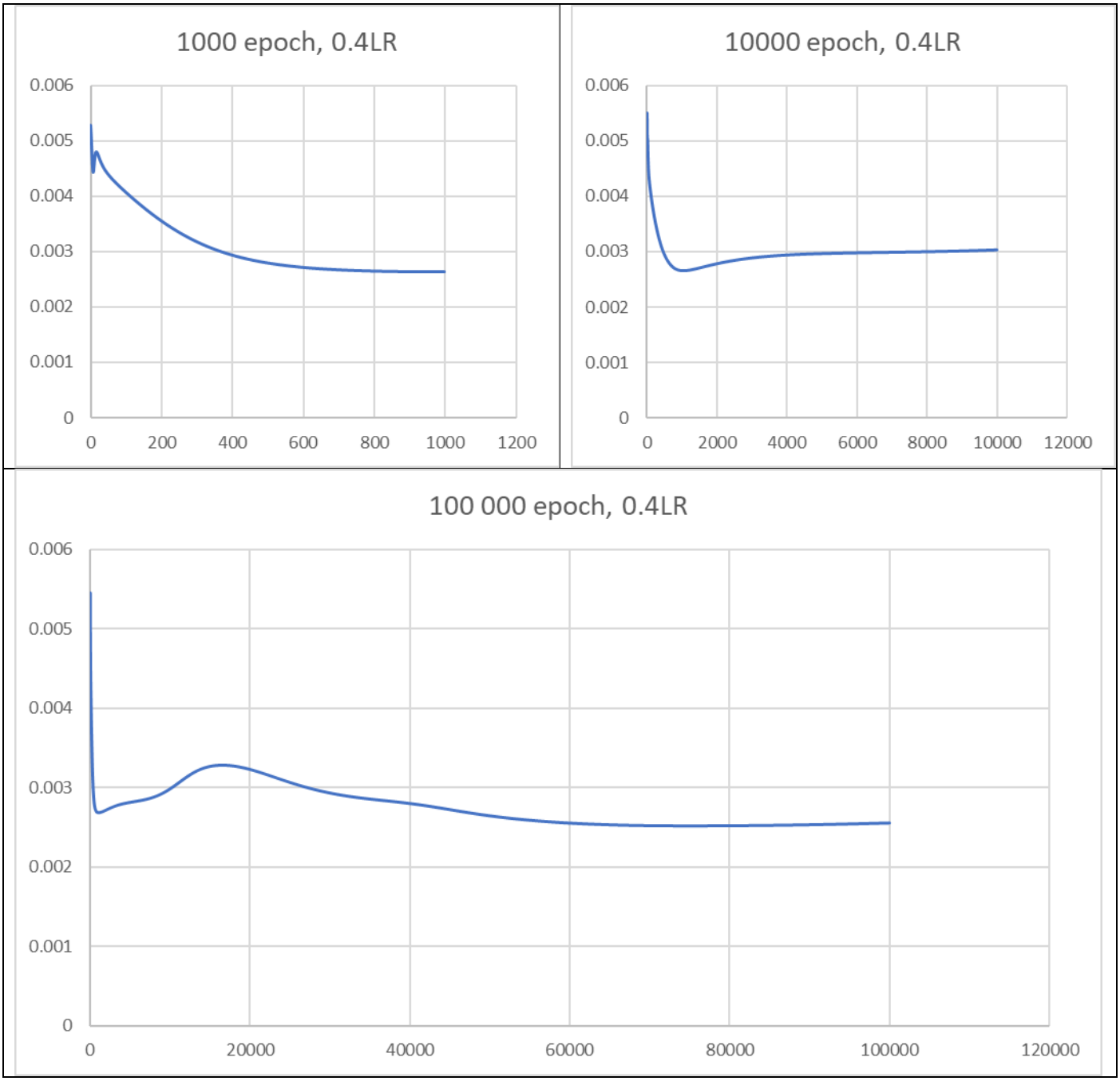
In the following pages, I have attached some graphs which show the RMSE, where the errors are plotted on the y axis and the epochs are plotted on the x axis.


1000 epochs, 0.8LR
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
200
400
600
800
1000
1200
10000 epochs, 0.8LR
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
2000
4000
6000
8000
10000
12000
100 000 epochs, 0.8LR
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
20000
40000
60000
80000
100000
120000


1000 epochs, 0.1LR
0.006
0.005
0.004
0.003
0.002
0.001
0
0
200
400
600
800
1000
1200


10000 epochs, 0.1LR
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
2000
4000
6000
8000
10000
12000


100 000 epochs, 0.1LR
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
20000
40000
60000
80000
100000
120000


1000 epoch, 0.4LR
0.006
0.005
0.004
0.003
0.002
0.001
0
0 200 400 600 800 1000 1200
10000 epoch, 0.4LR
0.006
0.005
0.004
0.003
0.002
0.001
0
0 2000 4000 6000 8000 10000 12000
100 000 epoch, 0.4LR
0.006
0.005
0.004
0.003
0.002
0.001
0
0 20000 40000 60000 80000 100000 120000

Comparison with data driven models

In this section, I will be comparing the outcomes of my training and validation sets to their actual data counterparts. In order to do this, I put the validation and test outcomes together with the actual validation and test set data into the same excel sheets. I then made line graphs with these new sheets, with both my neural network outputs combined with the data from the datasets, Figure 20 shows how I did this for my validation sets. Figure 21 and Figure 22 show my results.

	Date	Actual	Skelton
2	33977	56.66	124.20112
3	33981	125.4	115.54237
4	33986	160.1	149.81695
5	33990	124.3	68.109044
6	33995	116.4	18.364913
7	33999	80.07	29.395934
8	34003	43.38	4
9	34007	38.31	26.002442
10	34011	35.6	19.696611

Figure 20 - Comparison table of my validation set

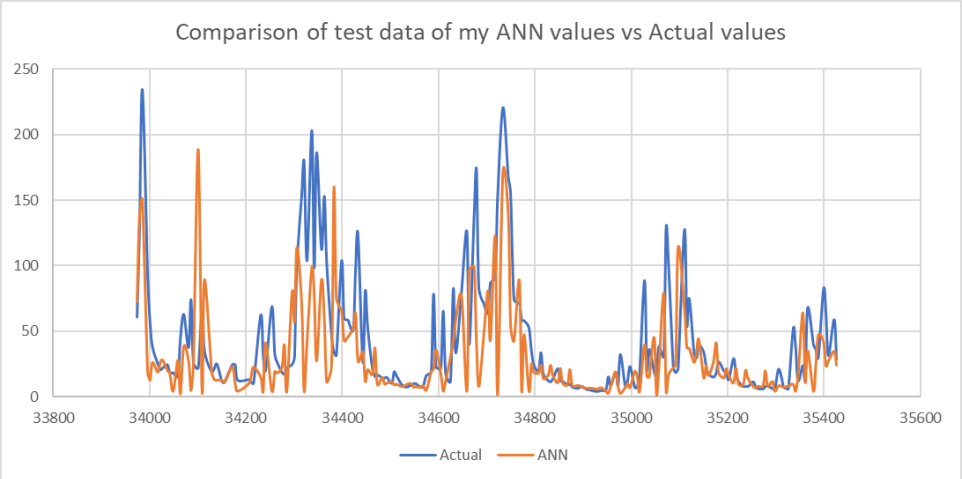


Figure 21 - comparison of test outputs after neural network training and the actual values from the test data set

