22COB107 MLP Coursework

By Louie McGregor Thomas F121701

Section A: Data Pre-Processing

When performing the prerequisite processing of the raw dataset, both an initial manual approach as well as a latter more programmatic approach was used. In the end, the MLP algorithm would be given 3 proportioned sets (60% - Training, 20% - Validation, 20% - Testing) of cleansed, formatted, standardised data for it to operate on.

When first receiving the Excel dataset file, I manually searched through the data set for any records that seemed erroneous and therefore had to be removed outright. Records that contained any alphabetic characters as well as extreme values such as -999 were expunged rather than corrected to ensure that every record was a valid, actual point of data recorded.

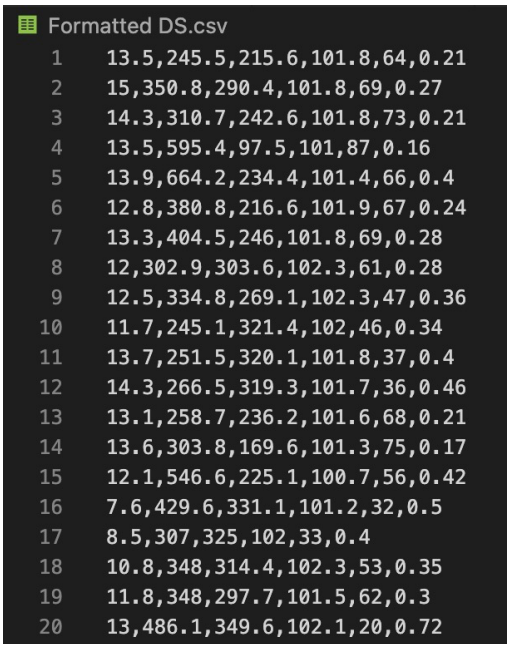
After the manual removal of erroneous records, I prepped the set to be read into the dataPreprocessor.py file by getting rid of the Date column as well as any header/descriptor in the Excel file. After this, I converted the Data Set into a .csv file and started writing the code required to programmatically cleanse and format the data further.

When programming the Data Pre-Processor, I took a modular approach by splitting each individual task into its own function that could then be called by the main GenerateDatasets() function at the end. This final sub-routine could then be called once by the generateANN.py file to provide 3 ‘freshly processed’ sets of training, validation and testing data that could be instantly used in the MLP Algorithm rather than having to perform each task individually.

Reading the Data from the .csv File and Splitting into 3 Distinct Subsets:

ReadData():

The purpose of this function was to take the name of the .csv file in as a parameter and thus open the file; read its contents; format the contents in to a 2 dimensional list containing floats; shuffle the records in the 2D list randomly using *random.shuffle()* (in order to avoid any seasonality trend to effect the MLP algorithm) then finally divide the list into 3 sub lists called ‘training’, ‘validation’, ‘test’ which contained 60%, 20% and 20% of the records stored respectively.



Extract of the 'Formatted DS.csv' file

Cleaning the 3 Data Subsets by removing records containing Outlier Values

After the initial reading and formatting of the data, it needed to be further cleansed to ensure any outlier data wouldn't negatively impact the operation of the MLP algorithm. Thus, the following code aims to programmatically remove any records that contain any outlier fields of data. In order to complete this task, I again made each step into its own subroutine thus decomposing the whole process further.

CalcMean():

The CalcMean() function is the first step in the process to remove any potential outliers from the whole data set. Upon taking in the Training, Validation and Testing sets as one whole concatenated 2D array, a new 1D array named meanSet is created spanning the length of a singular record with each element initialised to zero. The subroutine will then cycle through the records of the data, summing each field of data to its respective position in the meanSet array. Upon finishing this, each element in meanSet is divided by the length of the data set to calculate the mean value of each field of data.

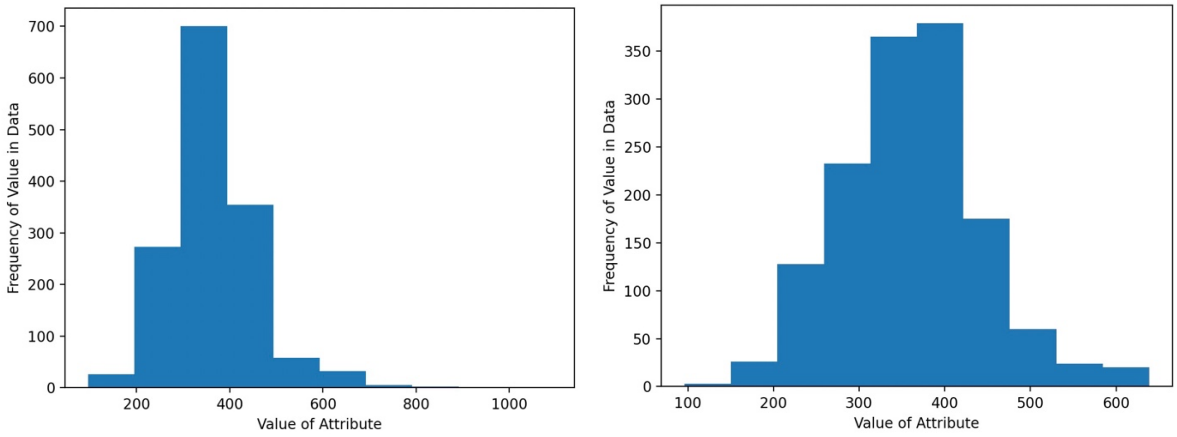
CalcSD():

This subroutine will then be called after *CalcMean(),* and use the meanSet calculated from the last function as well as all 3 data sets to calculate the standard deviation of each field in a similar process to its predecessor. A 1D array is initialised to store the S.D. of each column of data by cycling through the data set, summing the square difference between the points and their mean, then finally square rooting the result of the division of each sum by the

length of the data set. Thus, providing a set of standard deviations to use in order to remove records containing outlier values in the next function.

RemoveOutliers():

The last stage of cleaning the data is the final *RemoveOutliers()* function which takes in the data set, as well as the means and standard deviations calculated prior in order to compare each field in every record of the whole data set. If a record contains a field that lies further than 3 standard deviations away from the mean, it is marked for removal by appending it to a *recordsToRemove* list. After completing the scan through the data, the subroutine will then cycle through which records have been added in the 'removal list' and perform the removal from the dataset by using the built-in Python *.remove()* method. After the removal, the data is again subdivided into their respective sections (Training, Validation and Testing) and returned.



Before/After removing Outlier Records (Attribute: Wind Speed in cm/sec)

Standardising the Records of Data

After the data was cleansed fully, the next stage of the Data Pre-Processing was to standardise each data field into a range of [0.1, 0.9] in order to allow for easier handling by the MLP algorithm. A range of [0.1, 0.9] was chosen rather than [0, 1] in order for the resulting ANNs produced to be able to deal with potential outputs that exceeded the maximum recorded result.

FindMinMax():

This was a simple function to create a 2D array containing the minimum and maximum valued of each field in the data set. Upon being passed the Training + Validation sets as input, the subroutine would linearly pass through the data recording the upper and lower bounds of each column. After the scan, the function would return the resulting 2D 'minMax' array.

StandardiseDatasets():

Ran after the *FindMinMax()* function, *StandardiseDatasets()* would use the generated minimum and maximum values to standardise the data into the specified range. It would have all three data sets (Training + Validation + Testing) passed as one whole set, and after processing each data item, with their appropriate minimum and maximum value, would subdivide the set back into the appropriate proportions to be returned (the minMax array used would also be returned as this may be needed at a later time to destandardise the result as seen later).

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

Formula used to Standardise Data to range [0.1, 0.9]

```
for i in range(len(dataSet)):
    for j in range(len(dataSet[i])):
        dataSet[i][j] = 0.8*((dataSet[i][j] - minMax[j][0])/(minMax[j][1] - minMax[j][0])) + 0.1
```

Formula being implemented to standardise each field in each record

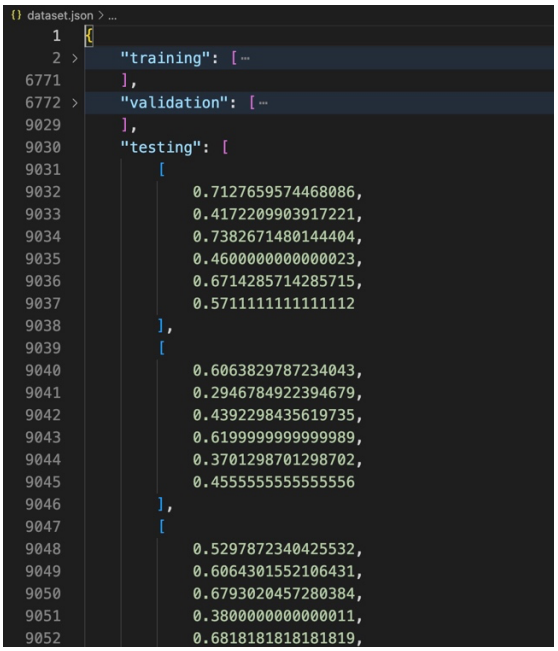
Creating the Final Datasets and Reading/Writing to JSON for Later Use

GenerateDatasets():

With all the previous subroutines performing the formatting, cleansing and standardising of the data separately, the *GenerateDatasets()* function serves as a 'wrapper' function for the main MLP algorithm to call in order to generate and save (to a JSON file) 3 freshly processed data sets that could then be utilised.