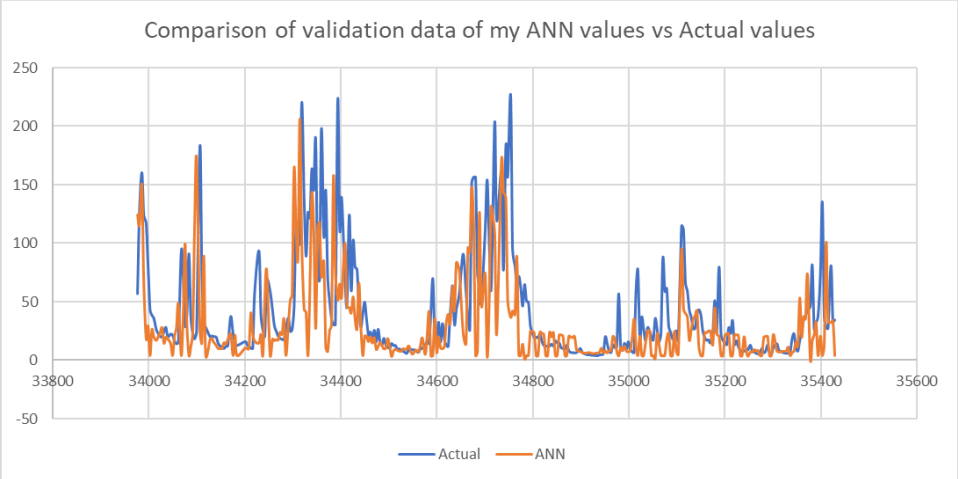


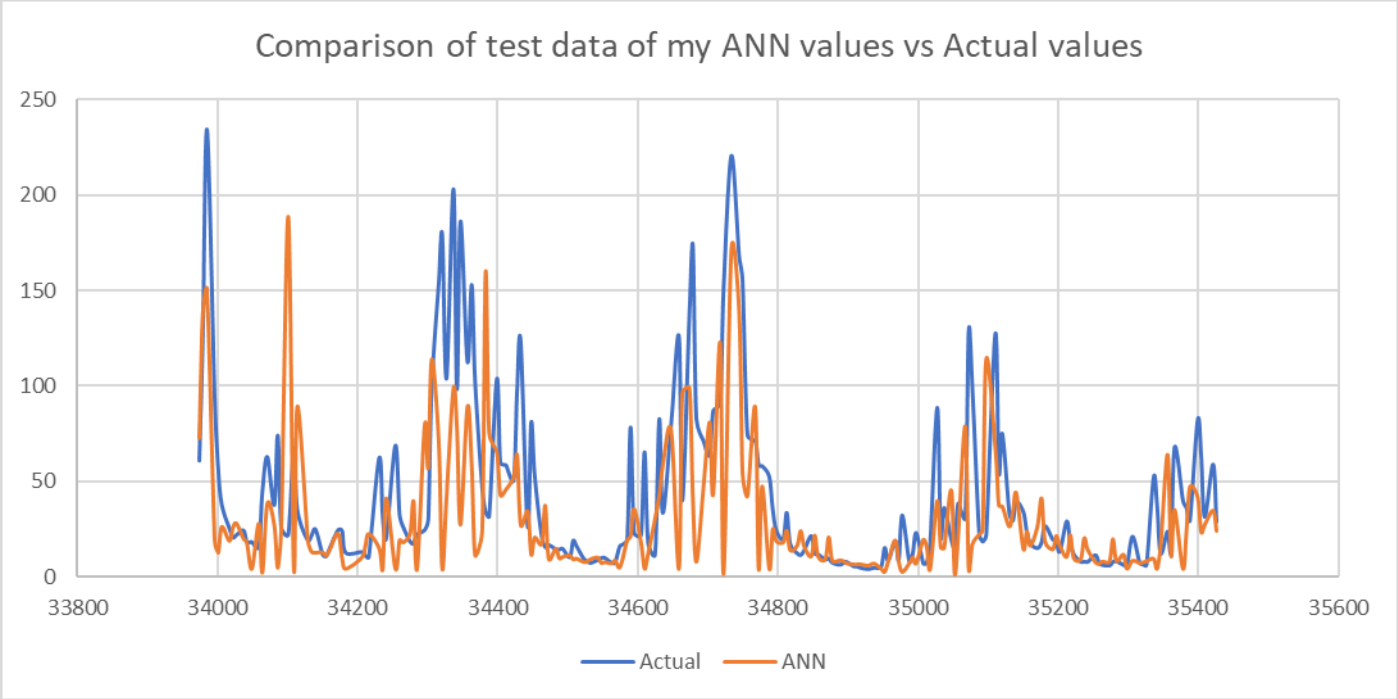
Figure 22 - comparison of validation outputs after neural network training and the actual values from the validation data set

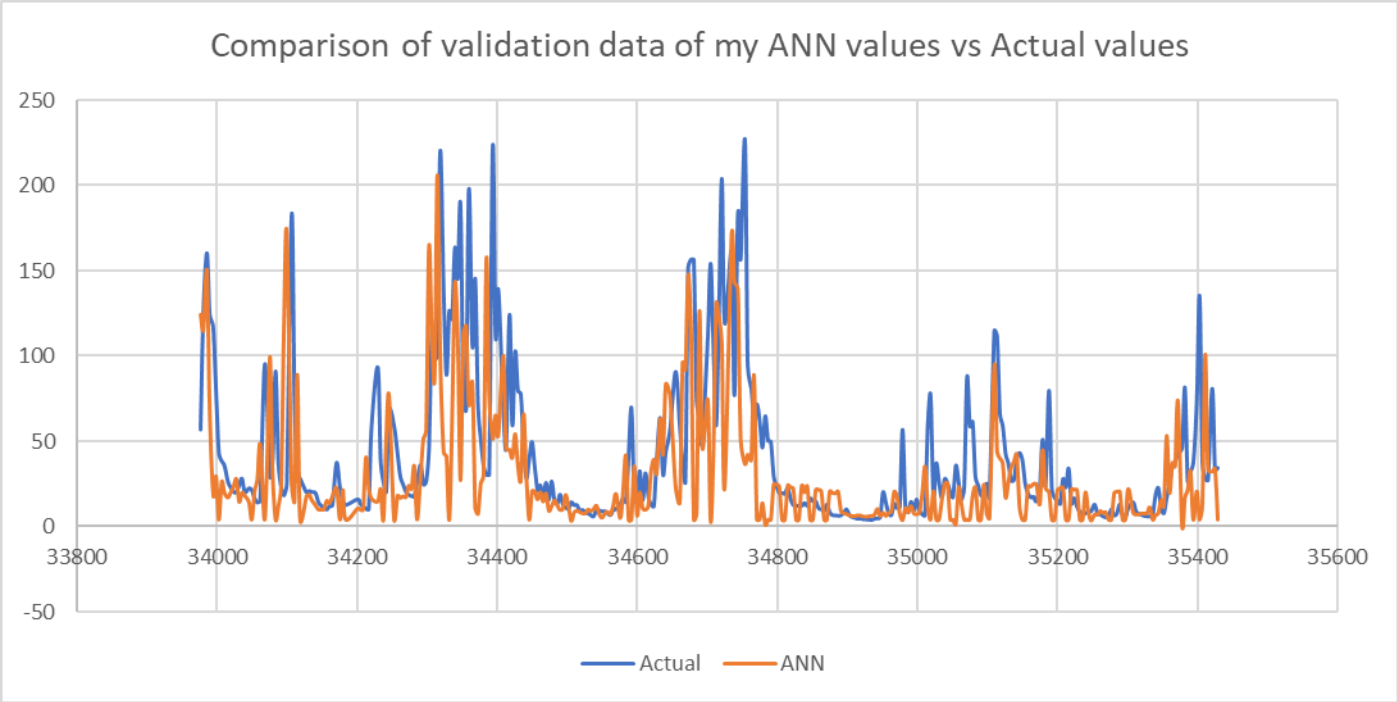
Reflecting on my graphs in Figure 21* and Figure 22[1]my neural network follows trends quite well, but sometimes peaks at the wrong times. In Figure 21 for example, there is a big peak where the actual data does not have one.