ReadDatasetJSON():

This is just a simple function to read any existing pre-processed datasets that are stored in a JSON file. This is so that the same training, validation and testing set can be used multiple times by the MLP algorithm rather than generating a new set each time.



Extract of the JSON file containing the Post-Processed Data Sets

Section B: Implementation of the MLP Algorithm

Thought Process Behind Implementation

Before coding the MLP algorithm directly into Python, I thought about the best possible approach to implement such a complex structure first and thus decomposed the problem into 2 subproblems: the *nodes* and then the *weighted connections* between them.

With regards to the nodes, an OOP approach was taken so that each node's S, δ and u values could be encapsulated within those objects and therefore each one could be accessed via the node itself. The 'input nodes', which did not use an S or δ value, were simply implemented as a different class containing only a u attribute.

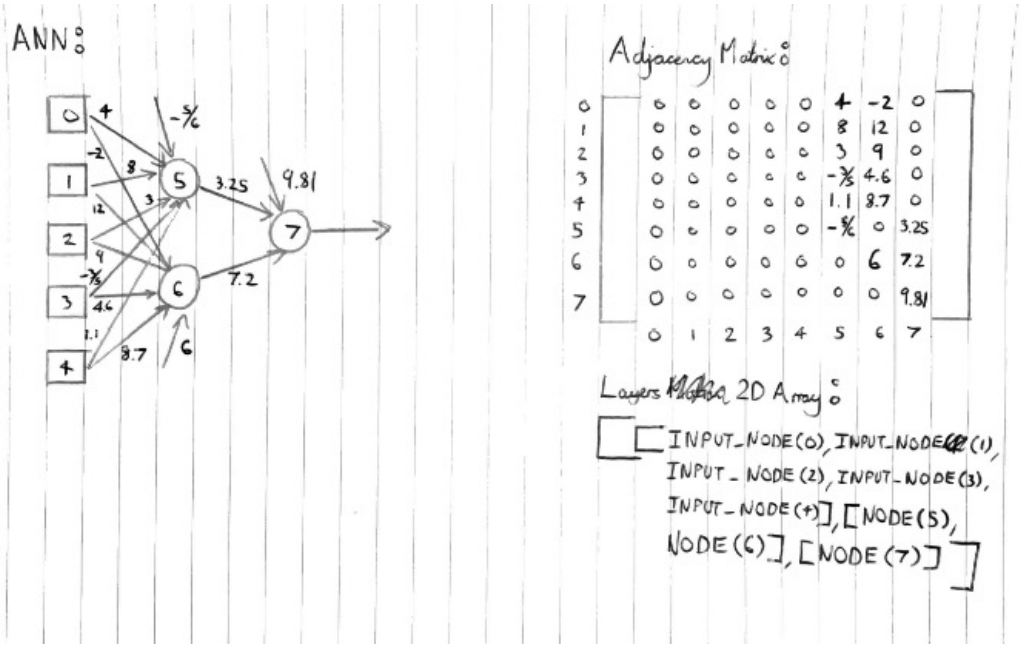
Note: *Inheritance was considered by implementing a general abstract node class to be inherited from, since INPUT_NODE and NODE were quite similar in nature. However, since only two different types of nodes were required, I thought it would generally be easier to just code the two classes separately.*

To store all these *node* objects and make them easy to access, a 2D list called *'Layers'* was created that contained 3 inner 1D lists correlating to each of the 'sections' of the ANN. The

first list containing x amount of *INPUT_NODE* objects (as many as there were predictors), the second having y amount of *NODE* objects (correlating to the amount of hidden nodes the network were to have), the third containing z amount of *NODE* objects correlating to the amount of output nodes there were (*for this coursework, only 1*). Thus, as seen in the sketch, a 5-2-1 ANN (or any x-y-z ANN) could be represented.

At this point, I had a collection of nodes being mapped to each 'layer' of the ANN but no representation of the weighted connections between them as well as each of their biases. To address this, I included a 'number' attribute to each node which identified each one uniquely in the ANN (starting from 0). Then, an adjacency matrix was implemented where each node's number correlated to a row and column in the matrix.

If a connection from node a to node b was present in the ANN, a non-zero (or non-null) value was stored at row a, column b in the matrix to represent the connection's weight (as can be seen in the sketch, *see node 0 to node 5*). If no such connection was present, zero or null is stored at that position. Biases are then represented as 'reflexive' connections in the matrix so, for example, node a's bias weight would be at position $[a, a]$ in the matrix. Again, if no bias exists (i.e. if the node is an *INPUT_NODE*), zero or null is stored at that position instead.



Sketch to Illustrate the Idea to Implement an ANN

```
class inputNode:
    def __init__(self, number):
        self.number = number
        self.u = -1

class node:
    def __init__(self, number):
        self.number = number
        self.delta = -1
        self.S = -1
        self.u = -1
```

Python Code of the Node Classes

Implementation of the Basic MLP Algorithm

When studying the Backpropagation Algorithm in lectures, the process was split into 4 parts which repeated X number of times:

1. Initialise weights and biases to a random starting point.
2. Select a data point from the Training Dataset (this will be chosen linearly).
3. Perform a Forward Pass on the datapoint and calculate the output.
4. Calculate the error between the predicted value and the actual value, then perform a Backward Pass adjusting the weights accordingly.

Part 1: Initialisation

The first task of initialising the different components in the MLP Algorithm was relatively easy as it involved just initialising the arrays and objects discussed earlier. The learning parameter ρ was set to 0.1 as this was a good basis to start off from (not too big as to oscillate, not too small as to be trapped by local minima). With the implementation of Bold Driver later, the value of ρ could be changed as the algorithm is being executed.

The Adjacency Matrix that was mentioned in the section prior would be initialised with default values of *None* rather than *0*, to prevent any accidental interpretation by the algorithm that a weight had the value of 0. Therefore, if any bug occurred with reading a weight that didn't actually exist, an *NoneType* exception would be thrown to alert me rather than the program just using the value of 0 and carrying on. Any 'valid' weight and bias on the Adjacency Matrix was first assigned a random value between $\pm 2/n$ where n is the amount of input nodes the ANN will have (correlating to the number of predictors used).

```
#Setting the Learning Rate to 0.1 initially
rho = 0.1

#Defining how many nodes in total the network has (n inputs + m hidden nodes + 1 output)
numberOfNodes = len(dataSet[0])+amountOfHiddenNodes

#Create the individual nodes in the network  (inputs -> hidden layer -> output node)
layers = [[classes.inputNode(i) for i in range(len(dataSet[0])-1)],

        [classes.node(i) for i in range(len(dataSet[0])-1, numberOfNodes-1)],

        [classes.node(numberOfNodes-1)]]

#Create Weight/Bias Matrix
adjMatrix =[[None for i in range(numberOfNodes)] for j in range(numberOfNodes)]
```

```
#Initialise Weights
for i in range(len(layers)-1):
    for j in range(len(layers[i])):
        thisNode = layers[i][j]
        for k in range(len(layers[i+1])):
            nextNode = layers[i+1][k]
            adjMatrix[thisNode.number][nextNode.number] = random.uniform(-2/(len(dataSet[0])-1),2/(len(dataSet[0])-1))

#Initialise Biases
for i in range(1, len(layers)):
    for j in range(len(layers[i])):
        thisNode = layers[i][j]
        adjMatrix[thisNode.number][thisNode.number] = random.uniform(-2/(len(dataSet[0])-1),2/(len(dataSet[0])-1))
```

Extracts of the GenerateANN() code showcasing the initialisation of variables and assignment of values.

Part 2: Forward Pass Implementation

Programming the Forward Pass part of the algorithm meant initially setting the input nodes' u value to be each of the inputs of the currently chosen data point respectively. Then performing a loop starting from the first 'hidden' layer to (and including) the final 'output' layer which, at each iteration, calculated every one of their containing node's S value as the weighted sum of the u values of its preceding layers' nodes. After a node's S value was computed, the relevant activation function was performed on it to then set that node's u value (the activation function used was either Sigmoid or $\tanh$, depending on which config was selected to train the ANN).

```
#Defining the Forward/Backward Pass
def ForwardPass(dataPoint):
    '''
        Forward Pass is simply inputting values, then calculating the output
    '''
    InsertInputs(dataPoint)
    CalcOutput()
```

```
#Procedure to set the input nodes 'u' value to the appropriate inputs
def InsertInputs(dataPoint):
    for i in range(len(layers[0])):
        layers[0][i].u = dataPoint[i]

#Procedure to calculate the 'u' value of each node with the current inputs
def CalcOutput():
    for i in range(1, len(layers)): #For each layer in the network, from the hidden layer to the output layer...

        for Node in layers[i]: #For each node in the current layer...

            temp = 0

            for prevNode in layers[i-1]: #For each node in the previous layer...
                temp += prevNode.u * adjMatrix[prevNode.number][Node.number] #Add up each of the 'predecessor' nodes weighted outputs

            temp += adjMatrix[Node.number][Node.number] #Add the nodes bias

            Node.S = temp #Set the resulting sum as the 'S' value of the node

            #Perform the Transfer function on the new 'S' value to generate a new 'u' value for the node
            if (useTanh): #If we are using tanh as Transfer...
                Node.u = ((math.e**Node.S)-(math.e**-Node.S))/((math.e**Node.S)+(math.e**-Node.S))
            else: #If we are using a normal Sigmoid function
                Node.u = 1/(1+(math.e**(-Node.S)))
```

Extracts of the GenerateANN() code illustrating the decomposition then implementation of the Forward Pass

Part 3: Backward Pass Implementation

Out of all the 'phases' of the Backpropagation Algorithm, coding the Backward Pass was the most challenging aspect due to it essentially being made up of two parts: the initial calculating of δ for each node (this being different for output nodes than normal hidden nodes), then the appropriate adjusting of weights and biases for each node after.

```
def BackwardPass(dataPoint):
    '''
        Backward Pass is taking in the datapoint, comparing the observed output
        to the modelled output and updating the weights and biases
    '''
    CalcDelta(dataPoint)
    UpdateWeights()
    UpdateBiases()
```

The BackwardPass() function is a wrapper function calling 3 individual tasks to be completed.

Implementing the *CalcDelta()* function included having a nested for loop to visit each node in reverse order to the ANN starting from the output node to the hidden nodes. At the output nodes, the δ value was computed via the expression $(C - u_o)f'(S_o)$ where $(C - u_o)$ is the error between the predicted output and the actual output and $f'(S_o)$ is the derivative of the activation function being used on the weighted sum of inputs.

At the hidden nodes, the δ value was calculated using $w_{j,O}\delta_O f'(S_j)$ where: $w_{j,O}$ is the weight of the edge from that hidden node to the output node, δ_O is the δ value of the output node (as this would be calculated first), and again $f'(S_O)$ is the derivative of the activation function being used on the weighted sum of that node's inputs.

The derivative activation function $f'(x)$ would of course vary depending on whether $tanh$ or a Sigmoid function was being used, or a simple linear function (restricted to only output nodes).

```
#Procedure to calculate each node's delta value
def CalcDelta(dataPoint):
    for i in range(len(layers)-1, 0, -1): #For each layer in the network, from the output layer to the hidden layer...

        for j in range(len(layers[i])): #For each node in the current layer

            if i == len(layers)-1: #First, calculate the output node's delta value...

                if (useWeightDecay): #If we are using weight decay...…

                else: #If not...
                    if (useLinearInOutput):
                        layers[i][j].delta = dataPoint[5] - layers[i][j].u
                    elif (useTanh):
                        layers[i][j].delta = (dataPoint[5] - layers[i][j].u) * (1 - layers[i][j].u**2)
                    else:
                        layers[i][j].delta = (dataPoint[5] - layers[i][j].u) * (layers[i][j].u * (1-layers[i][j].u))


            else: #Then, calculate all the hidden node deltas
                if(useTanh):
                    layers[i][j].delta = adjMatrix[layers[i][j].number][layers[i+1][0].number] * layers[i+1][0].delta * (1-(layers[i][j].u**2))
                else:
                    layers[i][j].delta = adjMatrix[layers[i][j].number][layers[i+1][0].number] * layers[i+1][0].delta * (layers[i][j].u*(1-layers[i][j].u))
```

Extract of CalcDelta()

After *CalcDelta()* would run its course, updating the Adjacency Weight Matrix accordingly would be the next step in the Backward Pass. This would be done by simply adding to the correct entry in the matrix (i.e. row i, column j meaning the weight of node i -> node j), $\rho\delta_j u_i$ where: ρ is the learning parameter that was mentioned earlier in the initialisation phase, δ_j would be the delta value of node j, and u_i would be the u value of node i.

For biases, this was made easier by just adding to each entry $[i, i]$ where i is the hidden/output node in question: $\rho\delta_i$.

```
#Procedure for updating the weights between nodes
def UpdateWeights():
    for i in range(len(layers)-1): #From the input layer to the hidden layer (inclusive)...
        for j in range(len(layers[i])): #...grab each node individually...
            Node = layers[i][j]
            for nextNode in layers[i+1]: #...for every node in front of it, update the weight between them
                #If using Momentum...
                if (useMomentum):…
                else:
                    adjMatrix[Node.number][nextNode.number] += rho * nextNode.delta * Node.u

#Procedure for updating the biases of each node
def UpdateBiases():
    for i in range(1, len(layers)): #From the hidden layer to the output layer (inclusive)...
        for j in range(len(layers[i])): #...grab each node individually and update their bias on the weight matrix
            Node = layers[i][j]
            #If using Momentum...
            if (useMomentum): …
            else:
                adjMatrix[Node.number][Node.number] += rho * Node.delta
```

Extracts of UpdateWeights() and UpdateBiases()

Summary

With these 3 stages implemented, the algorithm could easily be completed by initialising the values first then running a for loop to iterate a specified number of epochs which, at each iteration, cycled through the dataset point-by-point performing a forward pass then a backward pass on each data point. The training duration could then be specified by the user for X number of epochs.

Upon finishing the training of the neural network, another class was used called ANN to encapsulate the nodes (in the layer 2D list) as well as the adjacency matrix into a single object that could be returned and therefore used on validation or testing data, or simply saved to a JSON file.

```
#Train the ANN
for i in range(limit):
    epochsSoFar += 1

    #If using Bold Driver, every so often take an error reading and compare to previous error reading
    if (useBoldDriver and i % (limit//30) == 0 and i != 0):…

    #If not, train as usual
    else:
        for dataPoint in dataSet:
            ForwardPass(dataPoint)
            BackwardPass(dataPoint)

    #Progress Identifier
    print(f"Training at: {i/limit*100}%", end="\r")
print(end="\n\n")

#Encapsulate the ANN into an object then return
return classes.ANN(layers,adjMatrix, useLinearInOutput, useTanh, limit, useMomentum, useBoldDriver, useWeightDecay)
```

Extract of GenerateANN() to show the training loop and subsequent encapsulating of the neural network.

Implementation of Momentum

Momentum was a relatively easy addition to the algorithm as the improvement required only the addition of another term: $\alpha\Delta w_{i,j}$ where α is merely the momentum coefficient (typically set to 0.9) and $\Delta w_{i,j}$ is the difference in the previous two weights.

In order to implement this feature, I included another 2D matrix that had the same dimensions of the Weight Adjacency Matrix. This new structure, dubbed ‘Momentum Matrix’, was to store the weight changes at each edge (similar to how the Weight Adjacency Matrix stored the current weight of each edge).

```
#Create Weight/Bias Matrix
adjMatrix =[[None for i in range(numberOfNodes)] for j in range(numberOfNodes)]

#Create Momentum Matrix (stores the weight changes)
if (useMomentum):
    momentumMatrix = [[0 for i in range(numberOfNodes)] for j in range(numberOfNodes)]
```

Extract depicting the Initialisation of the Momentum Matrix. Values are initially set to zero as to still allow computation of the first weight change.

Due to the Momentum Matrix being of the same dimensions and layout as the Adjacency Matrix, the same coordinates used to reference the edge weight could then be used to reference the weight change. After calculating the new weight, the appropriate entry in the Momentum Matrix would then be updated to the difference of the newly calculated value and the previous value stored.

```
#If using Momentum...
if (useMomentum):
    oldWeight = adjMatrix[Node.number][nextNode.number]
    newWeight = adjMatrix[Node.number][nextNode.number] + (rho * nextNode.delta * Node.u) + (0.9 * momentumMatrix[Node.number][nextNode.number])

    adjMatrix[Node.number][nextNode.number] = newWeight

    momentumMatrix[Node.number][nextNode.number] = newWeight - oldWeight
```

Extract showcasing the process of calculating the new weight with the addition of the Momentum term, then the subsequent updating of the weight change.

Implementation of Bold Driver

Compared to the traditional training cycle of having a pre-defined immutable value for the learning parameter ρ, Bold Driver involves a regular but not excessive checking of the error that an ANN produces while training (typically via the use of a function like MSE). Therefore, to implement this, a modification to the *GenerateANN()* subroutine was made to allow for a alternative training method that accommodated Bold Driver.

If Bold Driver is active, upon initialisation of the nodes and weights, the algorithm will perform an initial error reading on the ANN before training via the MSE function (as this is a simple function to use and compute). Then, every so often in the training cycle, another error reading will take place. If the new error reading is more than 4% less than the old reading, the learning parameter will increase by 5% and the new reading will be stored to compare with the next eventual reading.