I wanted to add as well that there are some points on the graphs where my artificial network will predict really low values; these would be on the dates where I had hardcoded the flow value to be around 4 cusecs if the result predicted by my algorithm was less than 0. An example of where this takes place is in Figure 22 between 34600 and 34800.

The training outcomes also matched up to the actual training data set nicely too, it was able to follow patterns and was able to identify peaks and troths in the data set somewhat reliably. In the next few pages I have attached some graphs which compare the training data outcomes with the actual data set, these graphs will be grouped by years, see Figure 23, Figure 24, Figure 25 and Figure 26.

[1] Larger pictures of Figure 21 and Figure 22 have been attached on the next page


Comparison of test data of my ANN values vs Actual values
250
200
150
100
50
0
33800
34000
34200
34400
34600
34800
35000
35200
35400
35600
Actual
ANN


Comparison of validation data of my ANN values vs Actual values
250
200
150
100
50
0
-50
33800
34000
34200
34400
34600
34800
35000
35200
35400
35600
Actual
ANN

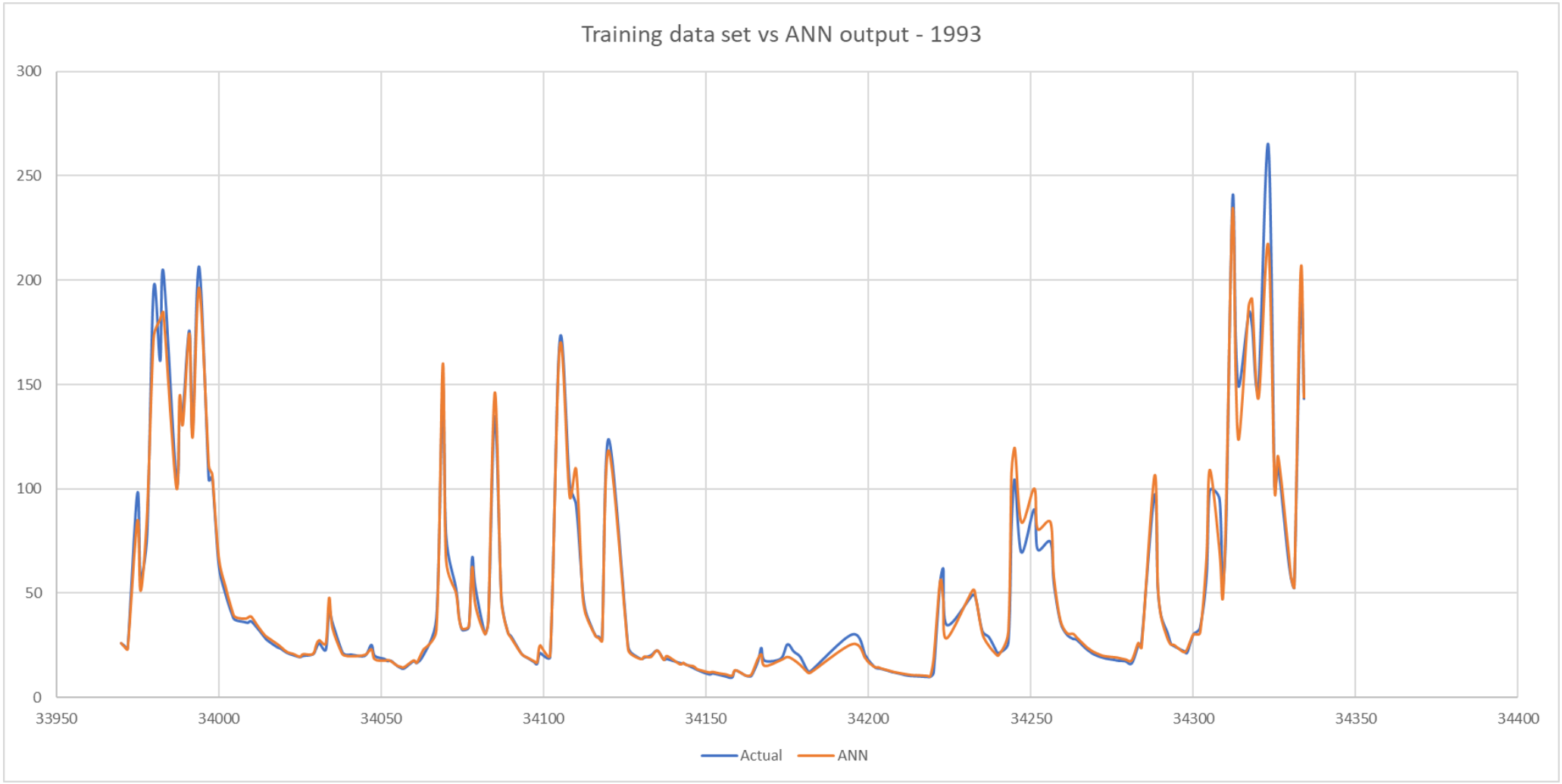


Figure 23 - Training data plotted against my ANN output for the year 1993

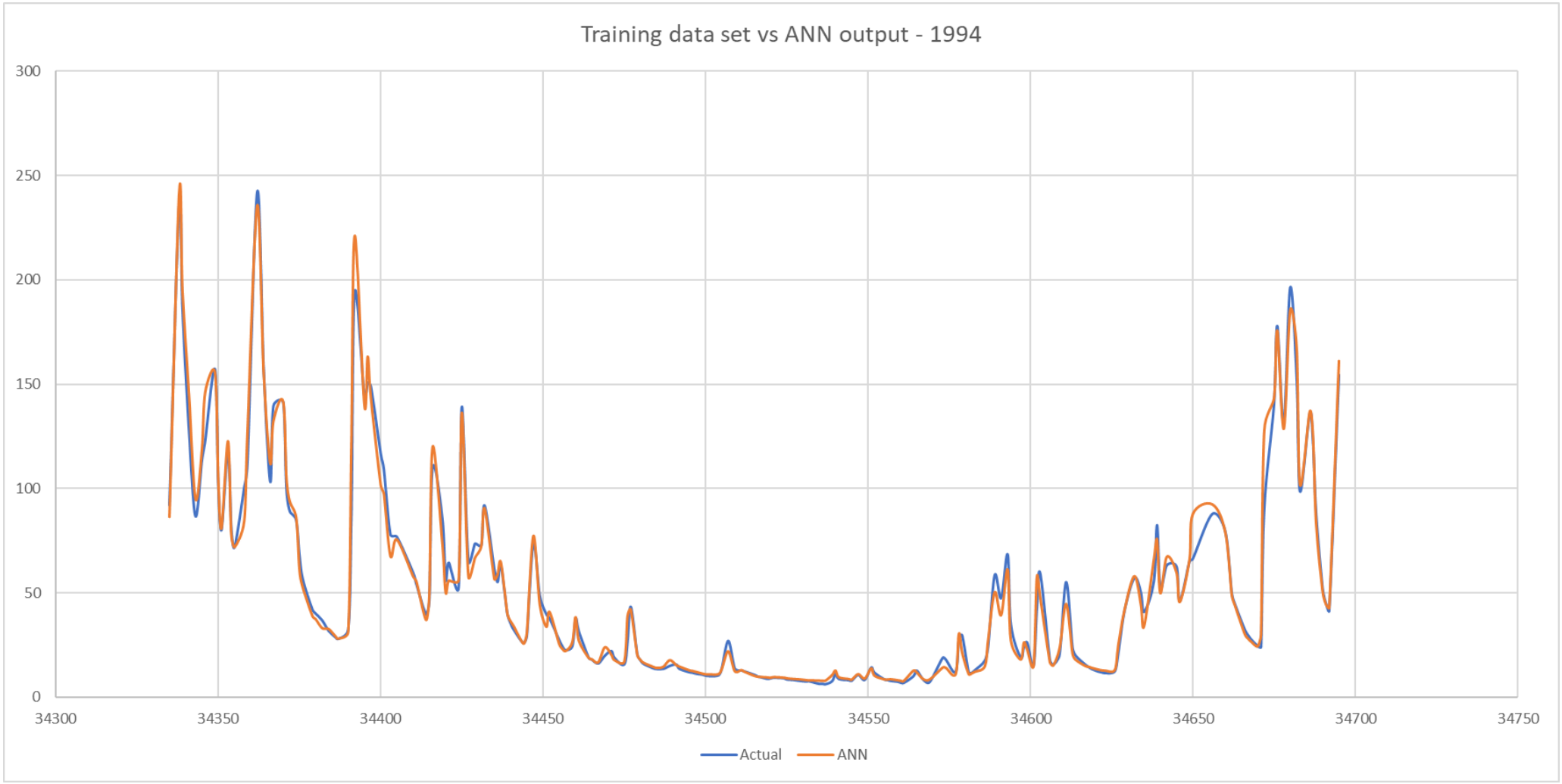


Figure 24 - Training data plotted against my ANN output for the year 1994

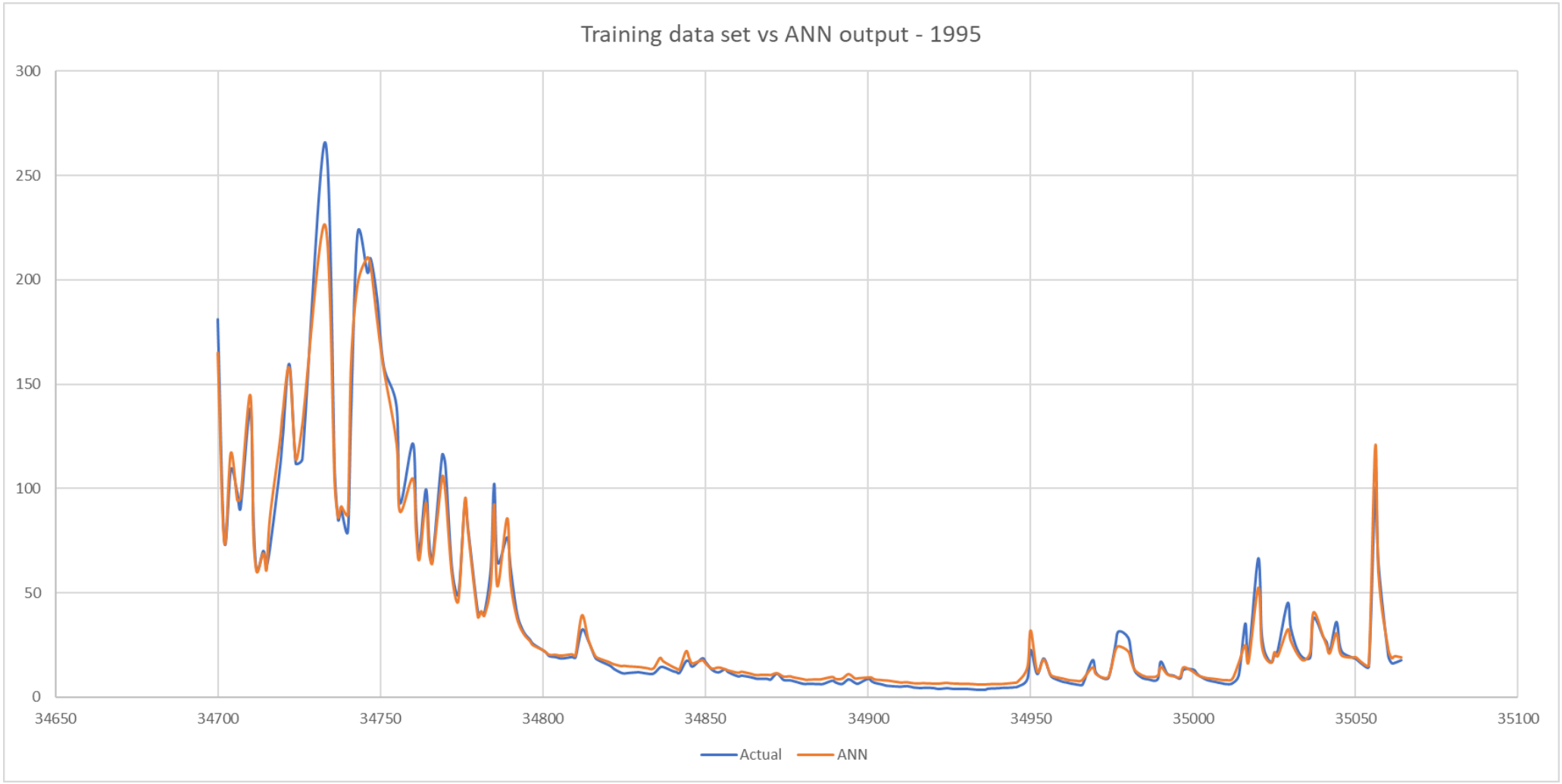


Figure 25 - Training data plotted against my ANN output for the year 1995; it can be observed that my neural network does not peak as much around the middle of feburary

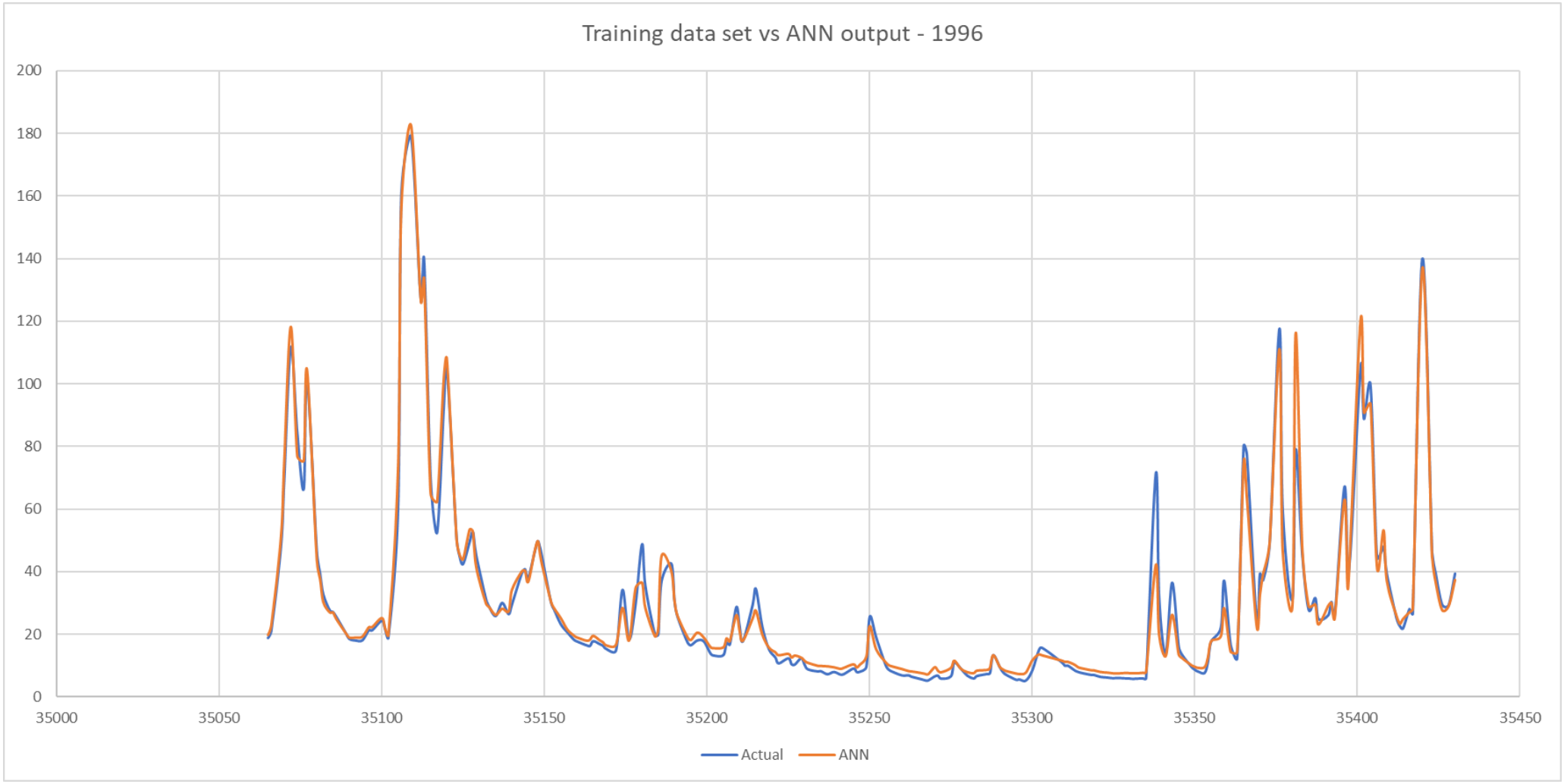


Figure 26 - Training data plotted against my ANN output for the year 1996

Source code

Driver Class

```
package skelton_ai;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.stream.IntStream;
import java.util.ArrayList;
import java.util.List;

public class Driver {
	static int NUMB_OF_EPOCHS = 5000;	// 5000 epochs defined as a constant

	static String[] resultHeaders = new String[] {"Date" , "Skelton"};	// define the headers for the excel file headers here


	public static void main(String[] args) throws IOException {
		NeuralNetwork neuralNetwork = new NeuralNetwork(6);		// construct a network with 4 hidden neurons
		BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));

		List<Double> rmse = new ArrayList<Double>();	// array of doubles to keep track of the overall rmse
		double[] result = new double[TrainingData.TRAINING_DATA.length]; // ann training outputs to be stored here

		boolean flag = true;
		while (flag) {
			System.out.println("> What do you want to do (run, run_validation, run_test, train, exit) ?");
			String command = bufferedReader.readLine();
			switch (command) {
				case "run":
					// Forward propagate  through all training data to get outputs
```

```
                                IntStream.range(0, TrainingData.TRAINING_DATA.length).forEach(i -> result[i] = neuralNetwork

        .fProp(TrainingData.TRAINING_DATA[i][0])
                                                                                                .getLayers()[2].getNeurons()[0]
                                                                                                .getOutput());
                                // params:: result to store, sheetName, Excel sheet headers, fileName, writeType
                                excel_handler.writeToExcel(result, "training result", resultHeaders,"results.xlsx", 0);                // write to
the excel file called "results.xlsx" to the "results" sheet
                                System.out.println("[results ready]");
                                break;

                        case "run_validation":
                                // Forward propagate  through all training data to get outputs
                                IntStream.range(0, TrainingData.TRAINING_DATA.length).forEach(i -> result[i] = neuralNetwork
                                        .fProp(TrainingData.TRAINING_DATA[i][0])
                                        .getLayers()[2].getNeurons()[0]
                                        .getOutput());
                                // params:: result to store, sheetName, Excel sheet headers, fileName, writeType
                                excel_handler.writeToExcel(runners.validate(result), "validation result", resultHeaders,"validation.xlsx", 0);
        // write to the excel file called "results.xlsx" to the "results" sheet
                                System.out.println("[results ready]");
                                break;

                        case "run_test":
                                // Forward propagate  through all training data to get outputs
                                IntStream.range(0, TrainingData.TRAINING_DATA.length).forEach(i -> result[i] = neuralNetwork
                                .fProp(TrainingData.TRAINING_DATA[i][0])
                                .getLayers()[2].getNeurons()[0]
                                .getOutput());
                                // params:: result to store, sheetName, Excel sheet headers, fileName, writeType
                                excel_handler.writeToExcel(runners.test(result), "test result", resultHeaders,"test.xlsx", 0);                //
write to the excel file called "results.xlsx" to the "results" sheet
```

```
                                System.out.println("[results ready]");
                                break;

                        case "train":
                                double[] result_rmse = new double[TrainingData.TRAINING_DATA.length];
                                IntStream.range(0,  NUMB_OF_EPOCHS).forEach(i -> {
                                        System.out.print("[epoch "+i+"]                ");
                                        // for every epoch, go through training set, and for each row in training set forward propagate the
inputs
                                        // and backpropagate the target result
                                        IntStream.range(0, TrainingData.TRAINING_DATA.length).forEach(j -> neuralNetwork

        .fProp(TrainingData.TRAINING_DATA[j][0])

        .backpropError(TrainingData.TRAINING_DATA[j][1][0]));
                                        // get the error on the output neuron and store in result_rmse so that the error can be tracked
                                        IntStream.range(0, TrainingData.TRAINING_DATA.length).forEach(k -> result_rmse[k] =
neuralNetwork
                                                        .getLayers()[2].getNeurons()[0]
                                                        .getOutput());
                                        System.out.println("Error: " + NeuralNetwork.calcRMSE(result_rmse));
                                        rmse.add(NeuralNetwork.calcRMSE(result_rmse));
                                });
                                excel_handler.writeErrors(rmse);          // write the rmse array to excel file so it can be analysed and so a
graph can be made from it
                                System.out.println("[done training]");
                                break;

                        case "exit":
                                flag = false;   // set flag to false, which will exit the while loop which runs program
                                break;
                }
```

```
                }
                System.exit(0);                 // terminate program
        }
}
```