However, if the error is greater than 1.04x the old reading then the weights are reset, the learning parameter is reduced by 30% and the training pass is rerun. This will repeat while the error is greater than the old one, but a forced break of the while loop will occur if the learning parameter is set to below 0.01. As a result, limits are also place on how high/low ρ can be (i.e. [0.01, 0.5]).

```
#While the New Error is worse than the old one, reset the weights/biases, decrease the learning rate and rerun the cycle
while (MSE > (oldMSE * 1.04)):
    print("Learning Rate Too Large. Decreasing by 30 percent and Rerunning...")
    rho = 0.7 * rho

    if (rho < 0.01):
        rho = 0.01
        print(f"Rho: {rho}")
        break

    print(f"Rho: {rho}")

    adjMatrix = oldWeights

    if (useMomentum):
        momentumMatrix = oldMomentum

    for dataPoint in dataSet:
        ForwardPass(dataPoint)
        BackwardPass(dataPoint)

    MSE = 0
    for dataPoint in dataSet:
        ForwardPass(dataPoint)

        observed = (dataPreprocessor.DestandardiseResult(layers[2][0].u, minMax))
        modelled = dataPreprocessor.DestandardiseResult(dataPoint[5],minMax)
        MSE += (observed-modelled)**2

    MSE = MSE/len(dataSet)
```

```
#If the New Error is signifcantly better than the old one, increase the learning rate
if (MSE < oldMSE * 0.96):
    print("Error Function Decreased. Increasing Learning Rate by 5%...")
    rho = 1.05 * rho
    if (rho > 0.5):
        rho = 0.5
    print(f"Rho: {rho}")
```

Extracts showing the comparison of the error readings taken and the modification of the learning parameter consequently.

Implementation of Weight Decay

$$\delta_O = (C - u_O + \upsilon\Omega) f'(S_O)$$

$$\upsilon = \frac{1}{\rho e}, \quad \Omega = \frac{1}{2n} \Sigma_{i=1}^{n} w_i^2$$

The Weight Decay modification to the equation calculating the δ value of the Output Node.

In order to accommodate the modification of Weight Decay, a counter had to be implemented within the training loop that could be accessed by the *CalcDelta()* function which served only to count how many epochs the training algorithm had completed in training thus far (e). When *CalcDelta()* is called and if it is currently computing the output node's δ value, υ is calculated first by taking the reciprocal of the product between the learning parameter ρ and the epoch counter e. Then, to calculate Ω, a loop thorough Weight Adjacency Matrix is performed to sum the square of any non-null value stored, after which the entire sum is divided by 2 times the amount of non-null weights (this is kept track by another counter when looping through the Matrix).

After the prerequisite terms are found, the modified δ value for the output node is computed with the above equation.

```
if (useWeightDecay): #If we are using weight decay...

    upsilon = 1/(rho*epochsSoFar)

    Omega = 0
    counter = 0

    for x in range(len(adjMatrix)):
        for y in range(len(adjMatrix[i])):
            if adjMatrix[x][y] != None:
                Omega += adjMatrix[x][y]**2
                counter += 1

    Omega /= 2*counter

    if (useLinearInOutput):
        layers[i][j].delta = dataPoint[5] - layers[i][j].u + (upsilon*Omega)
    elif (useTanh):
        layers[i][j].delta = (dataPoint[5] - layers[i][j].u + (upsilon*Omega)) * (1 - layers[i][j].u**2)
    else:
        layers[i][j].delta = (dataPoint[5] - layers[i][j].u + (upsilon*Omega)) * (layers[i][j].u * (1-layers[i][j].u))
```

Code extract showing the implementation of Weight Decay into the MLP Algorithm

Implementation of Annealing

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

The Annealing Equation to modify the value of ρ at each epoch.

The implementation of Annealing into the MLP algorithm was a simple recalculation of ρ's value using the formula above where:

$p =$ The 'End Value' of Rho
$q =$ The 'Starting Value' of Rho
$r =$ The Epoch Limit
$x =$ The amount of Epochs executed thus far

Since, as stated in previous sections, the method of training an ANN was a for loop iterating a predefined number of times continuously performing a Forward and Backward pass at each cycle, the values of r and x were already easy to access. For p and q, these could be initially set at the start of the initialisation process if Annealing was enabled.

```
if (useAnnealing):
    startRho = float(input("Please set a start value for Rho: "))
    endRho = float(input("Please set an end value for Rho: "))
```

```
if (useAnnealing):
    rho = endRho + ((startRho-endRho) * (1 - (1 / (1 + math.e ** (10 - (20*i/limit))))))
    print(f"Rho: {rho}")
```

Code showcasing the implementation of the Annealing modification (initialisation of values – top, calculation of rho – bottom)

***Note:** As Bold Driver and Annealing both modify the value of ρ during training, they cannot be both enabled at the same time (to avoid inconsistencies). If both are detected as enabled, the algorithm will alert the user and ask for which method to disable.*

Implementation of Batch Learning

As Batch Learning was a change to the core method of training an ANN, by editing weights after each epoch rather than after each data point, this modification was by far the hardest to program.

In the traditional training cycle, every forward pass through a data point would be subsequently followed with a Backward Pass which calculated each node's δ value and then using that value to edit each of the weights and biases in the network via the following equation:

$$w_{i,j}^{*} = w_{i,j} + \rho\delta_j u_i$$

In Batch Learning however, rather than calculating the new weights and biases after the computation of δ at each Backward Pass, the values of δ and the nodes' u would be multiplied together for each data point and summed to a corresponding position in a 2D array where each position in the array corresponded to which side those terms would eventually be added to (similar to the adjacency matrix and the momentum matrix).

Then, at the end of each epoch, each weight and bias would be modified by adding the appropriate value in the Batch Learning 2D Array multiplied by ρ and divided by the size of the Training Data set (as to provide an average). This is seen in the following equation:

$$w^*_{i,j} = w_{i,j} + \rho \frac{1}{N} \Sigma^N_{k=1} \delta_j u_i$$

Then, at the start of the next epoch, all the values in the 2D array are reset to zero for the next cycle through the data set.

Note: *Using purely the sum of $\delta_j u_i$ rather than the average was trialled in training ANNs with Batch Learning and whilst it reduced error at a faster rate, it did also make the training erratic and produce ANNs with varying levels of effectiveness. Therefore, an average was favoured.*

```
#Initialise 2D array to store sums of delta_j * u_i (row i and col j)
batchArray = [[0 for x in range(numberOfNodes)] for y in range(numberOfNodes)]
```

```
#Perform a Forward Pass, summing delta and u
for dataPoint in dataSet:
    ForwardPass(dataPoint)
    BatchCalcDelta(dataPoint)

#Edit the weights of the network with the sums
BatchEditWeights()
```

```
def BatchCalcDelta(dataPoint):
    #Calculate the Delta of each node
    CalcDelta(dataPoint)

    #For each of the edges, add onto the corresponding batch array entry j's delta and i's u
    for j in range(len(layers)-1):
        for node in layers[j]:
            for nextNode in layers[j+1]:
                batchArray[node.number][nextNode.number] += nextNode.delta * node.u

    #For each of the biases, add onto the corresponding batch array entry i's delta
    for j in range(1, len(layers)):
        for node in layers[j]:
            batchArray[node.number][node.number] += node.delta
```

```
#Subroutines for Batch Learning
def BatchEditWeights():
    for j in range(len(layers)-1):
        for Node in layers[j]:
            for nextNode in layers[j+1]:
                if (useMomentum):…
                else:
                    adjMatrix[Node.number][nextNode.number] += rho * batchArray[Node.number][nextNode.number]/len(dataSet)

    for j in range(1, len(layers)):
        for Node in layers[j]:
            if (useMomentum):…
            else:
                adjMatrix[Node.number][Node.number] += rho * batchArray[Node.number][Node.number]/len(dataSet)
```

Code depicting the Implementation of the 2D array used for Batch Learning, and the modified Backward Pass functions used in Training.

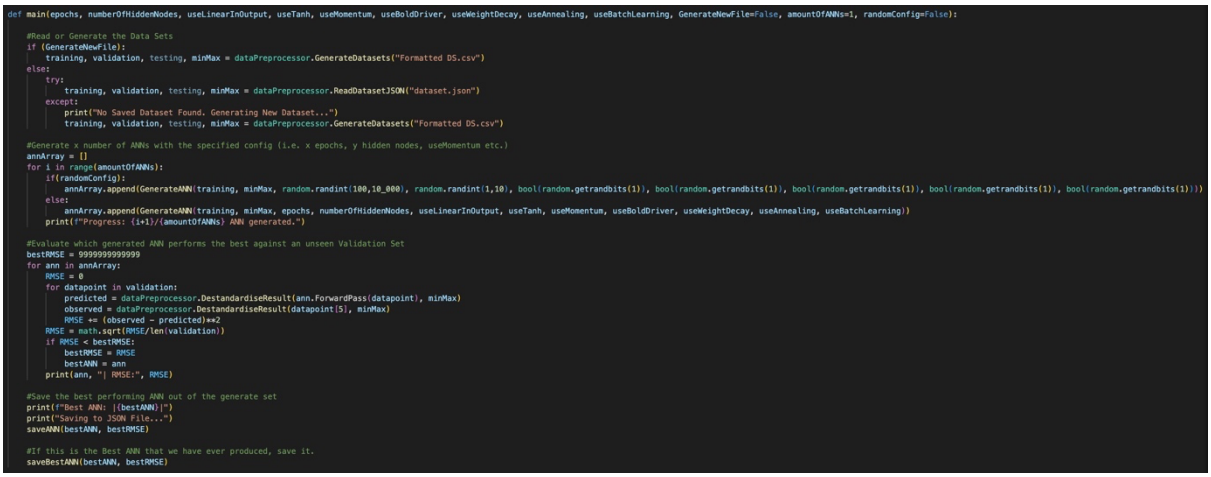
Section C: Network Training and Selection

Explanation of Network Training and Selection Process:

When generating neural networks, the algorithm allows a certain number of Boolean parameters to be set prior to execution to enable/disable certain modifications (i.e. useMomentum = False, useBoldDriver = True). As well as this, the user can specify the number of neural networks (e.g. 1, 5, 10, 20 etc.) to be generated from the training set in one execution after which they are all evaluated against the Validation Set. The neural network generated from that set with the best performance in the Validation Set (i.e. lowest RMSE) is then selected as the optimum ANN with the chosen parameters and saved to a JSON file which can then be later read and recreated to be used on the Testing Set (similar idea to Natural Selection in Genetic Algorithms). If a produced ANN has the lowest RMSE we have seen, it is saved as the best produced neural network.

All graphs and such that will be seen in this section are Neural Networks that performed the best against the Validation Set after generation and then performed against the Testing Set.

Note: *For each of the below experiments, 20 ANNs were generated for every different configuration and thus the best performing one was selected as the representative for those particular settings.*



```
def main(epochs, numberOfHiddenNodes, useLinearInOutput, useTanh, useMomentum, useBoldDriver, useWeightDecay, useAnnealing, useBatchLearning, GenerateNewFile=False, amountOfANNs=1, randomConfig=False):

    #Read or Generate the Data Sets
    if (GenerateNewFile):
        training, validation, testing, minMax = dataPreprocessor.GenerateDatasets("Formatted DS.csv")
    else:
        try:
            training, validation, testing, minMax = dataPreprocessor.ReadDatasetJSON("dataset.json")
        except:
            print("No Saved Dataset Found. Generating New Dataset...")
            training, validation, testing, minMax = dataPreprocessor.GenerateDatasets("Formatted DS.csv")

    #Generate x number of ANNs with the specified config (i.e. x epochs, y hidden nodes, useMomentum etc.)
    annArray = []
    for i in range(amountOfANNs):
        if(randomConfig):
            annArray.append(GenerateANN(training, minMax, random.randint(100,10_000), random.randint(1,10), bool(random.getrandbits(1)), bool(random.getrandbits(1)), bool(random.getrandbits(1)), bool(random.getrandbits(1)), bool(random.getrandbits(1))))
        else:
            annArray.append(GenerateANN(training, minMax, epochs, numberOfHiddenNodes, useLinearInOutput, useTanh, useMomentum, useBoldDriver, useWeightDecay, useAnnealing, useBatchLearning))
        print(f"Progress: {i+1}/{amountOfANNs} ANN generated.")

    #Evaluate which generated ANN performs the best against an unseen Validation Set
    bestRMSE = 9999999999999
    for ann in annArray:
        RMSE = 0
        for datapoint in validation:
            predicted = dataPreprocessor.DestandardiseResult(ann.ForwardPass(datapoint), minMax)
            observed = dataPreprocessor.DestandardiseResult(datapoint[5], minMax)
            RMSE += (observed - predicted)**2
        RMSE = math.sqrt(RMSE/len(validation))
        if RMSE < bestRMSE:
            bestRMSE = RMSE
            bestANN = ann
        print(ann, "| RMSE:", RMSE)

    #Save the best performing ANN out of the generate set
    print(f"Best ANN: |{bestANN}|")
    print("Saving to JSON File...")
    saveANN(bestANN, bestRMSE)

    #If this is the Best ANN that we have ever produced, save it.
    saveBestANN(bestANN, bestRMSE)
```

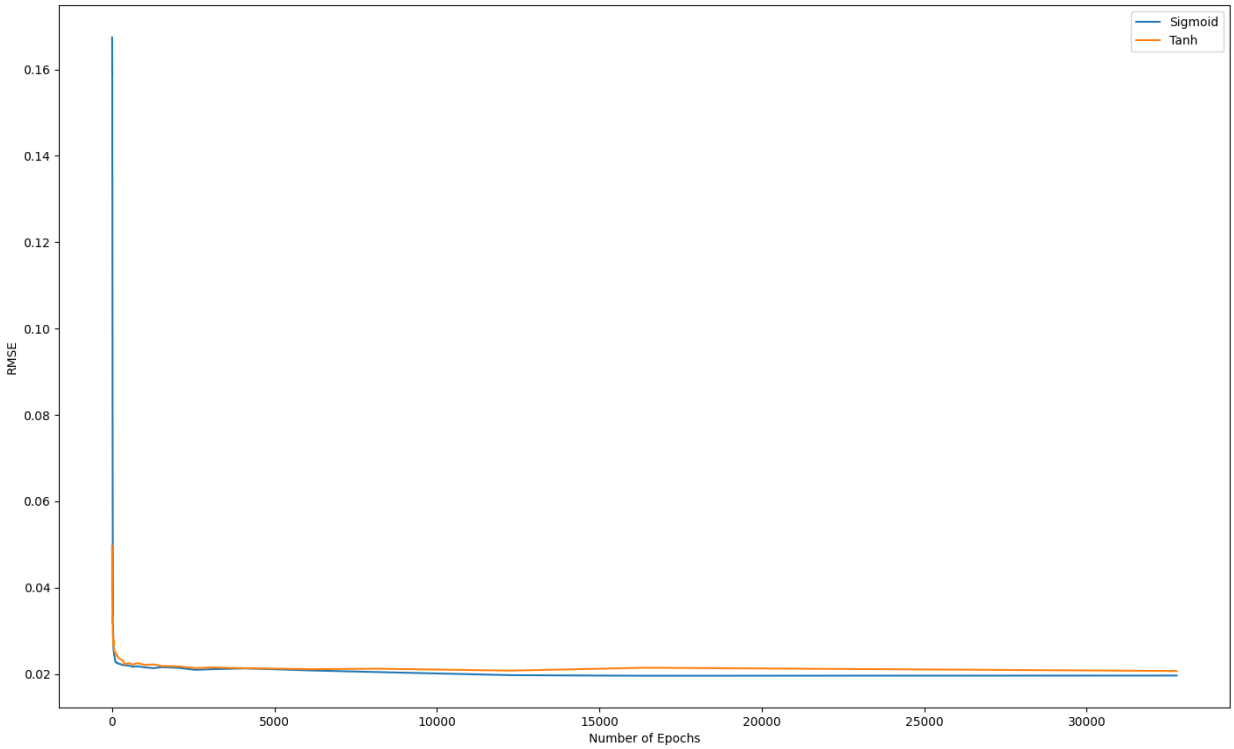
Extract of the code showing the production of ANNS using set configurations with GenerateANN()

Effect of Varying Number of Epochs:

One of the first two experiments I wanted to try with the Neural Network code was to see the effect of gradually increasing the number of epochs that the ANNs had to train on the Training Data, as well as to potentially to see if there was an effect of overtraining and at what kind of epoch count this would be seen at.

Additionally, I did the same experiment with two different transfer functions being used in the hidden/output nodes: a regular Sigmoid function and a Tanh function. The purpose of this was to identify which activation function responded better to being trained, what number of epochs for each function produced the most optimum ANNs, or if one was simply a better function than the other to predict correct outputs.

To ensure that this experiment was valid, the number of hidden nodes for every ANN generated was limited to 5 and all modifications were disabled.



Epoch Experiment Graph depicting the Number of Epochs trained vs the RMSE of the resulting ANN

After conducting this experiment and plotting the graph above, two interesting points emerged.

Firstly, ANNs that used Tanh as the activation function for the hidden/output nodes could initially predict values at a far greater accuracy than the traditional Sigmoid function ANNs after having only 1 epoch on the data (to ensure this was not a fluke/anomaly, multiple sets of each were generated and the difference in RMSE values between them remained consistent).

Secondly, after around 5000 epochs through the data, the effectiveness of training the ANNs began to diverge between using Tanh and the Sigmoid function. Sigmoid responded better with being trained for longer than Tanh, scoring lower RMSE values constantly at higher epoch counts.

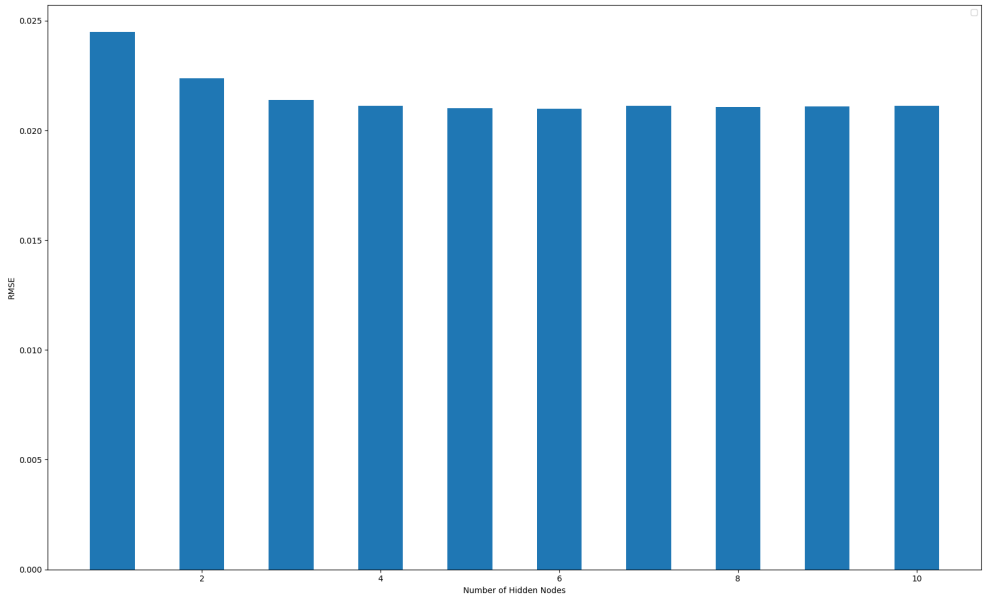
The effect of potential overtraining was not seen however despite training the ANNs up to 32,768 epochs. Maybe this would be seen at later epoch limits?