NeuralNetwork Class

```
package skelton_ai;

import java.util.stream.IntStream;
public class NeuralNetwork {
	static final double LEARNING_RATE = 0.01;
	final static int NUMB_OF_INPUT_NEURONS = TrainingData.TRAINING_DATA[0][0].length;	// will become 2 input neurons as we have 2 inputs: the date and the flow of skelton on that given day
	final static int NUMB_OF_OUTPUT_NEURONS = 1;
	private int numbOfHiddenNeurons;
	private Layer[] layers = new Layer[e_layerTypes.values().length];

	public NeuralNetwork(int numbOfHiddenNeurons) {
		/* initialises the number of hidden neurons and will also populate the layers array with each layer type
		 * (an input layer, a hidden layer and an output layer)
		 */
		this.numbOfHiddenNeurons = numbOfHiddenNeurons;
		layers[0] = new Layer(this, e_layerTypes.INPUT);
		layers[1] = new Layer(this, e_layerTypes.HIDDEN);
		layers[2] = new Layer(this, e_layerTypes.OUTPUT);
	}

	public static double calcRMSE(double results[]) {
		double total = 0;
		int dataLength = results.length;
		// loop through all results and compare with target result, get absolute value and calculate rmse from that
		for (int i=0; i<dataLength; i++) {
			total = Math.abs(results[i] - TrainingData.TRAINING_DATA[i][1][0]);
		}
		double returnValue = Math.sqrt(total/dataLength);
		return returnValue;
	}
```

```
	public NeuralNetwork fProp(double input[]) {
		// System.out.println("total neurons: " + (layers[0].getNeurons().length + layers[1].getNeurons().length +
layers[2].getNeurons().length));
		// forward propagate network which runs the network by taking in a vector input
		for (int i = 0; i < layers.length; i++) {
			switch (layers[i].getLayerType()) {
				case INPUT:
					// if we encounter an input layer, we go through all input neurons and set their outputs to the input that is
coming in
					for (int j = 0; j < layers[i].getNeurons().length; j++) {
						layers[i].getNeurons()[j].setOutput(input[j]);
					}

					break;

			case HIDDEN:
				// if we encounter a hidden layer, we go through all hidden neurons and we calculate the neuron's weighted sum
					// which then has the sigmoid activation function applied to it in order to calculate the output for each neuron
				for (int j = 0; j < layers[i].getNeurons().length; j++) {
					double weightedSum = 0;
					for (int k = 0; k < layers[i].getNeurons()[0].getWeights().length; k++)
						weightedSum += layers[i].getNeurons()[j].getWeights()[k] * layers[i-1].getNeurons()[k].getOutput();
					layers[i].getNeurons()[j].applySigmoid(weightedSum);
				}
				break;

			case OUTPUT:
				// if we encounter the output layer, we calculate the weighted sum for the output neuron and pass this
					// weighted sum to the sigmoid activation function which will then give us the output of the neuron
				double weightedSum = 0;
				for (int k = 0; k < layers[i].getNeurons()[0].getWeights().length; k++)
```

```
                                weightedSum += layers[i].getNeurons()[0].getWeights()[k] * layers[i-1].getNeurons()[k].getOutput();
                        layers[i].getNeurons()[0].applySigmoid(weightedSum);
                        break;
                    }
                }
                return this;
        }

        public NeuralNetwork backpropError(double targetResult) {
                Neuron[] inputNeuron = layers[0].getNeurons();
                Neuron[] hiddenNeuron = layers[1].getNeurons();
                Neuron outputNeuron = layers[layers.length-1].getNeurons()[0];
                // calculate error on output neuron and use this error in order to calculate the weight for the output neuron
                outputNeuron.setError((targetResult - outputNeuron.getOutput()) * outputNeuron.calcDerivative());
                for (int j = 0; j < outputNeuron.getWeights().length; j++)
                        outputNeuron.getWeights()[j] = outputNeuron.getWeights()[j] + LEARNING_RATE * outputNeuron.getError() *
hiddenNeuron[j].getOutput();
                        // then we back propagate the error to the hidden neurons
                for (int i = 0; i < hiddenNeuron.length; i++) {
                        hiddenNeuron[i].setError((outputNeuron.getWeights()[i] * outputNeuron.getError()) * hiddenNeuron[i].calcDerivative());
                        // we use this to then calculate the error on the hidden neurons
                        // we use hidden neuron errors to calculate the weight on the hidden neurons
                        for (int j = 0; j < hiddenNeuron[0].getWeights().length; j++)
                                hiddenNeuron[i].getWeights()[j] = hiddenNeuron[i].getWeights()[j] + LEARNING_RATE * hiddenNeuron[i].getError() *
inputNeuron[j].getOutput();
                }
                return this;
        }

        public int getNumbOfHiddenNeurons() {
                return numbOfHiddenNeurons;
        }
```

```
	public Layer[] getLayers() {
		return layers;
	}

	public String toString() {
		// toString method to display information about neural network
		StringBuffer returnValue = new StringBuffer();
		IntStream.range(0, layers.length).forEach(x -> returnValue.append(layers[x] + "  "));
		return returnValue.toString();
	}
}
```

Neuron Class

```
package skelton_ai;

import java.util.stream.IntStream;


public class Neuron {
	private double[] weights = null;
	private double output = 0;
	private double error = 0;
	private e_layerTypes layerType = null;

	public Neuron(e_layerTypes layerType, int numbOfWeights) {
		/*
		 * Initialises a neuron; if the neuron does not belong to the input layer, randomly generate a set of weights for it
		 */
		this.layerType = layerType;
		if (layerType != e_layerTypes.INPUT) {
			weights = new double[numbOfWeights];
			IntStream.range(0, numbOfWeights).forEach(x -> weights[x] = 0.5 - Math.random());
		}
	}

	public void applySigmoid(double weightedSum) {  // apply the Sigmoid transfer function to calculate the output
		output =  1.0 / (1 + Math.exp(-1.0 * weightedSum));
	}

	public double calcDerivative() {	// uses output to calculate derivative
		return output * (1.0 - output);
	}
	public double[] getWeights() {
		return weights;
```

```
	}
	public double getOutput() {
		return output;
	}
	public void setOutput(double output) {
		this.output = output;
	}
	public double getError() {
		return error;
	}
	public void setError(double error) {
		this.error = error;
	}

	public String toString() {
		// toString method to display information about a neuron
		StringBuffer returnValue = new StringBuffer("(" + layerType + ", ");
		if (layerType == e_layerTypes.INPUT) returnValue.append(String.format("%.2f", output)+")");
		else {
			IntStream.range(0, weights.length).forEach(x -> returnValue.append(String.format("%.2f",weights[x])+", "));
			if (layerType == e_layerTypes.HIDDEN) returnValue.append(String.format("%.2f",output)+")");
			else returnValue.append(String.format("%.5f",output)+")");
		}
		return returnValue.toString();
	}
}
```