Effect of Different Numbers of Hidden Nodes:

As the second of the initial two experiments, this had a similar motive as the Epoch Experiment but instead focused on the number of Hidden Nodes used. Each ANN was given 5000 epochs to cycle through the data (as the Epoch Experiment seemed to yield

similar/equivalent results for either transfer function at that epoch count) but increasing numbers of hidden nodes (i.e. the first 20 generated had 1 node, the next 20 had 2 nodes, etc.).

Again, for the sake of ensuring no interference and invalidation of the experiment, all modifications to the Backpropagation Algorithm were disabled.



Bar Chart depicting the effect of increasing the number of Hidden Nodes on an ANN

After generating the different ANN configurations and plotting their performance against the Testing Set, a trend started to emerge in that networks with more hidden nodes tended to be able to predict values to a higher degree of accuracy than those with lower hidden node counts. The shape of this trendline was somewhat similar to that of the Epoch Experiment in that, at lower hidden node counts, adding another hidden node had a large effect in the effectiveness of the generated ANNs however beyond around 5 hidden nodes, having more had little effect.

Interestingly, in this experiment, the network with the lowest RMSE value was the one with 6 hidden nodes at 0.0210 RMSE (4 d.p). However, seeing as the difference in effectiveness with higher hidden node counts was so small, I think this was a statistical fluke.

Observing the Modifications:

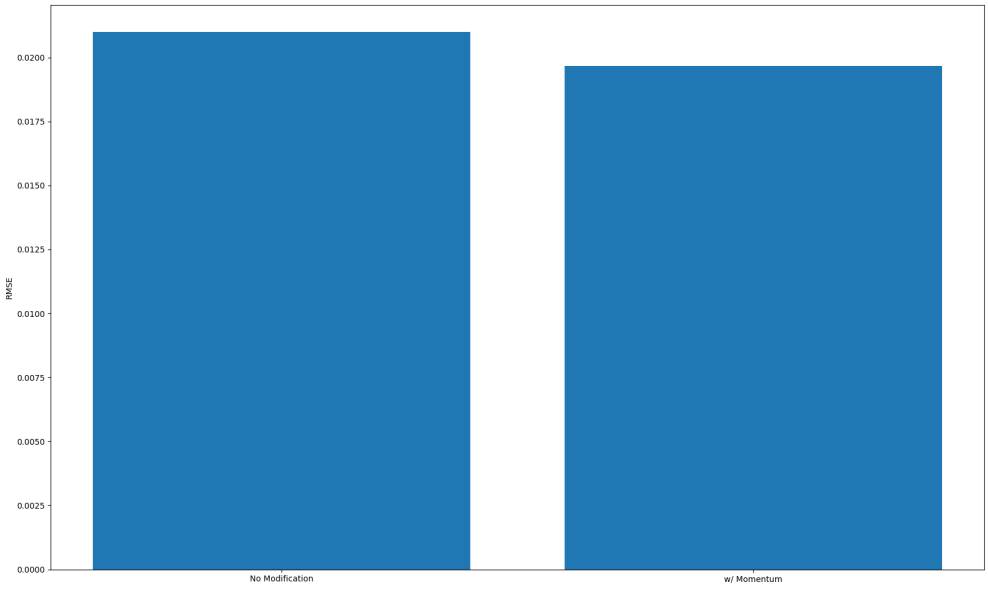
After observing the effect of changing the core aspects of an ANN (i.e. the amount of epochs trained and the amount of hidden nodes), I examined how much of an impact each of the

modifications to the Backpropagation algorithm had on the effectiveness of the resulting neural networks produced.

For each the below experiments, every ANN produced was given 5 hidden nodes, 5000 epochs and made to use the Sigmoid transfer function to ensure that only the chosen modification was the independent variable being explored and nothing else was influencing the result.

Note: *For ease of comparison in this report, the unmodified ANN consistently predicted values with a RMSE of 0.02100 (5 d.p).*

The Effect of Momentum:

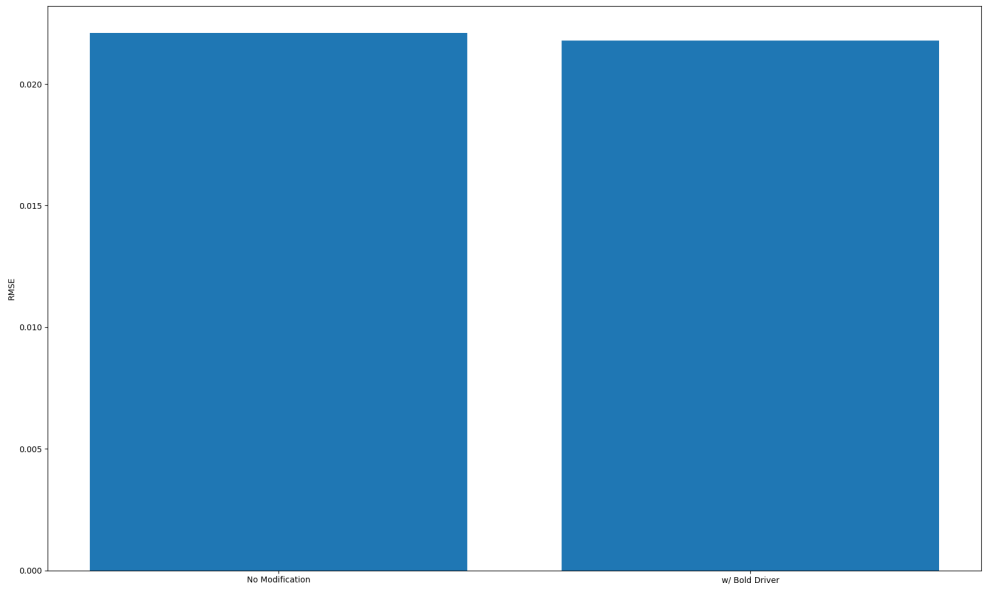


Bar Chart depicting the effect of an ANN having Momentum enabled/disabled

As can be seen in the graph above, Momentum had quite a significant effect on the effectiveness of an ANN (i.e. how quickly the ANN could find the global error minima). With Momentum, the RMSE value dropped to 0.01967 (5 d.p.) meaning that with the same number of epochs trained and Momentum enabled, produced ANNs were 6.33% (3 s.f.) more effective than their unmodded counterpart.

The Effect of Bold Driver:

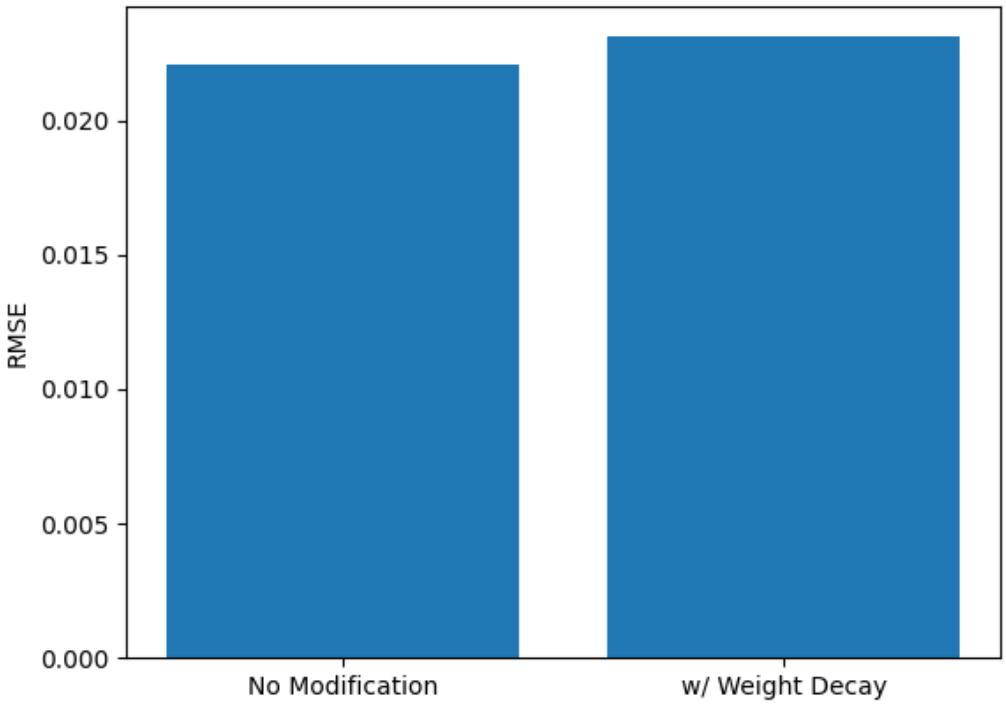
With Bold Driver enabled, even though ANNs produced were slightly more effective, the impact was far smaller than that of Momentum. With Bold Driver, ANNs produced had an RMSE value of 0.02077 (5 d.p) meaning that the modification only had the slight influence of improving ANNs by around 1.10% (3 s.f.).



Bar Chart depicting the effect of an ANN having Bold Driver enabled/disabled

The Effect of Weight Decay:

When plotting the effectiveness of Weight Decay being used on the production of ANNs, interestingly the modification had an adverse impact on the effectiveness of the neural networks produced. With Weight Decay, generated networks had an RMSE value of 0.02312 (5 d.p.) therefore showing that the modification caused ANNs to become 10.1% (3 s.f.) less effective.

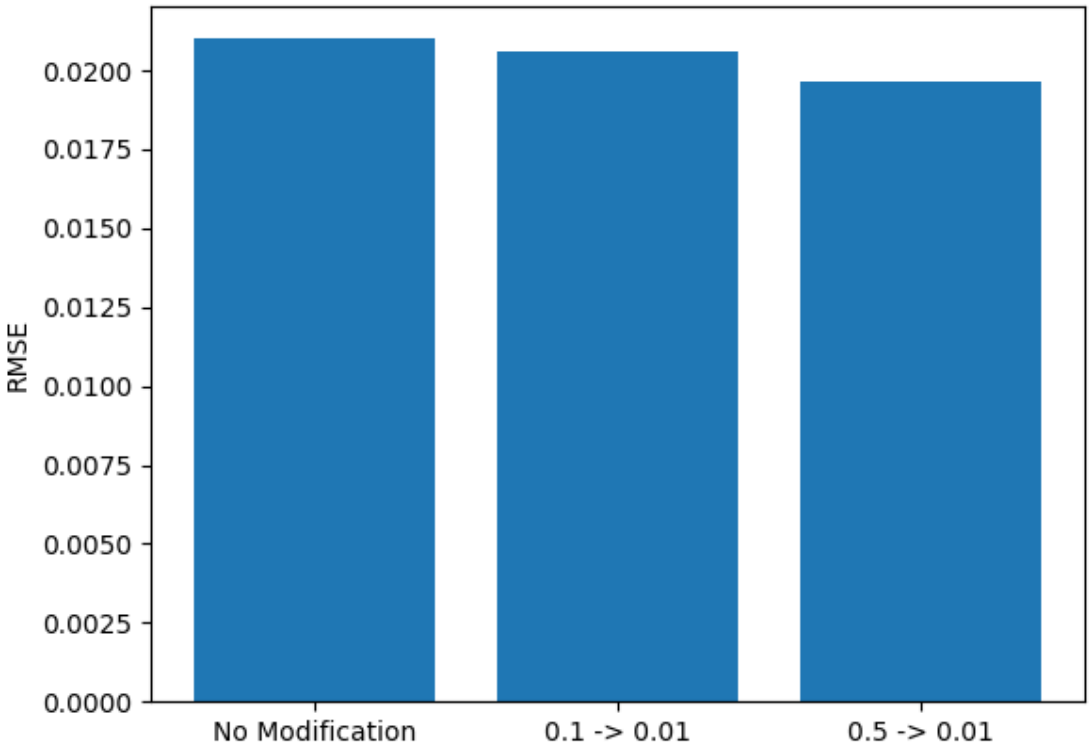


Bar Chart depicting the effect of an ANN having Weight Decay enabled/disabled

The Effect of Annealing:

With the modification of Annealing, seeing as the start values and end values for ρ were variable, I wanted to see if the range of values ρ could impact the algorithm into finding global minima much easier (the idea being that an initial large ρ could help initially 'scan' the error function than as ρ becomes smaller, it could 'fall' into the minimum value).

As can be seen with the below graph, my idea seemed to hold as ANNs that had ρ start from 0.5 then gradually fall to 0.1 had an RMSE value of 0.01963 (5 d.p.) whereas those that had a ρ start from 0.1 instead had an RMSE value of 0.02062 (5 d.p.). This therefore meant that ANNs that had a larger range for the Annealing modification were 6.52% more effective than unmodded ANNs whereas those with a smaller Annealing range were only marginally better by 1.81%.

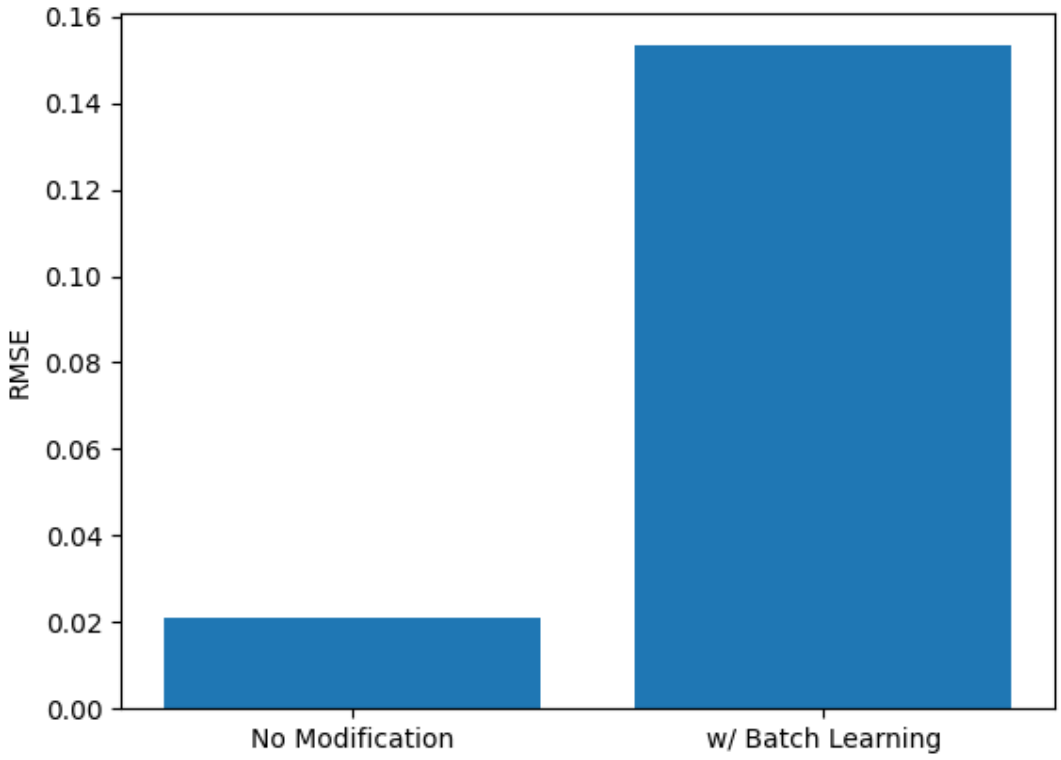


Bar Chart depicting the effect of an ANN having Annealing enabled at different limits

The Effect of Batch Learning:

With regards to Batch Learning, since the modification is less about trying to find the global minima more quickly but rather to 'stabilise' the weight changes, its impact in creating an ANN more effective than its unmodified counterpart would be negligible or even counter productive as weight changes were instead limited to once an epoch rather than after every data point. Since the number of times weights would change has decreased for the modified ANN, it could not have been 'trained' enough after 5000 epochs.

As can be seen with the below graph, the 'Batch Learning' ANN did suffer from not being able to modify the weights as many times as the 'normal' ANN. After attempting to predict the Testing Set, it resulted with an RMSE value of 0.15326 (5 d.p.) therefore meaning that it was 630% less effective than the unmodified network.



Bar Chart depicting the effect of an ANN having Batch Learning enabled/disabled

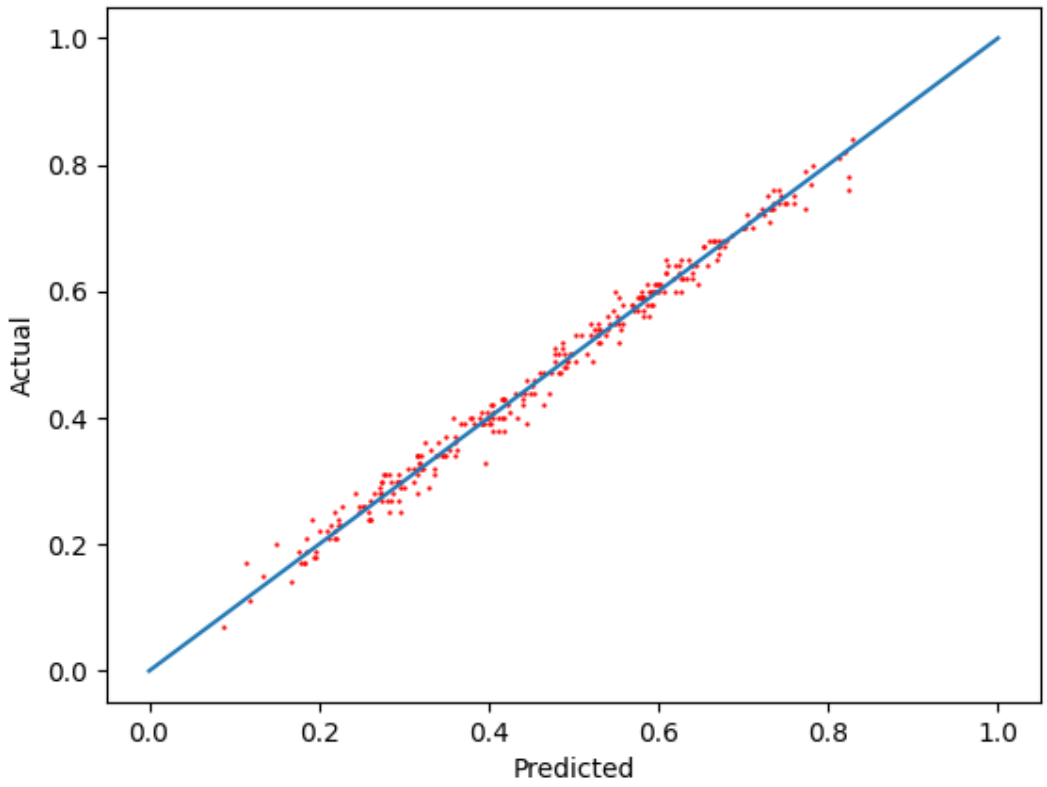
Section D: Evaluation of Final Network Model

After completing the experiments in the previous section, I concluded which modifications + how many epochs/hidden nodes would create the best possible Neural Network from the backpropagation algorithm.