Layer Class

```
package skelton_ai;

import java.util.stream.IntStream;

public class Layer {          // Represents a layer in the network
                              private Neuron[] neurons = null;        // each layer has an array of neurons that belong to it
                              private e_layerTypes layerType;                 // each layer also has a layer type

                              public Layer(NeuralNetwork neuralNetwork, e_layerTypes layerType) { // initialises layer type, will do different things
for different layer types
                                this.layerType = layerType;
                                switch (layerType) {
                                        case INPUT:
                                                // populate layer's neuron array with input neurons
                                                neurons = new Neuron[NeuralNetwork.NUMB_OF_INPUT_NEURONS];
                                                IntStream.range(0, NeuralNetwork.NUMB_OF_INPUT_NEURONS).forEach(i -> neurons[i] = new
Neuron(layerType, 0));
                                                // System.out.println(neurons.length);
                                                break;
                                        case HIDDEN:
                                                // populate layer's neuron array with hidden neurons
                                                // System.out.println("hidden layer neuron count: " + neuralNetwork.getNumbOfHiddenNeurons());
                                                neurons = new Neuron[neuralNetwork.getNumbOfHiddenNeurons()];
                                                IntStream.range(0, neuralNetwork.getNumbOfHiddenNeurons()).forEach(i ->
                                                                                                        neurons[i] = new
Neuron(layerType,NeuralNetwork.NUMB_OF_INPUT_NEURONS));
                                                break;
                                        case OUTPUT:
                                                // populate layer's neuron array with an output neurons
                                                neurons = new Neuron[NeuralNetwork.NUMB_OF_OUTPUT_NEURONS];
                                                neurons[0] = new Neuron(layerType, neuralNetwork.getNumbOfHiddenNeurons());
```

```
                                    break;
                                }
                            }
                            public Neuron[] getNeurons() {
                                return neurons;
                            }

                            public e_layerTypes getLayerType() {
                                return layerType;
                            }

                            public String toString() {
                                StringBuffer returnValue = new StringBuffer();
                                IntStream.range(0, neurons.length).forEach(x -> returnValue.append(neurons[x] + " "));
                                return returnValue.toString();
                            }
}
```

e_layerTypes Enum Class

```
package skelton_ai;

public enum e_layerTypes {
        INPUT,
        HIDDEN,
        OUTPUT
}
```

data_sets & TrainingData Class

Will be included as a separate pdf – it is a very long class.

excel_handler Class

```
package skelton_ai;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Iterator;
import java.util.List;
import java.util.Arrays;

import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class excel_handler {
	static String DIR_EXCEL = "C:\\Users\\alanw\\OneDrive\\Desktop\\AI coursework\\Spreadsheets\\full_excel.xlsx";
	static int MIN = 33970;
	static int MAX_SKELTON = 320;  // maximum flow of Skelton was 266 on the spreadsheet, so I multiplied 266 by 1.2 to get 320
	static int MAX_CH = 132;   // maximum flow of Crakehill was 110 on the spreadsheet, so I multiplied 110 by 1.2 to get 132
	static int MAX_SB = 54;            // maximum flow of SkipBridge was 45 on the spreadsheet, so I multiplied 45 by 1.2 to get 54
	static int MAX_WW = 162; // maximum flow of WestWick was 135 on the spreadsheet, so I multiplied 135 by 1.2 to get 162

	public static void main(String[] args) {
		getData(DIR_EXCEL, 0);    // print to console the training data set
		//getData(DIR_EXCEL, 1);  // print to console the validation data set
		//getData(DIR_EXCEL, 2);  // print to console the test data set

	}
```

```
	private static void getData(String fileDir, int sheetNo) {
		try {
			File file = new File(fileDir);   //creating a new file instance
			FileInputStream fis = new FileInputStream(file);   //obtaining bytes from the file
			//creating Workbook instance that refers to .xlsx file
			XSSFWorkbook wb = new XSSFWorkbook(fis);
			XSSFSheet sheet = wb.getSheetAt(sheetNo);		//creating a Sheet object to retrieve object
			Iterator<Row> itr = sheet.iterator();		//iterating over excel file
			int counter = 0;
			while (itr.hasNext()) {
				Row row = itr.next();
				Double date = new Double(normaliseDate(row.getCell(0).getNumericCellValue()));		// get date
				Double ckFlow = new Double(normaliseFlow(row.getCell(1).getNumericCellValue(), 0));		// get crakehill flow
				Double sbFlow = new Double(normaliseFlow(row.getCell(2).getNumericCellValue(), 1));		// get skipbridge
flow
				Double wwFlow = new Double(normaliseFlow(row.getCell(3).getNumericCellValue(), 2));		// get west wick flow
				Double target = new Double(normaliseFlow(row.getCell(4).getNumericCellValue(), 3));		// get skelton flow
				// System.out.println("{{" + key + "," + value + "},{" + value + "}},");
				System.out.println("{{" + date + "," + ckFlow + "," + sbFlow + "," + wwFlow + "},{" + target + "}},");
				counter++;

			}
		} catch(Exception e) {
			e.printStackTrace();
		}
	}

	public static void writeToExcel(double[] result, String sheetName, String[] headers, String fileName, int writeType) {
		Workbook workbook = new XSSFWorkbook();
		Sheet sheet = workbook.createSheet(sheetName);
		sheet.setColumnWidth(0, 8560);
		sheet.setColumnWidth(1, 8560);
```

```
                Row header = sheet.createRow(0);

                // create table worksheet headers
                Cell headerCell = header.createCell(0);
                headerCell.setCellValue(headers[0]);
                headerCell = header.createCell(1);
                headerCell.setCellValue(headers[1]);

                // fill rows depending on writeType
                if (writeType==0) {             // Training
                        for(int i=0; i<result.length-1; i++) {
                                Row row = sheet.createRow(i+1);
                                Cell cell = row.createCell(0);
                                cell.setCellValue(denormaliseDate(TrainingData.TRAINING_DATA[i][0][0]));         // de-normalise date and write it
to excel file

                                cell = row.createCell(1);
                                cell.setCellValue(denormaliseFlow(result[i+1], 3));// de-normalise Skelton flow and write it into the excel file
                        }
                } if (writeType==1) {  // Validation
                        for(int i=0; i<result.length-1; i++) {
                                Row row = sheet.createRow(i+1);
                                Cell cell = row.createCell(0);
                                cell.setCellValue(denormaliseDate(data_sets.VALIDATION_DATA[i][0][0]));                 // de-normalise date and
write it to excel file

                                cell = row.createCell(1);
                                cell.setCellValue(denormaliseFlow(result[i+1], 3));// de-normalise Skelton flow and write it into the excel file
                        }

                } if (writeType==2) { // Test
```

```
                for(int i=0; i<result.length-1; i++) {
                        Row row = sheet.createRow(i+1);
                        Cell cell = row.createCell(0);
                        cell.setCellValue(denormaliseDate(data_sets.TEST_DATA[i][0][0]));                    // de-normalise date and write it
to excel file

                        cell = row.createCell(1);
                        cell.setCellValue(denormaliseFlow(result[i+1], 3));// de-normalise Skelton flow and write it into the excel file
                }
        }

        File currDir = new File(".");
        String path = currDir.getAbsolutePath();
        String fileLocation = path.substring(0, path.length() - 1) + fileName;

        try {
                FileOutputStream outputStream = new FileOutputStream(fileLocation);
                workbook.write(outputStream);
                workbook.close();
        } catch(Exception e) {
                e.printStackTrace();
        }
    }

    public static void writeErrors(List<Double> rmse) {
        Workbook workbook = new XSSFWorkbook();
        Sheet sheet = workbook.createSheet("errors");
        sheet.setColumnWidth(0, 8560);
        sheet.setColumnWidth(1, 8560);

        Row header = sheet.createRow(0);
```

```
            // create table worksheet headers
            Cell headerCell = header.createCell(0);
            headerCell.setCellValue("epoch");
            headerCell = header.createCell(1);
            headerCell.setCellValue("error");

            // fill rows
            for(int i=0; i<rmse.size()-1; i++) {
                  Row row = sheet.createRow(i+1);
                  Cell cell = row.createCell(0);
                  cell.setCellValue(i);

                  cell = row.createCell(1);
                  cell.setCellValue(rmse.get(i+1));
            }

            File currDir = new File(".");
            String path = currDir.getAbsolutePath();
            String fileLocation = path.substring(0, path.length() - 1) + "errors.xlsx";
            try {
                  FileOutputStream outputStream = new FileOutputStream(fileLocation);
                  workbook.write(outputStream);
                  workbook.close();
            } catch(Exception e) {
                  e.printStackTrace();
            }
      }

      private static double normaliseDate(double key) {
            // normalises the date value
            double returnValue = ((key - MIN) / 2000) + 0.1;
            return returnValue;
```

```
    }

    private static double denormaliseDate(double key) {
        // denormalises the date value
        double returnValue = ((key - 0.1) * 2000) + MIN;
        return returnValue;
    }

    private static double normaliseFlow(double flow, int river) {
        double returnValue = 0;
        if(river==0) {          // crakehill
            returnValue = ((flow/MAX_CH)*0.8)+0.1;
        } if (river==1) {       // skip bridge
            returnValue = ((flow/MAX_SB)*0.8)+0.1;
        } if (river==2) {       // westwick
            returnValue = ((flow/MAX_WW)*0.8)+0.1;
        } if (river==3) {
            returnValue = ((flow/MAX_SKELTON)*0.8) + 0.1;
        }
        return returnValue;
    }

    private static double denormaliseFlow(double flow, int river) {
        double returnValue = 0;
        if(river==0) {          // crakehill
            returnValue = ((flow - 0.1) / 0.8) * MAX_CH;
        } if (river==1) {       // skip bridge
            returnValue = ((flow - 0.1) / 0.8) * MAX_SB;
        } if (river==2) {       // westwick
            returnValue = ((flow - 0.1) / 0.8) * MAX_WW;
        } if (river==3) {
            returnValue = ((flow - 0.1) / 0.8) * MAX_SKELTON;
```

```
                }
                return returnValue;
        }
}
```

runners Class

```
package skelton_ai;

import java.text.DecimalFormat;

public class runners {
	static double KEY_DIFFERENCE = 0.0005;		// the difference between each normalised date value

	public static double[] validate(double[] result) {
		double[] result_validation = new double[data_sets.VALIDATION_DATA.length];
		/*
		 *  for each date in validation set, find the 3 before it (if possible)		[DONE]
		 *  then find a correlation for those 3 points
		 *  if negative, average the difference and sub
		 *  if positive, avarage the difference and add
		 *
		 */
		DecimalFormat df = new DecimalFormat("#.####");
		double currDate = 0;

		for(int i=0; i<data_sets.VALIDATION_DATA.length; i++) {		// iterate through each row in validation data
			int[] indexSearch = new int [] {0,0,0};
			currDate = Double.parseDouble(df.format(data_sets.VALIDATION_DATA[i][0][0]));		// get the date, but have to
round some as ..01 and ..999 values
			int j = 0;
			for(int k=0; k<TrainingData.TRAINING_DATA.length; k++) {
				for(int n=0; n<10; n++) {		// n < 10 because there may not be 3 consecutive lagged results; have to be
opportunistic in lagging results
													// this is the case because I split the data set using modular
division
					if((Double.parseDouble(df.format(TrainingData.TRAINING_DATA[k][0][0]))-n*KEY_DIFFERENCE) ==
currDate) {
```

```
                                        indexSearch[j] = k;
                                        if (j==2) {
                                                break;
                                        } else {
                                                j++;
                                        }
                                }
                        }
                }
                /* link to the training data set now
                 *      if the list contains NO zeroes (or 3 lagged results), then work out correlation & average
                 */
                if(j==2) {              // if there are 3 lagged results
                        double average = Math.abs(result[indexSearch[0]] - ((result[indexSearch[0]] + result[indexSearch[1]] +
result[indexSearch[2]]) / 3));
                        if(average>result[indexSearch[0]]) {
                                // positive correlation so add average to last value
                                result_validation[i] = result[indexSearch[2]] + average;
                        } else {
                                result_validation[i] = result[indexSearch[2]] - average;
                        }
                }
                if (result_validation[i] < 0.1) {
                        result_validation[i] = 0.11;
                }
        }

        return result_validation;
    }

    public static double[] test(double[] result) {
        double[] result_test = new double[data_sets.TEST_DATA.length];
```

```
		/*
		 *  for each date in validation set, find the 3 before it (if possible)
		 *  then find a correlation for those 3 points
		 *  if negative, average the difference and sub
		 *  if positive, avarage the difference and add
		 *
		 */
		DecimalFormat df = new DecimalFormat("#.#####");
		double currDate = 0;

		for(int i=0; i<data_sets.TEST_DATA.length; i++) {		// iterate through each row in validation data
			int[] indexSearch = new int [] {0,0,0};
			currDate = Double.parseDouble(df.format(data_sets.TEST_DATA[i][0][0]));		// get the date, but have to round some
as ..01 and ..999 values
			int j = 0;
			for(int k=0; k<TrainingData.TRAINING_DATA.length; k++) {
				for(int n=0; n<10; n++) {	// n < 10 because there may not be 3 consecutive lagged results; have to be opportunistic in
lagging results
									// this is the case because I split the data set using modular division
					if((Double.parseDouble(df.format(TrainingData.TRAINING_DATA[k][0][0]))-n*KEY_DIFFERENCE) ==
currDate) {
						indexSearch[j] = k;
						if (j==2) {
							break;
						} else {
							j++;
						}
					}
				}
			}
			/* link to the training data set now
			 *	if the list contains NO zeroes (or 3 lagged results), then work out correlation & average
```

```
                        */
                        if(j==2) {              // if there are 3 lagged results
                                double average = Math.abs(result[indexSearch[0]] - ((result[indexSearch[0]] + result[indexSearch[1]] +
result[indexSearch[2]]) / 3));
                                if(average>result[indexSearch[0]]) {
                                        // positive correlation so add average to last value
                                        result_test[i] = result[indexSearch[2]] + average;
                                } else {
                                        result_test[i] = result[indexSearch[2]] - average;
                                }
                        }
                        if (result_test[i] < 0.1) {
                                result_test[i] = 0.11;
                        }
                }

                return result_test;
        }
}
```