After training 20 neural networks with the following settings:

- 32,768 epochs
- 6 hidden nodes
- Sigmoid Transfer Function
- Momentum Enabled
- Bold Driver Enabled
- Weight Decay/Batch Learning/Annealing disabled

I selected the one with the best RMSE value from predicting the Validation set to choose as my 'best produced' ANN (attained a value of 0.01861 to 5 decimal points). This ANN was then set to attempt to predict the Testing Set resulting in the following graph:



Scatter Graph depicting the accuracy of the best produced ANN. The closer the points are to the line, the more accurate the ANN.

For the sake of ease of interpretation, I had each point on the graph have the x coordinate of the value the ANN predicted for that data record in the Testing Set and the y coordinate being the actual value that data record had. After plotting a simple $y = x$ line on the graph (signifying perfect prediction), it can make it visually apparent how accurate the ANN was by how close each point falls to the line. To quantify this in a more statistical manner, I also used the *numpy.corrcoef()* function to get the correlation coefficient of the points to get a numerical understanding of how close the points were to the line. This came out to a value of 0.99410 to 5 decimal places, which meant that the points were almost at a perfect positive correlation and thus predicted values were mostly representative of their actual counterparts.

As well as plotting and analysing the above graph, I also calculated the Root Mean Squared Error, Coefficient of Efficiency and Mean Squared Relative Error of the ANN when performed on the Testing Set. These, respectively, came to:

$$RMSE = 0.01893 \text{ (5 d.p.)}$$
$$CE = 0.98795 \text{ (5 d.p.)}$$

$$MSRE = 0.00340 \text{ (5 d.p.)}$$

These values showed that the ANN I had produced had an almost perfect model of the unseen Testing Data (i.e. $CE \approx +1$) as well as having an exceptionally low average real and relative error.

Overall, I felt confident that the ANN produced was an effective model for predicting unseen data and therefore a success.

Note: *The ANN can be seen in the BestANN.json file where the configuration for recreating the neural network is specified.*

Section E: Comparison with Other Statistical Models

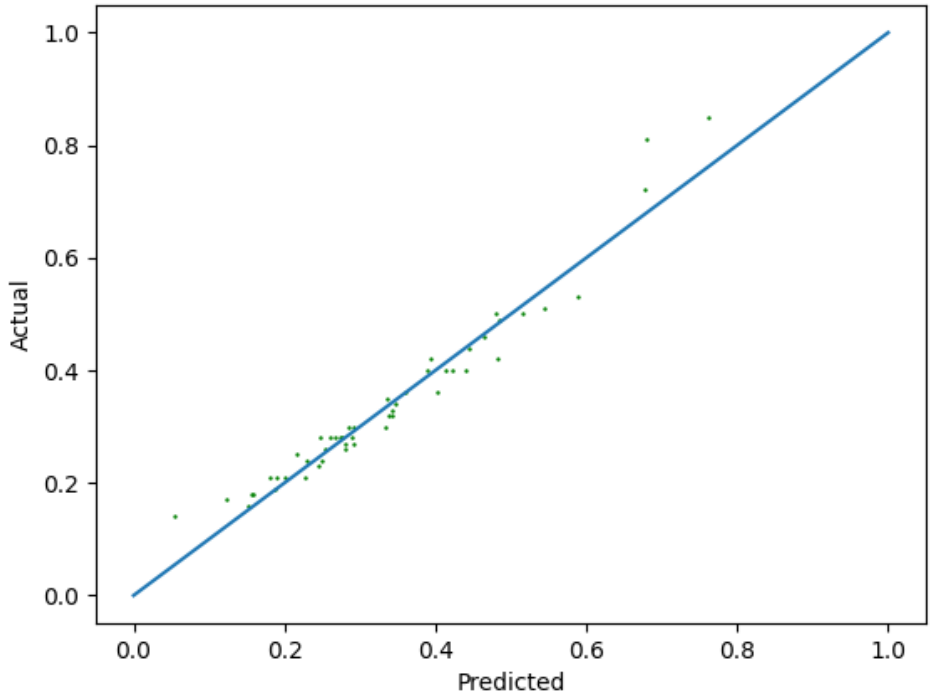
In order to know whether producing the last ANN was worth the effort of all of the previous experimentation, I chose to compare it to a simple LINEST statistical model in EXCEL.

(x5)	(x4)	(x3)	(x2)	(x1)	Constant	PanE (y value)
36	101.4	418.7	378.1	13.7	1	0.58815199

Screenshot Snippet of LINEST being used to predict Pan Evaporation with given inputs (x1 to x5)

In order to get predicted data from the LINEST function, I used part of the data set as known inputs and outputs for the function to approximate a trend then had it attempt to predict pan evaporation for unseen points in the data set and consequently record the predicted and actual values.

Upon doing this for approx. 50 unseen records, I stored the predicted values and the actual values in 2 1D arrays for Python to plot in a similar format as the graph in the previous section:



Scatter Graph depicting the accuracy of the LINEST Excel function

As well as this, I also calculated the same error functions to better evaluate the performance of the LINEST function (each to 5 decimal places):

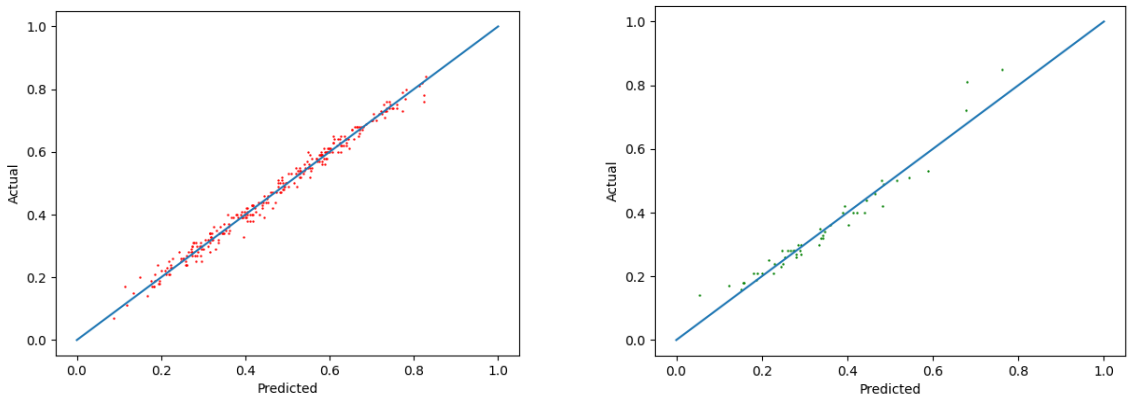
$$RMSE = 0.03457$$
$$CE = 0.96945$$
$$MSRE = 0.01456$$
$$\text{Correlation Coefficient} = 0.97417$$

This can then be compared in the table with the ANNs evaluation to identify if the ANN performed better than the basic Excel LINEST function:

	LINEST	**Neural Network**	**Did the ANN perform better? (Yes/No)**
Root Mean Squared Error (RMSE)	0.03457	0.01893	***Yes*** $(RMSE_{ANN} < RMSE_{LINEST})$
Coefficient of Efficiency (CE)	0.96945	0.98795	***Yes*** $(CE_{ANN} > CE_{LINEST})$

Mean Squared Relative Error (MSRE)	0.01456	0.00340	***Yes*** $(MSRE_{ANN} < MSRE_{LINEST})$
Correlation Coefficient	0.97417	0.99410	***Yes*** $(\rho_{ANN} > \rho_{LINEST})$

As can be seen, the ANN surpassed LINEST in all quantifiable metrics of accuracy from scoring lower on error measuring values like RMSE and MSRE to scoring higher on accuracy measuring values such as CE and the Correlation Coefficient ρ. This can also be seen visually by comparing the spread of the points on the graphs.



Comparison of the two scatter graphs produced by the ANN and LINEST. As can be seen, the ANN produces a much more accurate prediction model.

In conclusion, the ANN produced more accurate predictions for unknown outputs than the LINEST Excel function had done therefore showing that Neural Network was a successful model of Pan Evaporation and thus worth the time of developing rather than using a simple statistical model in its place.